Module -1

Enumerations, Autoboxing, and Annotations(Metadata)

Enumerations

- Enumerations included in JDK 5. An enumeration is a list of named constants. It is similar to final variables.
- **Enum in java** is a data type that contains fixed set of constants.
- An enumeration defines a class type in Java. By making enumerations into classes, so it can have constructors, methods, and instance variables.
- An enumeration is created using the enum keyword.

  Ex:

  enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland }

  The identifiers Jonathan, GoldenDel, and so on, are called enumeration constants.
  Each is implicitly declared as a public, static final member of Apple.

  Enumeration variable can be created like other primitive variable. It does not use the new for creating object.
  Ex:Apple ap;

  Ap is of type Apple, the only values that it can be assigned (or can contain) are those defined by the enumeration. For example, this assigns:

  ap = Apple.RedDel;

  <u>**Example Code-1**</u>

```
enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland }

class EnumDemo
{
public static void main(String args[])
{
Apple ap;
ap = Apple.RedDel;
 System.out.println("Value of ap: " + ap);// Value of ap: RedDel
 ap = Apple.GoldenDel;
 if(ap == Apple.GoldenDel)
 System.out.println("ap contains GoldenDel.\n"); // ap contains GoldenDel.
 switch(ap)
{
 case Jonathan:
 System.out.println("Jonathan is red.");
 break;
 case GoldenDel:
 System.out.println("Golden Delicious is yellow.");// Golden Delicious is yellow
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

```
 break;
case RedDel:
System.out.println("Red Delicious is red.");
break;
case Winesap:
System.out.println("Winesap is red.");
break;
 case Cortland:
System.out.println("Cortland is red.");
 break;
 }
}
}
```

The values( ) and valueOf( ) Methods All enumerations automatically contain two predefined methods: values( ) and valueOf( ).

Their general forms are shown here:

public static enum-type[ ] values( )

public static enum-type valueOf(String str)

The values( ) method returns an array that contains a list of the enumeration constants.

The valueOf( ) method returns the enumeration constant whose value corresponds to the string passed in str.

Example Code-2:

```
enum Season { WINTER, SPRING, SUMMER, FALL } ;

class EnumExample1{

public static void main(String[] args) {

for (Season s : Season.values())

        System.out.println(s);

        Season s = Season.valueOf("WINTER");

        System.out.println("S contains " + s);

}       }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

Example Code-3

```
class EnumExample5{

enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDA
Y, SATURDAY}

public static void main(String args[]){

Day day=Day.MONDAY;

  switch(day){

case SUNDAY:

 System.out.println("sunday");

 break;

case MONDAY:

 System.out.println("monday");

 break;

default:

System.out.println("other day");

}

}}
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

Class Type Enumeration

```
enum Apple {

 Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

private int price;

 Apple(int p) { price = p; }

int getPrice() { return price; }

}
```

```
class EnumDemo3 {

public static void main(String args[])

{ Apple ap;

System.out.println("Winesap costs " + Apple.Winesap.getPrice() + " cents.\n");

System.out.println("All apple prices:");

for(Apple a : Apple.values())

 System.out.println(a + " costs " + a.getPrice() + " cents.");

}

}
```

The Class type enumeration contains three things

The first is the instance variable price, which is used to hold the price of each variety of apple.

The second is the Apple constructor, which is passed the price of an apple.

The third is the method getPrice( ), which returns the value of price.

When the variable ap is declared in main( ), the constructor for Apple is called once for each constant that is specified.

the arguments to the constructor are specified, by putting them inside parentheses after each constant, as shown here:

Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);

These values are passed to the parameter of Apple(),which then assigns this value to price. The constructor is called once for each constant.

Because each enumeration constant has its own copy of price, you can obtain the price of a specified type of apple by calling getPrice().

For example, in main() the price of a Winesap is obtained by the following call:Apple.Winesap.getPrice()

Enum Super class

All enumerations automatically inherit one: java.lang.Enum.

Enum class defines several methods that are available for use by all enumerations.

ordinal( )

To obtain a value that indicates an enumeration constant's position in the list of constants. This is called its ordinal value, and it is retrieved by calling the ordinal( ) method, shown here:

final int ordinal( )

It returns the ordinal value of the invoking constant. Ordinal values begin at zero. Thus, in the Apple enumeration, Jonathan has an ordinal value of zero, GoldenDel has an ordinal value of 1, RedDel has an ordinal value of 2, and so on.

compareTo( )

To compare the ordinal value of two constants of the same enumeration by using the compareTo( ) method. It has this general form:

final int compareTo(enum-type e)

equals()

equals method is overridden method from Object class, it is used to compare the enumeration constant. Which returns true if both constants are same.

Program to demonstrate the use of ordinal(), compareTo(), equals()

```
enum Apple { Jonathan, GoldenDel, RedDel, Winesap, Cortland }

class EnumDemo4

{ public static void main(String args[])

{ Apple ap, ap2, ap3;

System.out.println("Here are all apple constants" + " and their ordinal values: ");

 for(Apple a : Apple.values())

System.out.println(a + " " + a.ordinal());
```

```
ap =  Apple.RedDel;

ap2 = Apple.GoldenDel;

ap3 = Apple.RedDel;

System.out.println();

if(ap.compareTo(ap2) < 0) System.out.println(ap + " comes before " + ap2);

if(ap.compareTo(ap2) > 0) System.out.println(ap2 + " comes before " + ap);

if(ap.compareTo(ap3) == 0) System.out.println(ap + " equals " + ap3);

System.out.println();

if(ap.equals(ap2)) System.out.println("Error!");

if(ap.equals(ap3)) System.out.println(ap + " equals " + ap3);

if(ap == ap3) System.out.println(ap + " == " + ap3);

}

}
```

Wrappers Classes

Java uses primitive types such as int or double, to hold the basic data types supported by the language.

The primitive types are not part of the object hierarchy, and they do not inherit Object.

Despite the performance benefit offered by the primitive types, there are times when you will need an object representation.

Many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types.

To handle the above situation, Java provides type wrappers, which are classes that encapsulate a primitive type within an object.

The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Character:

Character is a wrapper around a char. The constructor for Character is

Character(char ch)

Here, ch specifies the character that will be wrapped by the Character object being created.

To obtain the char value contained in a Character object, call charValue( ), shown here:

char charValue( )

Boolean:

Boolean is a wrapper around boolean values. It defines these constructors:

Boolean(boolean boolValue)

Boolean(String boolString)

In the first version, boolValue must be either true or false.

In the second version, if boolString contains the string “true” (in uppercase or lowercase), then the new Boolean object will be true. Otherwise, it will be false.

To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:

boolean booleanValue( )

It returns the boolean equivalent of the invoking object.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

Integer Wrapper class example code:

Integer(int num)

Integer(String str)

```
class Wrap

{ public static void main(String args[])

 {

  Integer iOb = new Integer(100);
```

```
   int i = iOb.intValue();

   System.out.println(i + " " + iOb);

 }

}
```

This program wraps the integer value100 inside an Integer object called iOb.

The program then obtains this value by calling intValue() and stores the result in i.

The process of encapsulating a value within an object is called boxing.

Thus, in the program, this line boxes the value 100 into an Integer:

Integer iOb = new Integer(100);

The process of extracting a value from a type wrapper is called unboxing.

The program unboxes the value in iOb with this statement:

int i = iOb.intValue();

AutoBoxing

Auto boxing is the process by which a primitive type is automatically encapsulated(boxed) into its equivalent type wrapper

whenever an object of that type is needed. There is no need to explicitly construct an object.

```
Integer iOb = 100; // autobox an int
```

Auto-unboxing

Auto- unboxing is the process by which the value of a boxed object is automatically extracted from a type wrapper when it is assigned to primitive type value is needed.

There is no need to call a method such as intValue( ).

```
int i = iOb; // auto-unbox
```

Example Program:

```
class AutoBoxUnBox
{
 public static void main(String args[]) {
Integer iOb = 100; // autobox an int
int i = iOb; // auto-unbox
System.out.println(i + " " + iOb);  // displays 100 100
}
}
```

Explain auto boxing and auto unboxing during method call

```
class AutoBox2 {
static int m(Integer v)
{ return v ; }
public static void main(String args[]) {
Integer iOb = m(100);
System.out.println(iOb);// 100
}
}
```

In the program, notice that m( ) specifies an Integer parameter and returns an int result.

Inside main( ), m( ) is passed the value 100.

Because m( ) is expecting an Integer, this value is automatically boxed.

Then, m( ) returns the int equivalent of its argument. This causes v to be auto-unboxed.

Next, this int value is assigned to iOb in main( ), which causes the int return value to be autoboxed.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

Explain auto boxing and unboxing during expression evaluation

Autoboxing/Unboxing Occurs in Expressions autoboxing and unboxing take place whenever a conversion into an object or from an object is required.

This applies to expressions. Within an expression, a numeric object is automatically unboxed.

The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
class AutoBox3

{ public static void main(String args[]) {

Integer iOb, iOb2; int i;

iOb = 100;

System.out.println("Original value of iOb: " + iOb);

 ++iOb; // auto unbox and rebox

System.out.println("After ++iOb: " + iOb);

iOb2 = iOb + (iOb / 3);

 System.out.println("iOb2 after expression: " + iOb2);

i = iOb + (iOb / 3); // auto unbox and rebox

System.out.println("i after expression: " + i);

}

}
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

```
++iOb;
```

This causes the value in iOb to be incremented.

It works like this: iOb is unboxed, the value is incremented, and the result is reboxed.

Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed,the standard type promotions and conversions are applied.For example, the following program is perfectly valid:

```
Integer iOb = 100; Double dOb = 98.6;

dOb = dOb + iOb; // type promoted to double
```

```
System.out.println("dOb after expression: " + dOb);

Integer iOb = 2;

switch(iOb) {

case 1: System.out.println("one");

          break;

case 2: System.out.println("two");

         break;

default:

     System.out.println("error");

}
```

When the switch expression is evaluated, iOb is unboxed and its int value is obtained.

As the examples in the program show, because of autoboxing/unboxing, using numeric objects in an expression is both intuitive and easy.

<u>Autoboxing/unboxing a Boolean and Character.</u>

```
class AutoBox5 { public static void main(String args[]) {

      Boolean b = true; // auto boxing boolean

     if(b)

     System.out.println("b is true");// auto unboxed when used in conditional expression

      Character ch = 'x'; // box a char

     char ch2 = ch; // unbox a char

     System.out.println("ch2 is " + ch2);

     }}
```

The output is shown here:

b is true ch2 is x

Annotations

Annotations (Metadata) Beginning with JDK 5, a new facility was added to Java that enables you to embed supplemental information into a source file.

This information, called an annotation, does not change the actions of a program. Thus, an annotation leaves the semantics of a program unchanged.

However this information can be used by various tools during both development and deployment.

For example, an annotation might be processed by a source-code generator. The term metadata is also used to refer to this feature, but the term annotation is the most descriptive and more commonly used.

An annotation is created through a mechanism based on the interface. Let's begin with an example.

Here is the declaration for an annotation called MyAnno:

@interface MyAnno { String str(); int val(); }

@ that precedes the keyword interface.

This tells the compiler that an annotation type is being declared.

Next, notice the two members str( ) and val( ).

All annotations consist solely of method declarations.

However, you don't provide bodies for these methods.

Instead, Java implements these methods.

Moreover, the methods act much like fields.

An annotation cannot include an extends clause.

MyAnno(str = "Annotation Example", val = 100)

public static void myMeth() { // ...

Notice that no parentheses follow str in this assignment.

What is retention policy ? Explain the use of retention tag.

- A retention policy determines at what point an annotation is discarded.
- Java defines three such policies, which are encapsulated within the java.lang.annotation.
- RetentionPolicy enumeration.

They are SOURCE, CLASS, and RUNTIME.

- An annotation with a retention policy of SOURCE is retained only in the source file and is discarded during compilation.
- An annotation with a retention policy of CLASS is stored in the .class file during compilation. However, it is not available through the JVM during run time.
- An annotation with a retention policy of RUNTIME is stored in the .class file during compilation and is available through the JVM during run time.

A retention policy is specified for an annotation by using one of Java's built-in annotations: @Retention.

- @Retention(retention-policy)

Here, retention-policy must be one of the previously discussed enumeration constants.

If no retention policy is specified for an annotation, then the default policy of CLASS is used.

The following version of MyAnno uses @Retention to specify the RUNTIME retention policy.

Thus, MyAnno will be available to the JVM during program execution.

```
@Retention(RetentionPolicy.RUNTIME)

@interface MyAnno { String str(); int val(); }
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

```
import java.lang.annotation.*;

import java.lang.reflect.*;

// An annotation type declaration.

 @Retention(RetentionPolicy.RUNTIME)

@interface MyAnno { String str(); int val(); }

class Meta {

// Annotate a method.
```

```
@MyAnno(str = "Annotation Example", val = 100)

public static void myMeth()

{ Meta ob = new Meta();

try {

// First, get a Class object that represents // this class.

Class c = ob.getClass();

// Now, get a Method object that represents // this method.

Method m = c.getMethod("myMeth");

// Next, get the annotation for this class.

MyAnno anno = m.getAnnotation(MyAnno.class);

System.out.println(anno.str() + " " + anno.val()); }

catch (NoSuchMethodException exc)

{ System.out.println("Method Not Found."); }

}

public static void main(String args[]) { myMeth(); }

}
```

The output from the program is shown here:

Annotation Example 100

This program uses reflection as described to obtain and display the values of str and val in the MyAnnoannotation associated with myMeth()in the Metaclass.

MyAnno anno = m.getAnnotation(MyAnno.class);

notice the expression MyAnno.class. This expression evaluates to a Class object of type MyAnno, the annotation.

This construct is called a class literal. You can use this type of expression whenever a Class object of a known class is needed.

However, to obtain a method that has parameters, you must specify class objects representing the types of those parameters as arguments to getMethod( ). For example, here is a slightly different version of the preceding program:

```
import  java.lang.annotation.*;

import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)

@interface MyAnno { String str(); int val(); }

class Meta {

// myMeth now has two arguments.

@MyAnno(str = "Two Parameters", val = 19)

 public static void myMeth(String str, int i)

 { Meta ob = new Meta();

try { Class c = ob.getClass();

// Here, the parameter types are specified.

Method m = c.getMethod("myMeth", String.class, int.class);

MyAnno anno = m.getAnnotation(MyAnno.class);

System.out.println(anno.str() + " " + anno.val()); }

catch (NoSuchMethodException exc)

{ System.out.println("Method Not Found."); }

}

public static void main(String args[])

 { myMeth("test", 10); }

}
```

The output from this version is shown here:

Two Parameters 19

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

myMeth( ) takes a String and an int parameter.

To obtain information about this method, getMethod( ) must be called as shown here:

Method m = c.getMethod("myMeth", String.class, int.class);

Here, the Class objects representing String and int are passed as additional arguments.

Obtaining All Annotations

You can obtain all annotations that have RUNTIME retention that are associated with an item by calling getAnnotations( ) on that item.

It has this general form:

Annotation[ ] getAnnotations( )

It returns an array of the annotations.

getAnnotations( ) can be called on objects of type Class, Method, Constructor, and Field. Here is another reflection example that shows how to obtain all annotations associated with a class and with a method.

It declares two annotations.

It then uses those annotations to annotate a class and a method.

Example code:

```
import java.lang.annotation.*; import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)

@interface MyAnno { String str(); int val(); }

@Retention(RetentionPolicy.RUNTIME)

@interface What { String description(); }

@What(description = "An annotation test class")

 @MyAnno(str = "Meta2", val = 99) class Meta2 {

@What(description = "An annotation test method")

 @MyAnno(str = "Testing", val = 100)

 public static void myMeth() { Meta2 ob = new Meta2();

try { Annotation annos[] = ob.getClass().getAnnotations();
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
// Display all annotations for Meta2.

System.out.println("All annotations for Meta2:");

 for(Annotation a : annos) System.out.println(a);

System.out.println();

// Display all annotations for myMeth.

 Method m = ob.getClass( ).getMethod("myMeth");

 annos = m.getAnnotations();

System.out.println("All annotations for myMeth:");

 for(Annotation a : annos)

System.out.println(a);

} catch (NoSuchMethodException exc) { System.out.println("Method Not Found."); }

}

public static void main(String args[]) { myMeth(); }

}
```

The output is shown here:

```
 All annotations for Meta2:

 @What(description=An annotation test class)

 @MyAnno(str=Meta2, val=99)

All annotations for myMeth:

@What(description=An annotation test method)

@MyAnno(str=Testing, val=100)
```

The program uses getAnnotations( ) to obtain an array of all annotations associated with the Meta2 class and with the myMeth( ) method. As explained, getAnnotations( ) returns an array of Annotation objects.

Recall that Annotation is a super-interface of all annotation interfaces and that it overrides toString( ) in Object.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

Thus, when a reference to an Annotation is output, its toString( ) method is called to generate a string that describes the annotation, as the preceding output shows.

The AnnotatedElement Interface

The methods declared in AnnotatedElement Interface

1. getAnnotation( ) --- It can be invoked with method, class. It return the used annotation.
2. getAnnotations( ) --- It can be invoked with method, class. It return the used annotations.
3. getDeclaredAnnotations( ) -- It returns all non-inherited annotations present in the invoking object.
4. isAnnotationPresent( ), which has this general form:
   It returns true if the annotation specified by annoType is associated with the invoking object. It returns false otherwise.

Default Values in annotation

You can give annotation members default values that will be used if no value is specified when the annotation is applied.

A default value is specified by adding a default clause to a member's declaration. It has this general form:

type member( ) default value;

// An annotation type declaration that includes defaults.

@interface MyAnno { String str() default "Testing"; int val() default 9000; }

@MyAnno() // both str and val default

@MyAnno(str = "some string") // val defaults

@MyAnno(val = 100) // str defaults

@MyAnno(str = "Testing", val = 100) // no defaults

Example:

```
import java.lang.annotation.*;

import java.lang.reflect.*;

 @Retention(RetentionPolicy.RUNTIME)
```

```
@interface MyAnno { String str() default "Testing"; int val() default 9000; }

class Meta3 {

 @MyAnno()

public static void myMeth() { Meta3 ob = new Meta3();

try { Class c = ob.getClass();

Method m = c.getMethod("myMeth");

MyAnno anno = m.getAnnotation(MyAnno.class);

System.out.println(anno.str() + " " + anno.val()); }

catch (NoSuchMethodException exc)

 {

System.out.println("Method Not Found."); }

}

public static void main(String args[]) { myMeth(); }

}
```

Output:

Testing 9000

Marker Annotations

A marker annotation is a special kind of annotation that contains no members.

Its sole purpose is to mark a declaration. Thus, its presence as an annotation is sufficient.

The best way to determine if a marker annotation is present is to use the method isAnnotationPresent( ), which is a defined by the AnnotatedElement interface.

Here is an example that uses a marker annotation.

Because a marker interface contains no members, simply determining whether it is present or absent is sufficient.

```
import java.lang.annotation.*;

import java.lang.reflect.*;
```

```
@Retention(RetentionPolicy.RUNTIME)

@interface MyMarker { }

class Marker {

@MyMarker

public static void myMeth() { Marker ob = new Marker();

try { Method m = ob.getClass().getMethod("myMeth");

if(m.isAnnotationPresent(MyMarker.class))

System.out.println("MyMarker is present.");

} catch (NoSuchMethodException exc)

{ System.out.println("Method Not Found."); }

}

public static void main(String args[]) { myMeth(); }

}
```

Output

MyMarker is present.

```
public static void main(String args[]) { myMeth(); }

}
```

Built in Annotations

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Built-In Java Annotations used in java code

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations

- @Target
- @Retention
- @Inherited
- @Documented

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.Sometimes, we does the silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

Example:
```
class Animal{
void eatSomething()
{System.out.println("eating something");}
}
class Dog extends Animal{
@Override
void eatsomething()
{
System.out.println("eating foods");
}//Compile time error  }
```

@SuppressWarnings

annotation: is used to suppress warnings issued by the compiler.

If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time because we are using non-generic collection.

```
import java.util.*;

class TestAnnotation2{

@SuppressWarnings("unchecked")

public static void main(String args[]){

ArrayList list=new ArrayList();

list.add("sonoo");

}}
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

@Deprecated

@Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions.

So, it is better not to use such methods.

```
class A{
void m(){System.out.println("hello m");}
@Deprecated
void n(){System.out.println("hello n");}
}
 class TestAnnotation3{
   public static void main(String args[]){
   A a=new A();
   a.n();
   }}
```

Error message: `Test.java uses or overrides a deprecated API.`

@Inherited

is a marker annotation that can be used only on another annotation declaration. Furthermore, it affects only annotations that will be used on class declarations. @Inherited causes the annotation for a superclass to be inherited by a subclass.

```
@Inherited
public @interface MyCustomAnnotation {
}
@MyCustomAnnotation
public class MyParentClass {
  ...
}
public class MyChildClass extends MyParentClass {
   ...
}
```

Here the class MyParentClass is using annotation @MyCustomAnnotation which is marked with @inherited annotation. It means the sub class MyChildClass inherits the @MyCustomAnnotation.

CAMBRIDGE (SOURCE DIGINOTES)

@Documented

@Documented annotation indicates that elements using this annotation should be documented by JavaDoc.

```
@Documented
public @interface MyCustomAnnotation {
  //Annotation body
}
@MyCustomAnnotation
public class MyClass {
     //Class body
}
```

While generating the javadoc for class MyClass, the annotation @MyCustomAnnotation would be included in that

@Target

It specifies where we can use the annotation.

For example: In the below code, we have defined the target type as METHOD which means the below annotation can only be used on methods.

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.METHOD})
public @interface MyCustomAnnotation {

}
public class MyClass {
   @MyCustomAnnotation
   public void myMethod()
   {
       //Doing something
   }
}
```

If you do not define any Target type that means annotation can be applied to any element.

@Retention is mention in earlier topic.

BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

Module 2-The collections and Framework

1. **Explain brief about collection frame work.**
    - The Java Collections Framework standardizes the way in which groups of objects are handled by your programs.
    - The framework had to be high-performance.
    - The implementations for the fundamental collections(dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
    - The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
    - Extending and/or adapting a collection had to be easy.
    - Mechanisms were added that allow the integration of standard arrays into the Collections Framework.
    - Algorithms are another important part of the collection mechanism.
    - Algorithms operate on collections and are defined as static methods within the Collections class.
    - An iterator offers a general-purpose, standardized way of accessing the elements within a collection, one at a time. Thus, an iterator provides a means of enumerating the contents of a collection.
    - Because each collection implements Iterator, the elements of any collection class can be accessed through the methods defined by Iterator.

2. **What are the recent changes to collection framework?**

Recently, the Collections Framework underwent a fundamental change that significantly increased its power and streamlined its use. The changes were the addition of generics, autoboxing/unboxing, and the for-each style for loop.

Generics

The addition of generics caused a significant change to the Collections Framework because the entire Collections Framework has been reengineered for it. All collections are now generic, and many of the methods that operate on collections take generic type parameters Generics add the one feature that collections had been missing: type safety.

Prior to generics, all collections stored Object references, which meant that any collection could store any type of object. Thus, it was possible to accidentally store in compatible type s in a collection.

Doing so could result in run-time type mismatch errors. With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.

Autoboxing/unboxing

Autoboxing facilitates the Use of Primitive Types.

Autoboxing/unboxing facilitates the storing of primitive types in collections.

As you will see, a collection can store only references, not primitive values. In the past, if you wanted to store a primitive value, such as an int, in a collection, you had to manually box it into its type wrapper.

Source diginotes.in

When the value was retrieved, it needed to be manually unboxed (by using an explicit cast) into its proper primitive type.

Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. There is no need to manually perform these operations.

The For-Each Style for Loop

collection can be cycled through by use of the for-each style for loop.

Earlier it was done with Iteratable interface. For each loop is easier than the earlier iterator.

3. **List the Collection Interfaces?**
   - The Collections Framework defines several interfaces. This section provides an overview of each interface. Collection enables you to work with groups of objects; it is at the top of the collections hierarchy.
   - Deque extends Queue to handle a double-ended queue.
   - List extends Collection to handle sequences
   - NavigableSet extends SortedSet to handle retrieval of elements based on closest-match searches.
   - Queue extends Collection to handle special types of lists in which elements are removed only from the head.
   - Set extends Collection to handle sets, which must contain unique elements.
   - SortedSet extends Set to handle sorted sets.
4. **Give the syntax of collection interface. Explain the methods present in collection interface.**

interface Collection<E>

E specifies the type of objects that the collection

Collection extends the Iterable interface.

Iterating through the list cane be done through the iteratable interface.

Methods in collection interface

add

boolean add(E obj)

adds obj to the invoking collection.

Returns true if obj was added to the collection.

Returns false if obj is already a member of the collection and the collection does not allow duplicates.

addAll

boolean addAll(Collection<? extends E> c)

Adds all the elements of c to the invoking collection.

Returns true if the operation succeeded

(i.e., the elements were added). Otherwise, returns false.

Source diginotes.in

clear

void clear( )

Removes all elements from the invoking collection.

contains

boolean contains(Object obj)

Returns true if obj is an element of the invoking collection.

Otherwise, returns false.

containsAll

boolean containsAll(Collection<?> c)

Returns true if the invoking collection contains all elements of c. Otherwise, returns false.

equals

boolean equals(Object obj)

Returns true if the invoking collection and obj are equal.

Otherwise, returns false.

hashCode

int hashCode( ) Returns the hash code for the invoking collection.

isEmpty

boolean isEmpty( )

Returns true if the invoking collection is empty.

Otherwise, returns false.

iterator

Iterator<E> iterator( ) Returns an iterator for the invoking collection.

remove

boolean remove(Object obj)

Removes one instance of obj from the invoking collection.

Returns true if the element was removed. Otherwise, returns false.

removeAll

boolean removeAll(Collection<?> c)

Removes all elements of c from the invoking collection.

Source diginotes.in

Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

retainAll

boolean retainAll(Collection<?> c)

Removes all elements from the invoking collection except those in c.

Returns true if the collection changed (i.e., elements were removed). Otherwise, returns false.

size

int size( ) Returns the number of elements held in the invoking collection.

toArray

Object[ ] toArray( )

Returns an array that contains all the elements stored in the invoking collection.

The array elements are copies of the collection elements.

The array elements are copies of the collection elements.

If the size of array equals the number of elements, these are returned in array.

5. Explain the methods present in List interface.

List interface extends collection interface. It includes new method. Which are given below.

void add(int index , E obj)

Inserts obj into the invoking list at the index passed in index.

Any pre existing elements at or beyond the point of insertion are shifted up.

boolean addAll(int index , Collection<? extends E> c)

Inserts all elements of C into the invoking list at the index passed in

index . Any pre existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.

E get(int index)

Returns the object stored at the specified index within the invoking collection.

int indexOf(Object obj)

Source diginotes.in

Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, –1 is returned.

int lastIndexOf(Object obj)

Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, –1 is returned.

ListIterator<E> listIterator( )

Returns an iterator to the start of the invoking list.

ListIterator<E> listIterator(int index)

Returns an iterator to the invoking list that begins at the specified index.

E remove(int index)

Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.

E set(int index , E obj)

Assigns obj to the location specified by index within the invoking list.

List<E> subList(int start , int end)

Returns a list that includes elements from start to end –1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

6. **Explain Set Interface and set method:**

The Set interface defines a set.

It extends Collection and declares the behaviour of a collection that does not allow duplicate elements.

Therefore, the add() method returns false if an attempt.

Set is a generic interface that has this declaration:

interface Set<E>

Here, E specifies the type of objects that the set will hold.

The SortedSet Interface

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order.

SortedSet is a generic interface that has this declaration: interface SortedSet<E> Here, E specifies the type of objects that the set will hold.

In addition to those methods defined by Set, the SortedSet interface declares the methods.

Comparator<? super E> comparator( )

Returns the invoking sorted set's comparator.

Source diginotes.in

If the natural ordering is used for this set, null is returned.

E first( )

Returns the first element in the invoking sorted set.

SortedSet<E> headSet(E end)

Returns a SortedSet containing those elements less than end that are contained in the invoking sorted set.

Elements in the returned sorted set are also referenced by the invoking sorted set.

E last( )

Returns the last element in the invoking sorted set.

SortedSet<E> subSet(E start , E end)

Returns a SortedSet that includes those elements between start and end– 1. Elements in the returned collection are also referenced by the invoking object.

SortedSet<E> tailSet(E start)

Returns a SortedSet that contains those elements greater than or equal to start that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

Several methods throw a NoSuchElementException when no items are contained in the invoking set.

A ClassCastException is thrown when an object is in compatible with the elements in a set.

A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the set.

An IllegalArgumentException is thrown if an invalid argument is used.

7. NavigableSet Interface and method

The NavigableSet interface extends SortedSet and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.

NavigableSet is a generic interface that has this declaration:

interface NavigableSet<E>

Here, E specifies the type of objects that the set will hold.

NavigableSet adds the following

E ceiling(E obj)

Searches the set for the smallest element

If such an element is found, it is returned. Otherwise, null is returned.
Iterator<E> descendingIterator( )

Source diginotes.in

Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.

NavigableSet<E> descendingSet( )

Returns a NavigableSet that is the reverse of the invoking set. The resulting set is backed by the invoking set.

E floor(E obj)

Searches the set for the largest element e such that e <= obj . If such an element is found, it is returned. Otherwise, null is returned.

NavigableSet<E> headSet(E upperBound , boolean incl)

Returns a NavigableSet that includes all elements from the invoking set that are less than upperBound . If incl is true, then an element equal to upperBound is included. The resulting set is backed by the invoking set.

E higher(E obj) Searches the set for the largest element e such that e > obj . If such an element is found, it is returned. Otherwise, null is returned.

E lower(E obj)

Searches the set for the largest element e such that e < obj . If such an element is found, it is returned. Otherwise, null is returned.

E pollFirst( )

Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. null is returned if the set is empty.

E pollLast( )

Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. null is returned if the set is empty.

NavigableSet<E> subSet(E lowerBound , boolean lowIncl , E upperBound , boolean highIncl)

Returns a NavigableSet that includes all elements from the invoking set that are greater than lowerBound and less than upperBound .

If lowIncl is true, then an element equal to lowerBound is included.

If highIncl is true, then an element equal to upperBound is included.

The resulting set is backed by the invoking set.

NavigableSet<E> tailSet(E lowerBound , boolean incl)

Returns a NavigableSet that includes all elements from the invoking set that are greater than lowerBound . If incl is true, then an element equal to lowerBound is included. The resulting set is backed by the invoking set

8. The Queue Interface and methods

Source diginotes.in

The Queue interface extends Collection and declares the behaviour of a queue, which is often a first-in, first-out list. However, there are types of queues in which the ordering is based upon other criteria. Queue is a generic interface that has this declaration:

interface Queue<E>

E element( )

Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException if the queue is empty.

boolean offer(E obj)

Attempts to add obj to the queue. Returns true if obj was added and false otherwise.

E peek( )

Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed.

E poll( )

Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty.

E remove( )

Removes the element at the head of the queue, returning the element in the process. It throws NoSuchElementException if the queue is empty.

9. Deque interface

It extends Queue and declares the behavior of a double-ended queue.

Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks.

Deque is a generic interface that has this declaration:

interface Deque<E>

void addFirst(E obj)

Adds obj to the head of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.

void addLast(E obj)

Adds obj to the tail of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.

Iterator<E> descendingIterator( )

Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.

E getFirst( )

Returns the first element in the deque. The object is not removed from the deque. It throws NoSuchElementException if the deque is empty.

Source diginotes.in

E getLast( )

Returns the last element in the deque.

The object is not removed from the deque. It throws NoSuchElementException if the deque is empty.

boolean offerFirst(E obj)

Attempts to add obj to the head of the deque. Returns true if obj was added and false otherwise. Therefore, this method returns false when an attempt is made to add obj to a full, capacity-restricted deque.

boolean offerLast(E obj)

Attempts to add obj to the tail of the deque. Returns true if obj was added and false otherwise.

E peekFirst( )

Returns the element at the head of the deque. It returns null if the deque is empty. The object is not removed.

E peekLast( )

Returns the element at the tail of the deque. It returns null if the deque is empty. The object is not removed.

E pollFirst( )

Returns the element at the head of the deque, removing the element in the process. It returns null if the deque is empty.

E pollLast( ) Returns the element at the tail of the deque, removing the element in the process. It returns null if the deque is empty.

E pop( ) Returns the element at the head of the deque, removing it in the process. It throws NoSuchElementException if the deque is empty.

void push(E obj) Adds obj to the head of the deque. Throws an IllegalStateException if a capacity-restricted deque is out of space.

E removeFirst( ) Returns the element at the head of the deque, removing the element in the process. It throws NoSuchElementException if the deque is empty.

boolean removeFirstOccurrence(Object obj)

Removes the first occurrence of obj from the deque. Returns true if successful and false if the deque did not contain obj .

E removeLast( )

Returns the element at the tail of the deque, removing the element in the process. It throws NoSuchElementException if the deque is empty.

boolean removeLastOccurrence(Object obj)

Removes the last occurrence of obj from the deque. Returns true if successful and false if the deque did not contain

Source diginotes.in

10. The Collection Classes with example code

AbstractCollection
Implements most of the Collection interface.
AbstractList
Extends AbstractCollection and implements most of the List interface.
Queue interface.
AbstractSequentialList
Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
AbstractSet
Extends AbstractCollection and implements most of the Set interface.
EnumSet
Extends AbstractSet for use with enum elements.
HashSet
Extends AbstractSet for use with a hash table.
LinkedHashSet
Extends HashSet to allow insertion-order iterations.
PriorityQueue
Extends AbstractQueue to support a priority-based queue.
TreeSet Implements a set stored in a tree. Extends AbstractSet.
LinkedList Implements a linked list by extending AbstractSequentialList.
ArrayList Implements a dynamic array by extending AbstractList.
ArrayDeque Implements a dynamic double-ended queue by extending AbstractCollection and implementing the Deque interface.

ArrayList

ArrayList class extends **AbstractList** and implements the **List** interface

ArrayList is a generic class that has this declaration:

class ArrayList<E>

ArrayList has the constructors shown here:
ArrayList( )
constructor builds an empty array list
ArrayList(Collection<? extends E> *c*)
builds an array list that is initialized with the elements of the collection *c*.
ArrayList(int *capacity*)

builds an array list that has the specified initial *capacity*. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

```
class ArrayListDemo {
public static void main(String args[]) {
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " +
al.size());
```


Source diginotes.in

```
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " +
al.size());
System.out.println("Contents of al: " + al);
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " +
al.size());

}}
```

Converting ArrayList to Array

```
class ArrayListToArray {
public static void main(String args[]) {
ArrayList<Integer> al = new ArrayList<Integer>();
al.add(1);
al.add(2);
al.add(3);
al.add(4);
System.out.println("Contents of al: " + al);
Integer ia[] = new Integer[al.size()];
ia = al.toArray(ia);
int sum = 0;
for(int i : ia) sum += i;
System.out.println("Sum is: " + sum);
}
}
```

LinkedList

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces.
It provides a linked-list data structure. **LinkedList** is a generic class that has this declaration:

```
class LinkedList<E>
```

Here, **E** specifies the type of objects that the list will hold. **LinkedList** has the two constructors
LinkedList( )
LinkedList(Collection<? extends E> *c*)
The first constructor builds an empty linked list.
The second constructor builds a linked list that is initialized with the elements of the collection *c*.
Example code:

```
import java.util.*;
class LinkedListDemo {
```


Source diginotes.in

```
public static void main(String args[]) {
LinkedList<String> ll = new LinkedList<String>();
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "+ ll);
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "+ ll);
String val = ll.get(2);
ll.set(2, val + " Changed");
System.out.println("ll after change: " + ll);
}
}
```

HashSet

HashSet extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage.

Java HashSet class is used to create a collection that uses a hash table for storage.

It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.

HashSet is a generic class that has this declaration:

class HashSet<E>

Here, **E** specifies the type of objects that the set will hold.

Constructor

HashSet( )
HashSet(Collection<? extends E> *c*)
HashSet(int *capacity*)
HashSet(int *capacity*, float *fillRatio*)

Example:

```
import java.util.*;
class HashSetDemo {
public static void main(String args[]) {
HashSet<String> hs = new HashSet<String>();
```


Source diginotes.in

```
hs.add("B");
hs.add("A");
hs.add("D");
hs.add("E");
hs.add("C");
hs.add("F");
System.out.println(hs);
}
}
```

output

[D, A, F, C, B, E]

LinkedHashSet

LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

- Contains unique elements only like HashSet.
- Provides all optional set operations, and permits null elements.
- Maintains insertion order.

The **LinkedHashSet** class extends **HashSet** and adds no members of its own.
It is a generic class that has this declaration:

class LinkedHashSet<E>

Here, **E** specifies the type of objects that the set will hold.

LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted.
This allows insertion-order iteration over the set.
That is, when cycling through a **LinkedHashSet** using an iterator, the elements will be returned in the order in which they were inserted.
This is also the order in which they are contained in the string returned by **toString( )** when called on a **LinkedHashSet** object.
To see the effect of **LinkedHashSet**, try substituting **LinkedHashSet** for **HashSet** in the preceding program. The output will be

[B, A, D, E, C, F]

which is the order in which the elements were inserted.

TreeSet

TreeSet extends **AbstractSet** and implements the **NavigableSet** interface.
It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.
TreeSet is a generic class that has this declaration:

class TreeSet<E>

Here, **E** specifies the type of objects that the set will hold.

Source diginotes.in

TreeSet has the following constructors:

TreeSet( )
TreeSet(Collection<? extends E> *c*)
TreeSet(Comparator<? super E> *comp*)
TreeSet(SortedSet<E> *ss*)

Example

```
import java.util.*;
class TreeSetDemo {
public static void main(String args[]) {
TreeSet<String> ts = new TreeSet<String>();
ts.add("C");
ts.add("A");
ts.add("B");
ts.add("E");
ts.add("F");
ts.add("D");
System.out.println(ts);
}
}
```

The output from this program is shown here:

[A, B, C, D, E, F]

PriorityQueue

PriorityQueue extends **AbstractQueue** and implements the **Queue** interface.
It creates a queue that is prioritized based on the queue's comparator.
PriorityQueue is a generic class that has this declaration:
class PriorityQueue<E>
Here, **E** specifies the type of objects stored in the queue.
PriorityQueues are dynamic, growing as necessary.
PriorityQueue defines the six constructors shown here:
PriorityQueue( )
PriorityQueue(int *capacity*)
PriorityQueue(int *capacity*, Comparator<? super E> *comp*)
PriorityQueue(Collection<? extends E> *c*)
PriorityQueue(PriorityQueue<? extends E> *c*)
PriorityQueue(SortedSet<? extends E> *c*)

ArrayDeque

Java SE 6 added the **ArrayDeque** class, which extends **AbstractCollection** and implements the **Deque** interface.
It adds no methods of its own.
ArrayDeque creates a dynamic array and has no capacity restrictions.
ArrayDeque is a generic class that has this declaration:
class ArrayDeque<E>
Here, **E** specifies the type of objects stored in the collection.
ArrayDeque defines the following constructors:
ArrayDeque( )
ArrayDeque(int *size*)
ArrayDeque(Collection<? extends E> *c*)

Source diginotes.in

Example:
```
import java.util.*;
class ArrayDequeDemo {
public static void main(String args[]) {
ArrayDeque<String> adq = new ArrayDeque<String>();
adq.push("A");
adq.push("B");
adq.push("D");
adq.push("E");
adq.push("F");
System.out.print("Popping the stack: ");
while(adq.peek() != null)
System.out.print(adq.pop() + " ");
System.out.println();
}
}
```
The output is shown here:

Popping the stack: F E D B A

Accessing a collection Via an Iterator:

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an **iterator( )** method that returns an iterator to the start of the collection.

By using this iterator object, you can access each element in the collection. Element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.
2. Set up a loop that makes a call to **hasNext( )**. Have the loop iterate as long as **hasNext( )** returns **true**.
3. Within the loop, obtain each element by calling **next( )**.

4. For collections that implement **List**, you can also obtain an iterator by calling **listIterator( )**.
5.As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element.
6. Otherwise, **ListIterator** is used just like **Iterator**.

```
import java.util.*;
class IteratorDemo {
public static void main(String args[]) {

ArrayList<String> al = new ArrayList<String>();
al.add("C");
al.add("A");
al.add("E");
```


Source diginotes.in

```
al.add("B");
al.add("D");
al.add("F");

System.out.print("Original contents of al: ");

Iterator<String> itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();

ListIterator<String> litr = al.listIterator();
while(litr.hasNext()) {
String element = litr.next();
litr.set(element + "+");
}
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
String element = litr.previous();
System.out.print(element + " ");
}
System.out.println();
}
}
```

Output:
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+

For Each loop for iterating through collection:

```
import java.util.*;
class ForEachDemo {
public static void main(String args[]) {
ArrayList<Integer> vals = new ArrayList<Integer>();
vals.add(1);
vals.add(2);
vals.add(3);
vals.add(4);
vals.add(5);
System.out.print("Original contents of vals: ");
for(int v : vals)
```


Source diginotes.in

```
System.out.print(v + " ");
System.out.println();
int sum = 0;
for(int v : vals)
sum += v;
System.out.println("Sum of values: " + sum);
}
}
```

Output:

Original contents of vals: 1 2 3 4 5
Sum of values: 15

Storing User Defined Classes in Collections:

collections are not limited to the storage of built-in objects.

The power of collections is that they can store any type of object, including objects of classes that you create.
User defined objects stored in **LinkedList** to store mailing addresses:

```
import java.util.*;
class Address {
private String name;
private String street;
private String city;
private String state;
private String code;
Address(String n, String s, String c,
String st, String cd) {
name = n;
street = s;
city = c;
state = st;
code = cd;
}
public String toString() {
return name + "\n" + street + "\n" +
city + " " + state + " " + code;
}}
class MailList {
public static void main(String args[]) {
LinkedList<Address> ml = new LinkedList<Address>();
ml.add(new Address("J.W. West", "11 Oak Ave",
"Urbana", "IL", "61801"));
ml.add(new Address("Ralph Baker", "1142 Maple Lane",
"Mahomet", "IL", "61853"));
ml.add(new Address("Tom Carlton", "867 Elm St",
"Champaign", "IL", "61820"));
for(Address element : ml)
System.out.println(element + "\n");
```


Source diginotes.in

```
System.out.println();
}
}
```

The output from the program is shown here:

J.W. West
11 Oak Ave
Urbana IL 61801
Ralph Baker
1142 Maple Lane
Mahomet IL 61853
Tom Carlton
867 Elm St
Champaign IL 61820

Random Access Interface:

RandomAccess interface contains no members.

However, by implementing this interface, a collection signals that it supports efficient random access to its elements.

By checking for the **RandomAccess** interface, client code can determine at run time whether a collection is suitable for certain types of random access operations—especially as they apply to large collections.

RandomAccess is implemented by **ArrayList** and by the legacy **Vector** class, among others.

Working With Maps:

A *map* is an object that stores associations between keys and values, or *key/value pairs.*

Keys and values are objects. Keys must be unique, but the values may be duplicated.

Some maps can accept a **null** key and **null** values, others cannot.

There is one key point about maps that is important to mention at the outset: they don't implement the **Iterable** interface. This means that you *cannot* cycle through a map using a for-each style **for** loop. Furthermore, you can't obtain an iterator to a map.

However, as you will soon see, you can obtain a collection-view of a map, which does allow the use of either the **for** loop or an iterator.

The Map Interfaces

Because the map interfaces define the character and nature of maps, this discussion of maps begins with them.

The following interfaces support maps:
The Map Interface
The **Map** interface maps unique keys to values. A *key* is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using its key.

Map is generic:
interface Map<K, V>

Source diginotes.in

Here, **K** specifies the type of keys, and **V** specifies the type of values.
The methods declared by **Map.**

Several methods
throw a **ClassCastException** when an object is incompatible with the elements in a map.

A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the map.

An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map.

An **IllegalArgumentException** is thrown if an invalid argument is used.

Maps revolve around two basic operations: **get( )** and **put( )**. To put a value into a map, use **put( )**, specifying the key and the value.

To obtain a value, call **get( )**, passing the key as an argument. The value is returned.
maps are not, themselves, collections because they do not implement the **Collection** interface. However, you can obtain a collection-view of a map. To do this, you can use the **entrySet( )** method. It returns a **Set** that contains the elements in the map.

To obtain a collection-view of the keys, use **keySet( )**.

To get a collection-view of the values, use **values( )**.

Collection-views are the means by which maps are integrated into the larger Collections Framework.

SortedMap
The **SortedMap** interface extends **Map**. It ensures that the entries are maintained in ascending
order based on the keys.

SortedMap is generic and is declared as shown here:

interface SortedMap<K, V>

K specifies the type of keys,

V specifies the type of values.

Several methods throw a **NoSuchElementException** when no items are in the invoking map.
A **ClassCastException** is thrown when an object is incompatible with the elements in a map.
A **NullPointerException** is thrown if an attempt is made to use a **null** object when **null** is not allowed in the map.

An **IllegalArgumentException** is thrown if an invalid argument is used.
Sorted maps allow very efficient manipulations of *submaps*

To obtain a submap, use **headMap( )**, **tailMap( )**, or **subMap( )**.

Source diginotes.in

To get the first key in the set, call **firstKey( )**.

To get the last key, use **lastKey( )**.

NavigableMap Interface

The **NavigableMap** interface was added by Java SE 6.

It extends **SortedMap** and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys. **NavigableMap** is a generic interface that has this declaration:
interface NavigableMap<K,V>

Here, **K** specifies the type of the keys, and **V** specifies the type of the values associated with the keys.

Several methods throw a **ClassCastException** when an object is incompatible with the keys in the map.

A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** keys are not allowed in the set.

An **IllegalArgumentException** is thrown if an invalid argument is used.
equal to start.

Map.Entry Interface

The **Map.Entry** interface enables you to work with a map entry. Recall that the **entrySet( )** method declared by the **Map** interface returns a **Set** containing the map entries.

Each of these set elements is a **Map.Entry** object. **Map.Entry** is generic and is declared like this:
interface Map.Entry<K, V> Here, **K** specifies the type of keys, and **V** specifies the type of values.
the methods declared by **Map.Entry**.

Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

HashMap:

The **HashMap** class extends **AbstractMap** and implements the **Map** interface. It uses a hash table to store the map.

This allows the execution time of **get( )** and **put( )** to remain constant
even for large sets. **HashMap** is a generic class that has this declaration:

class HashMap<K, V>

Source diginotes.in

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following constructors are defined:
HashMap( )
HashMap(Map<? extends K, ? extends V> *m*)
HashMap(int *capacity*)
HashMap(int *capacity*, float *fillRatio*)
The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m.* The third form initializes the capacity of the hash map to *capacity.* The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments.

The meaning of capacity and fill ratio is the same as for **HashSet**, described earlier. The default capacity is 16.

The default fill ratio is 0.75.
HashMap implements **Map** and extends **AbstractMap**. It does not add any methods of its own.

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
HashMap<String, Double> hm = new HashMap<String, Double>();
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
Set<Map.Entry<String, Double>> set = hm.entrySet();
for(Map.Entry<String, Double> me : set) {
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());
}
System.out.println();
double balance = hm.get("John Doe");
hm.put("John Doe", balance + 1000);
System.out.println("John Doe's new balance: " +
hm.get("John Doe"));
}
}
```

Output from this program is shown here (the precise order may vary):

```
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0
John Doe’s new balance: 4434.34
```


Source diginotes.in

The program begins by creating a hash map and then adds the mapping of names to balances. Next, the contents of the map are displayed by using a set-view, obtained by calling **entrySet( )**. The keys and values are displayed by calling the **getKey( )** and **getValue( )** methods
that are defined by **Map.Entry**. Pay close attention to how the deposit is made into John Doe's
account. The **put( )** method automatically replaces any preexisting value that is associated with the specified key with the new value. Thus, after John Doe's account is updated, the hash map will still contain just one "John Doe" account.

TreeMap

The **TreeMap** class extends **AbstractMap** and implements the **NavigableMap** interface.
It creates maps stored in a tree structure.

A **TreeMap** provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.

TreeMap is a generic class that has this declaration:

```
class TreeMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following **TreeMap** constructors are defined:

```
TreeMap( )
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)
```

The first form constructs an empty tree map that will be sorted by using the natural order of its keys. The second form constructs an empty tree-based map that will be sorted by using the **Comparator** *comp.* (Comparators are discussed later in this chapter.) The third form initializes
a tree map with the entries from *m,* which will be sorted by using the natural order of the keys. The fourth form initializes a tree map with the entries from *sm,* which will be sorted in the same order as *sm.*

TreeMap has no methods beyond those specified by the **NavigableMap** interface and the **AbstractMap** class.

The following program reworks the preceding example so that it uses **TreeMap**:

```
import java.util.*;
class TreeMapDemo {
public static void main(String args[]) {
// Create a tree map.
TreeMap<String, Double> tm = new TreeMap<String, Double>();
// Put elements to the map.
tm.put("John Doe", new Double(3434.34));
tm.put("Tom Smith", new Double(123.22));
tm.put("Jane Baker", new Double(1378.00));
tm.put("Tod Hall", new Double(99.22));
tm.put("Ralph Smith", new Double(-19.08));
```


Source diginotes.in

```
// Get a set of the entries.
Set<Map.Entry<String, Double>> set = tm.entrySet();
// Display the elements.
for(Map.Entry<String, Double> me : set) {
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());
}
System.out.println();
double balance = tm.get("John Doe");
tm.put("John Doe", balance + 1000);
System.out.println("John Doe's new balance: " +
tm.get("John Doe"));
}
}
```

The following is the output from this program:

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22
John Doe’s current balance: 4434.34
```

TreeMap sorts the keys.

However, in this case, they are sorted by first name
instead of last name.

You can alter this behavior by specifying a comparator when the map
is created, as described shortly.

LinkedHashMap

LinkedHashMap extends **HashMap**.

It maintains a linked list of the entries in the map, in the
1order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating through a collection-view of a **LinkedHashMap**, the elements will be returned in the order in which they were inserted.

LinkedHashMap that returns its elements in the order in which they were last accessed.

LinkedHashMap is a generic class that has this declaration:

class LinkedHashMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

LinkedHashMap defines the following constructors:

LinkedHashMap( )
LinkedHashMap(Map<? extends K, ? extends V> *m*)
LinkedHashMap(int *capacity*)

Source diginotes.in

LinkedHashMap(int *capacity*, float *fillRatio*)
LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*)
The first form constructs a default **LinkedHashMap**.

The second form initializes the **LinkedHashMap** with the elements from *m.* The third form initializes the capacity. The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for **HashMap**. The default capactiy is 16. The default ratio is 0.75. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access.

IdentityHashMap

IdentityHashMap extends **AbstractMap** and implements the **Map** interface.

It is similar to **HashMap** except that it uses reference equality when comparing elements. **IdentityHashMap** is a generic class that has this declaration:
class IdentityHashMap<K, V>
Here, **K** specifies the type of key, and **V** specifies the type of value. The API documentation explicitly states that **IdentityHashMap** is not for general use.

The EnumMap Class

EnumMap extends **AbstractMap** and implements **Map**. It is specifically for use with keys of an **enum** type. It is a generic class that has this declaration:
class EnumMap<K extends Enum<K>, V>
Here, **K** specifies the type of key, and **V** specifies the type of value. Notice that **K** must extend **Enum<K>**, which enforces the requirement that the keys must be of an **enum** type. **EnumMap** defines the following constructors:

EnumMap(Class<K> *kType*)
EnumMap(Map<K, ? extends V> *m*)
EnumMap(EnumMap<K, ? extends V> *em*)

The first constructor creates an empty **EnumMap** of type *kType.* The second creates an **EnumMap** map that contains the same entries as *m.* The third creates an **EnumMap** initialized with the values in *em.*

Comparator interface

Comparator is a generic interface that has this declaration:
interface Comparator<T>
Here, **T** specifies the type of objects being compared.
The **Comparator** interface defines two methods: **compare( )** and **equals( )**. The **compare( )** method, shown here, compares two elements for order:
int compare(T *obj1*, T *obj2*)
obj1 and *obj2* are the objects to be compared.

This method returns zero if the objects are equal.
It returns a positive value if *obj1* is greater than *obj2.* Otherwise, a negative value is returned. **ClassCastException** if the types of the objects are not compatible for comparison.

By overriding **compare( )**, you can alter the way that objects are ordered.

Source diginotes.in

For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison. The **equals( )** method, shown here, tests whether an object equals the invoking comparator:

boolean equals(Object *obj*)

Here, *obj* is the object to be tested for equality. The method returns **true** if *obj* and the invoking object are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**.

```
import java.util.*;
class MyComp implements Comparator<String> {
public int compare(String a, String b) {
String aStr, bStr;
aStr = a;
bStr = b;
return bStr.compareTo(aStr);
}
}
class CompDemo {
public static void main(String args[]) {
TreeSet<String> ts = new TreeSet<String>(new MyComp());
ts.add("C");
ts.add("A");
ts.add("B");
ts.add("E");
ts.add("F");
ts.add("D");
for(String element : ts)
System.out.print(element + " ");
System.out.println();
}}
```

Output:

F E D C B A

The Collection Algorithms:

Collections Framework defines several algorithms that can be applied to collections and maps.

algorithms are defined as static methods within the **Collections** class.

static <T> Boolean addAll(Collection <? super T> c, T ... elements)

Inserts the elements specified by elements into the collection specified by c. Returns true if the elements were added and false otherwise.

static <T> Queue<T> asLifoQueue(Deque<T> c)

Returns a last-in, first-out view of c.

static <T>int binarySearch(List<? extends T> list, T value, Comparator<? super T> c)

Searches for value in list ordered according to c. Returns the position of value in list, or a negative value if value is not found.

static <T> int binarySearch(List<? extends Comparable<? super T>> list,T value)

Source diginotes.in

Searches for value in list. The list must be sorted. Returns the position of value in list, or a negative value if value is not found.

static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> t)
Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a ClassCastException.

static <E> List<E> checkedList(List<E> c, Class<E> t)
Returns a run-time type-safe view of a List. An attempt to insert an incompatible element will cause a ClassCastException.

static <K, V> Map<K, V> checkedMap(Map<K, V> c, Class<K> keyT, Class<V> valueT)
Returns a run-time type-safe view of a Map.
An attempt to insert an incompatible element will cause a ClassCastException.

static <E> List<E> checkedSet(Set<E> c, Class<E> t)
Returns a run-time type-safe view of a Set. An
attempt to insert an incompatible element will cause a ClassCastException.

static int frequency(Collection<?> c, Object obj)
Counts the number of occurrences of obj in c and returns the result.
static int indexOfSubList(List<?> list, List<?> subList)
Searches list for the first occurrence of subList.
Returns the index of the first match, or –1 if no match is found.
static int lastIndexOfSubList(List<?> list, List<?> subList)
Searches list for the last occurrence of subList.
Returns the index of the last match, or –1 if no
match is found.

```
import java.util.*;
class AlgorithmsDemo {
public static void main(String args[]) {
LinkedList<Integer> ll = new LinkedList<Integer>();
ll.add(-8);
ll.add(20);
ll.add(-20);
ll.add(8);
Comparator<Integer> r = Collections.reverseOrder();
Collections.sort(ll, r);
System.out.print("List sorted in reverse: ");
for(int i : ll)
System.out.print(i+ " ");
System.out.println();
Collections.shuffle(ll);
System.out.print("List shuffled: ");
for(int i : ll)
System.out.print(i + " ");
System.out.println();
System.out.println("Minimum: " + Collections.min(ll));
```


Source diginotes.in

```
System.out.println("Maximum: " + Collections.max(ll));
}
}
```

Output:
List sorted in reverse: 20 8 -8 -20
List shuffled: 20 -20 8 -8
Minimum: -20
Maximum: 20
Notice that **min( )** and **max( )** operate on the list after it has been shuffled. Neither requires a sorted list for its operation.

Why Generic Collections?
As mentioned at the start of this chapter, the entire Collections Framework was refitted for generics when JDK 5 was released.

Furthermore, the Collections Framework is arguably the single most important use of generics in the Java API.

The reason for this is that generics add type safety to the Collections Framework. Before moving on, it is worth taking some time to examine in detail the significance of this improvement.

```
import java.util.*;
class OldStyle {
public static void main(String args[]) {
ArrayList list = new ArrayList();
list.add("one");
list.add("two");
list.add("three");
list.add("four");
Iterator itr = list.iterator();
while(itr.hasNext()) {
String str = (String) itr.next(); // explicit cast needed here.
System.out.println(str + " is " + str.length() + " chars long.");
}
}
}
```

Prior to generics, all collections stored references of type **Object**.

This allowed any type of reference to be stored in the collection.

The preceding program uses this feature to store references to objects of type **String** in **list**, but any type of reference could have been stored. Unfortunately, the fact that a pre-generics collection stored **Object** references could easily lead to errors.

First, it required that you, rather than the compiler, ensure that only objects of
the proper type be stored in a specific collection. For example, in the preceding example, **list** is clearly intended to store **String**s, but there is nothing that actually prevents another type of reference from being added to the collection.

Source diginotes.in

For example, the compiler will find nothing wrong with this line of code:
list.add(new Integer(100));

Because **list** stores **Object** references, it can store a reference to **Integer** as well as it can store a reference to **String**.
However, if you intended **list** to hold only strings, then the preceding statement would corrupt the collection. Again, the compiler had no way to know that the preceding statement is invalid.

The second problem with pre-generics collections is that when you retrieve a reference from the collection, you must manually cast that reference into the proper type.
This is why the preceding program casts the reference returned by **next( )** into **String**. Prior to generics, collections simply stored **Object** references. Thus, the cast was necessary when retrieving
objects from a collection.

Aside from the inconvenience of always having to cast a retrieved reference into its proper type, this lack of type safety often led to a rather serious, but surprisingly easy-to-create, error. Because **Object** can be cast into any type of object, it was possible to cast a reference obtained from a collection into the *wrong type*. For example, if the following statement were added to the preceding example, it would still compile without error, but generate a run-time exception when executed:

Integer i = (Integer) itr.next();

• Ensures that only references to objects of the proper type can actually be stored in a collection. Thus, a collection will always contain references of a known type.
• Eliminates the need to cast a reference retrieved from a collection. Instead, a reference retrieved from a collection is automatically cast into the proper type. This prevents run-time errors due to invalid casts and avoids an entire category of errors.

The Legacy Classes and Interfaces

As explained at the start of this chapter, early versions of **java.util** did not include the Collections Framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing objects.

When collections were added (by J2SE 1.2), several of the original classes were reengineered to support the collection interfaces.

Thus, they are fully compatible with the framework. While no classes have actually been deprecated, one has been rendered obsolete.

Of course, where a collection duplicates the functionality of a legacy class,you will usually want to use the collection for new code. In general, the legacy classes are supported because there is still code that uses them.

One other point: none of the collection classes are synchronized, but all the legacy classes are synchronized.

Source diginotes.in

This distinction may be important in some situations. Of course, you can
easily synchronize collections, too, by using one of the algorithms provided by **Collections**.

The legacy classes defined by **java.util** are shown here:

Dictionary
Hashtable
Properties
Stack
Vector

There is one legacy interface called **Enumeration**.

The following sections examine **Enumeration** and each of the legacy classes, in turn.
The Enumeration Interface
The **Enumeration** interface defines the methods by which you can *enumerate* (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by **Iterator**.
interface Enumeration<E>
where **E** specifies the type of element being enumerated.

Vector

Vector implements a dynamic array. It is similar to **ArrayList**, but with two differences: **Vector** is synchronized, and it contains many legacy methods that are not part of the Collections.
Vector is declared like this:
class Vector<E>
Here, **E** specifies the type of element that will be stored.

Here are the **Vector** constructors:
Vector( )
Vector(int *size*)
Vector(int *size*, int *incr*)
Vector(Collection<? extends E> *c*)

- The first form creates a default vector, which has an initial size of 10.
- The second form creates a vector whose initial capacity is specified by *size.*
- The third form creates a vector whose initial capacity is specified by *size* and whose increment is specified by *incr.*

The increment specifies the number of elements to allocate each time that a vector is resized upward.
The fourth form creates a vector that contains the elements of collection *c.*

Stack

Stack is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** only defines the default constructor, which creates an empty stack. With the release of JDK 5, **Stack** was retrofitted for generics and is declared as shown here:

class Stack<E>

Source diginotes.in

Here, **E** specifies the type of element stored in the stack.
Stack includes all the methods defined by **Vector.**

Dictionary

Dictionary is an abstract class that represents a key/value storage repository and operates much like **Map**.
Given a key and value, you can store the value in a **Dictionary** object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.
Although not currently deprecated, **Dictionary** is classified as obsolete, because it is fully superseded by **Map**. However, **Dictionary** is still in use and thus is fully discussed here.

class Dictionary<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values. The abstract methods defined by **Dictionary** are listed in Table 17-17.

Hashtable

Hashtable was part of the original **java.util** and is a concrete implementation of a Dictionary.
HashMap, **Hashtable** stores key/value pairs in a hash table. However, neither keys nor values can be **null**. When using a **Hashtable**, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
Hashtable was made generic by JDK 5.
It is declared like this: class Hashtable<K, V>

Hashtable( )
Hashtable(int *size*)
Hashtable(int *size*, float *fillRatio*)
Hashtable(Map<? extends K, ? extends V> *m*)

The first version is the default constructor.

The second version creates a hash table that has an initial size specified by *size*.

The third version creates a hash table that has an initial size specified by *size* and a fill ratio specified by *fillRatio*. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hashtable multiplied by its fill ratio, the hash table is expanded. If you do not specify a fill ratio, then 0.75 is used.

Finally, the fourth version creates a hash table that is initialized with the elements in *m*. The capacity of the hash table is set to twice the number of elements in *m*. The default load factor of 0.75 is used.

Source diginotes.in

Properties

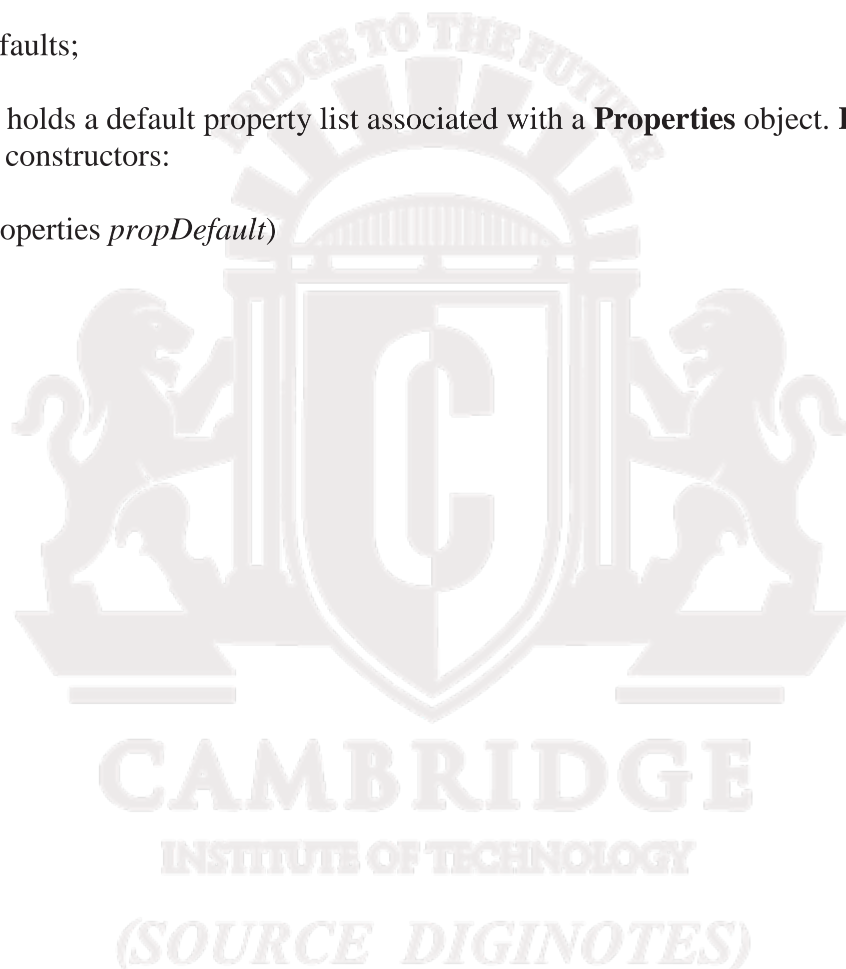
Properties is a subclass of **Hashtable**. It is used to maintain lists of values in which the key is a **String** and the value is also a **String**.

The **Properties** class is used by many other Java classes. For example, it is the type of object returned by **System.getProperties( )** when obtaining environmental values.

Although the **Properties** class, itself, is not generic, several of its methods are.
Properties defines the following instance variable:

Properties defaults;

This variable holds a default property list associated with a **Properties** object. **Properties** defines these constructors:
Properties( )
Properties(Properties *propDefault*)


CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

Source diginotes.in

Module 3

Syllabus -String Handling :

The String Constructors, String Length, Special String Operations, String Literals, String Concatenation, String Concatenation with Other Data Types, String Conversion and toString( ) Character Extraction, charAt( ), getChars( ), getBytes( ) toCharArray(), String Comparison, equals( ) and equalsIgnoreCase( ), regionMatches( ) startsWith( ) and endsWith( ), equals( ) Versus == , compareTo( ) Searching Strings, Modifying a String, substring( ), concat( ), replace( ), trim( ), Data Conversion Using valueOf( ), Changing the Case of Characters Within a String, Additional String Methods, StringBuffer , StringBuffer Constructors, length( ) and capacity( ), ensureCapacity( ), setLength( ), charAt( ) and setCharAt( ), getChars( ),append( ), insert( ), reverse( ), delete( ) and deleteCharAt( ), replace( ), substring( ), Additional StringBuffer Methods, StringBuilder.

1. **What are the different types of String Constructors available in Java?**

   The String class supports several constructors.

   a. To create an empty String
      the default constructor is used.

   Ex: String s = new String();
   will create an instance of String with no characters in it.

   b. To create a String initialized by an array of characters, use the constructor shown here:

      ```
                String(char chars[ ])
       ex:      char chars[] = { 'a', 'b', 'c' };
                String s = new String(chars);
      ```

      This constructor initializes s with the string “abc”.

   c. To specify a subrange of a character array as an initializer using the following constructor:
      String(char chars[ ], int startIndex, int numChars)

      Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use. Here is an example:

      ```
      char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
      String s = new String(chars, 2, 3);
      ```

      This initializes s with the characters cde.

   d. To construct a String object that contains the same character sequence as another String object using this constructor:

      String(String strObj)
      Here, strObj is a String object.

      ```
      class MakeString
      { public static void main(String args[])
      { char c[] = {'J', 'a', 'v', 'a'};
      String s1 = new String(c);
      ```


source diginotes.in

```
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
}
}
```

The output from this program is as follows:

Java
Java

As you can see, s1 and s2 contain the same string.

e. To Construct string using byte array:

Even though Java's char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set.

Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array.

Ex: String(byte asciiChars[ ])

String(byte asciiChars[ ], int startIndex, int numChars)

The following program illustrates these constructors:

```
class SubStringCons
{ public static void main(String args[])
{
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);
System.out.println(s1);
String s2 = new String(ascii, 2, 3);
System.out.println(s2);
}
}
```

This program generates the following output:

ABCDEF
CDE

f. To construct a String from a StringBuffer by using the constructor shown here:

Ex: String(StringBuffer strBufObj)

g. Constructing string using Unicode character set and is shown here:

String(int codePoints[ ], int startIndex, int numChars)

codePoints is an array that contains Unicode code points.

h. Constructing string that supports the new StringBuilder class.

Ex : String(StringBuilder strBuildObj)

source diginotes.in

Note:
String can be constructed by using string literals.
String s1="Hello World"

String concatenation can be done using + operator. With other data type also.

String Length

1. The length of a string is the number of characters that it contains. To obtain this value, call the length( ) method,
2. Syntax:
   int length( )
3. Example
   char chars[] = { 'a', 'b', 'c' }; String s = new String(chars);
   System.out.println(s.length());// 3

toString()

1. Every class implements toString( ) because it is defined by Object. However, the default Implementation of toString() is sufficient.
2. For most important classes that you create, will want to override toString() and provide your own string representations.
   String toString( )
3. To implement toString( ), simply return a String object that contains the human-readable string that appropriately describes an object of your class.
4. By overriding toString() for classes that you create, you allow them to be fully integrated into Java's programming environment. For example, they can be used in print() and println() statements and in concatenation expressions.

5. The following program demonstrates this by overriding toString( ) for the Box class:

```
class Box
{
double width; double height; double depth;
 Box(double w, double h, double d)
{ width = w; height = h; depth = d; }

public String toString()
{ return "Dimensions are " + width + " by " + depth + " by " + height + "."; }
}

class toStringDemo {
public static void main(String args[])
{
 Box b = new Box(10, 12, 14);
String s = "Box b: " + b;
```


source diginotes.in

```
System.out.println(b);
System.out.println(s);
}
}
```

The output of this program is shown here:
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0

Character Extraction

The String class provides a number of ways in which characters can be extracted from a String object. String object can not be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

A. charAt( )

1. description:

To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method.

2. Syntax

char charAt(int where)

Here, where is the index of the character that you want to obtain.
charAt( ) returns the character at the specified location.

3. example,

```
char ch;
 ch = "abc".charAt(1);
```

assigns the value “b” to ch.

B. getChars( )

1. to extract more than one character at a time, you can use the getChars() method.
2. Syntax

   void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

Here, sourceStart specifies the index of the beginning of the substring, sourceEnd specifies an index that is one past the end of the desired The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart.

3. class getCharsDemo {

```
public static void main(String args[])
{ String s = "This is a demo of the getChars method.";
int start = 10;
 int end = 14;
 char buf[] = new char[end - start];
  s.getChars(start, end, buf, 0);
```


source diginotes.in

```
    System.out.println(buf);
   }
}
```

Here is the output of this program:
demo

C. getBytes( )

1. This method is called getBytes( ), and it uses the default character-to-byte conversions provided by the platform.
   Syntax:
   byte[ ] getBytes( )
   Other forms of getBytes( ) are also available.
2. getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

D. toCharArray( )

If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ).

It returns an array of characters for the entire string.

It has this general form:
char[ ] toCharArray( )

2. String Comparison:

The String class includes several methods that compare strings or substrings within strings.

equals( )

To compare two strings for equality, use equals( ).
It has this general form:
boolean equals(Object str)

Here, str is the String object being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive.

A. equalsIgnoreCase( )

To perform a comparison that ignores case differences, call equalsIgnoreCase( ). When it compares two strings, it considers A-Z to be the same as a-z.
It has this general form:
boolean equalsIgnoreCase(String str)
Here, str is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and false otherwise.

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
 public static void main(String args[]) {
```


source diginotes.in

```
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " equals " + s3 + " -> " + s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " -> " + s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
s1.equalsIgnoreCase(s4)); } }
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

B. regionMatches( )

1. The regionMatches() method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons.
2. Syntax:

boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)

boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)

3. For both versions, startIndex specifies the index at which the region begins within the invoking String object.

The String being compared is specified by str2. The index at which the comparison will start within str2 is specified by str2 StartIndex. The length of the substring being compared is passed in numChars.

4. In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

C. startsWith( ) and endsWith( )

1. The startsWith( ) method determines whether a given String begins with a specified string.
2. endsWith( ) determines whether the String in question ends with a specified string.
3. Syntax

```
boolean startsWith(String str)
boolean endsWith(String str)
```


source diginotes.in

Here, str is the String being tested.
If the string matches, true is returned.
Otherwise, false is returned.
For example,

```
"Foobar".endsWith("bar")

"Foobar".startsWith("Foo")
```

are both true.

4. A second form of startsWith( ), shown here, lets you specify a starting point:

```
boolean startsWith(String str, int startIndex)
```

Here, startIndex specifies the index into the invoking string at which point the search will begin. For example,

```
"Foobar".startsWith("bar", 3)
```

returns true.

D. **equals( ) Versus ==**

It is important to understand that the equals( ) method and the == operator perform two different operations.
the equals( ) method compares the characters inside a String object.
The == operator compares two object references to see whether they refer to the same instance.

```
class EqualsNotEqualTo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
}
}
```

E. **compareTo( )**

1. Sorting applications, you need to know which is less than, equal to, or greater than the next.
2. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The String method compareTo() serves this purpose.
3. It has this general form:

```
int compareTo(String str)
```

Here, str is the String being compared with the invoking String. The result of the comparison is returned and is interpreted,

4. Less than zero when invoking string is less than str.
5. Greater than zero when invoking string is greater than str.
6. Zero The two strings are equal.

source diginotes.in

```
// A bubble sort for Strings.
class SortString
{ static String arr[] = { "Now", "is", "the", "time", "for", "all", "good", "men",
"to", "come", "to", "the", "aid", "of", "their", "country" };
 public static void main(String args[])
 { for(int j = 0; j < arr.length; j++)
{ for(int i = j + 1; i < arr.length; i++)
 { if(arr[i].compareTo(arr[j]) < 0)
 { String t = arr[j];
arr[j] = arr[i];
 arr[i] = t;
}
} System.out.println(arr[j]);
}
}
}
```

The output of this program is the list of words:

Now aid all come country for good is men of the the their time to to
As you can see

7. Ignore case differences when comparing two strings, use compareToIgnoreCase( ), This method returns the same results as compareTo(), except that case differences are ignored.

5. Searching String

A. indexOf( ) and lastIndexOf( )

1. indexOf( ) Searches for the first occurrence of a character or substring.

2. lastIndexOf( ) Searches for the last occurrence of a character or substring.

3. These two methods are overloaded in several different ways

4. return the index at which the character or substring was found, or –1 on failure.

5. To search for the first occurrence of a character, int indexOf(int ch)
6. To search for the last occurrence of a character,
   int lastIndexOf(int ch) Here, ch is the character being sought

7. To search for the first or last occurrence of a substring, use int indexOf(String str) int lastIndexOf(String str) Here, str specifies the substring.

source diginotes.in

8. You can specify a starting point for the search using these forms:
   int indexOf(int ch, int startIndex)
   int lastIndexOf(int ch, int startIndex)

9. int indexOf(String str, int startIndex) int lastIndexOf(String str, int startIndex) Here, startIndex specifies the index at which point the search begins.
10. For indexOf( ), the search runs from startIndex to the end of the string. For lastIndexOf( ), the search runs from startIndex to zero. The following example shows how to use the various index methods to search inside of Strings:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
 public static void main(String args[])
{ String s = "Now is the time for all good men " + "to come to the aid
of their country.";
System.out.println(s);
System.out.println("indexOf(t) = " + s.indexOf('t'));
System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t'));
System.out.println("indexOf(the) = " + s.indexOf("the"));
System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " + s.indexOf('t', 10));
System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " + s.lastIndexOf("the",
60));
}
}
```

Output

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
 indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
 lastIndexOf(the, 60) = 55
```

6. Modifying a String

String objects are immutable, whenever you want to modify a String, you must either copy it into a StringBuffer or StringBuilder, or use one of the following String methods, which will construct a new copy of the string with your modifications complete.

source diginotes.in

A. Substring( )

1. You can extract a substring using substring( ). It has two forms. The first is String substring(int startIndex)
2. Here, startIndex specifies the index at which the substring will begin. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.
3. The second form of substring( ) allows you to specify both the beginning and ending index of the substring:
   String substring(int startIndex, int endIndex)
   Here, startIndex specifies the beginning index, and endIndex specifies the stopping point.
4. The string returned contains all the characters from the beginning index, up to, but not including, the ending index. The following program uses substring( ) to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace {
public static void main(String args[]) {
String org = "This is a test. This is, too.";
String search = "is";
String sub = "was";
String result = "";
int i;
do  {
System.out.println(org);
i = org.indexOf(search);
 if(i != -1) { result = org.substring(0, i);
result = result + sub;
result = result + org.substring(i + search.length());
 org = result;
} } while(i != -1);
}
}
```

The output from this program is shown here:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
 Thwas was a test. Thwas is, too.
 Thwas was a test. Thwas was, too.
```

B. concat( )

source diginotes.in

1. concatenate two strings using concat( )

String concat(String str)

2. This method creates a new object that contains the invoking string with the contents of str appended to the end.
3. concat( ) performs the same function as +.
4. String s1 = "one";
   String s2 = s1.concat("two");

C. replace( )

1. The replace( ) method has two forms.
2. The first replaces all occurrences of one character in the invoking string with another character.

Syntax:

String replace(char original, char replacement)

Here, original specifies the character to be replaced by the character specified by replacement. The resulting string is returned.

Example

String s = "Hello".replace('l', 'w');

puts the string "Hewwo" into s.

The second form of replace( ) replaces one character sequence with another. It has this general form:

String replace(CharSequence original, CharSequence replacement)

D. trim( )

The trim() method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

Syntax:

String trim( )

Example:

String s = "   Hello World   ".trim();

This puts the string "Hello World" into s.

The trim( ) method is quite useful when you process user commands.

```
// Using trim() to process commands.
import java.io.*;
class UseTrim
{ public static void main(String args[]) throws IOException {
BufferedReader br = new BufferedReader(new
nputStreamReader(System.in));
String str;
System.out.println("Enter 'stop' to quit.");
System.out.println("Enter State: ");
do { str = br.readLine();
```


source diginotes.in

```
 str = str.trim();
if(str.equals("Illinois"))
System.out.println("Capital is Springfield.");
 else if(str.equals("Missouri"))
System.out.println("Capital is Jefferson City.");
else if(str.equals("California"))
System.out.println("Capital is Sacramento.");
else if(str.equals("Washington"))
 System.out.println("Capital is Olympia."); // ... }
while(!str.equals("stop"));
}
}
```

5. Data Conversion

1. The valueOf( ) method converts data from its internal format into a human-readable form.
2. It is a static method that is overloaded within String for all of Java's built-in types so that each type can be converted properly into a string.

3. valueOf( ) is also overloaded for type Object, so an object of any class type you create can also be used as an argument

Syntax:

```
static String valueOf(double num)
static String valueOf(long num)
 static String valueOf(Object ob)
static String valueOf(char chars[ ])
```

4. valueOf( ) is called when a string representation of some other type of data is needed. example, during concatenation operations.

5. Any object that you pass to valueOf( ) will return the result of a call to the object's toString( ) method.

6. There is a special version of valueOf() that allows you tospecify a subset of a char array.

Syntax:

```
static String valueOf(char chars[ ], int startIndex, int numChars)
```

7. Here, chars is the array that holds the characters, startIndex is the index into the array of characters at which the desired substring begins, and numChars specifies the length of the substring.

6. Changing Case of Characters

source diginotes.in

A. toLowerCase( )

1. converts all the characters in a string from uppercase to lowercase.

2. This method return a String object that contains the lowercase equivalent of the invoking String.

3. Non alphabetical characters, such as digits, are unaffected.

Syntax

String toLowerCase( )

B. toUpperCase()

1. converts all the characters in a string from lowercase to uppercase.
2. This method return a String object that contains the uppercase equivalent of the invoking String.

3. Non alphabetical characters, such as digits, are unaffected.

Syntax

String toUpperCase( )

```
class ChangeCase {
 public static void main(String args[]) {
String s = "This is a test.";
System.out.println("Original: " + s);
String upper = s.toUpperCase();
String lower = s.toLowerCase();
System.out.println("Uppercase: " + upper);
System.out.println("Lowercase: " + lower);
}
}
```

Output:

Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.

StringBuffer

StringBuffer is a peer class of String that provides much of the functionality of strings. As you know, String represents fixed-length, immutable character sequences.

StringBuffer represents growable and writeable character sequences.

StringBuffer may have characters and substrings inserted in the middle or appended to the end.

source diginotes.in

StringBuffer will automatically grow to make room for such additions and often has more characters pre allocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer defines these four constructors:

StringBuffer( )

StringBuffer(int size)

StringBuffer(String str)

StringBuffer(CharSequence chars)

a. The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
b. The second version accepts an integer argument that explicitly sets the size of the buffer.
c. The third version accepts a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

d. StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time.

A. length( ) and capacity( )

a. The current length of a StringBuffer can be found via the length( ) method, while the total allocated capacity can be found through the capacity( ) method.

Syntax

int length( )

int capacity( )

b. Example:

```
class StringBufferDemo
{
  public static void main(String args[])
  {
    StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
  }
```


source diginotes.in

}

Output

buffer = Hello

length = 5

capacity = 21

B. ensureCapacity( )

a. If you want to pre allocate room for a certain number of characters after a StringBuffer has been constructed, you can use ensureCapacity( ) to set the size of the buffer.
b. This is useful if you know in advance that you will be appending a large number of small strings to a StringBuffer.
Syntax

void ensureCapacity(int capacity)

Here, capacity specifies the size of the buffer.

C. setLength( )

a. To set the length of the buffer with in a StringBufferobject,

Syntax:

void setLength(int len)

Here, len specifies the length of the buffer. This value must be nonnegative.

When you increase the size of the buffer, null characters are added to the end of the existing buffer.

If you call setLength( ) with a value less than the current value returned by length(), then the characters stored beyond the new length will be lost.

D. charAt( ) and setCharAt( )

a. The value of a single character can be obtained from a StringBuffer via the charAt()method. You can set the value of a character within a StringBuffer using setCharAt( ).
b. Syntax

char charAt(int where)

void setCharAt(int where, char ch)

c. For charAt( ), where specifies the index of the character being obtained.
d. For setCharAt( ), where specifies the index of the character being set, and ch specifies the new value of that character.

source diginotes.in

```
// Demonstrate charAt() and setCharAt().
 class setCharAtDemo {
public static void main(String args[])
 { StringBuffer sb = new StringBuffer("Hello");
 System.out.println("buffer before = " + sb);
 System.out.println("charAt(1) before = " + sb.charAt(1));
sb.setCharAt(1, 'i');
sb.setLength(2);
System.out.println("buffer after = " + sb);
System.out.println("charAt(1) after = " + sb.charAt(1)); } }
```

Output

buffer before = Hello

charAt(1) before = e

buffer after = Hi

charAt(1) after = i

E. getChars( )

a. To copy a substring of a StringBuffer into an array, use the getChars( ) method.

Syntax

Syntax

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring.

b. This means that the substring contains the characters from sourceStart through sourceEnd–1.

c. The array that will receive the characters is specified by target.

The index within target which the substring will be copied is passed in targetStart.

d. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

F. append( )

1. The append() method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object. It has several overloaded versions. Here are a few of its forms:

StringBuffer append(String str)

source diginotes.in

StringBuffer append(int num)

StringBuffer append(Object obj)

2. The result is appended to the current StringBuffer object.
3. The buffer itself is returned by each version of append( ).
4. This allows subsequent calls to be chained together, as shown in the following example:

```
class appendDemo {
 public static void main(String args[])
 { String s; int a = 42;
   StringBuffer sb = new StringBuffer(40);
   s = sb.append("a = ").append(a).append("!").toString();
   System.out.println(s);
 }
}
```

Output

a = 42!

G. **insert( )**

1. The insert() method inserts one string in to another.
2. It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences.
3. Like append(),it calls String.valueOf() to obtain the string representation of the value it is called with.
4. This string is then inserted into the invoking StringBuffer object.
5. These are a few of its forms:

StringBuffer insert(int index, String str)

StringBuffer insert(int index, char ch)

StringBuffer insert(int index, Object obj)

Here, index specifies the index at which point the string will be inserted into the invoking StringBuffer object.

6. The following sample program inserts “like” between “I” and “Java”:

```
class insertDemo { public static void main(String args[]) {
   StringBuffer sb = new StringBuffer("I Java!");
  sb.insert(2, "like ");
```


source diginotes.in

```
        System.out.println(sb);
      }
    }
```

7. Output

   I like Java!

H. reverse( )

You can reverse the characters within a StringBuffer object using reverse( ), shown here:

StringBuffer reverse( )

This method returns the reversed object on which it was called.

The following program demonstrates reverse( )

```
 class ReverseDemo {
 public static void main(String args[])
 { StringBuffer s = new StringBuffer("abcdef");
System.out.println(s);
 s.reverse();
 System.out.println(s);
}
}
```

Output

abcdef

fedcba

I. delete( ) and deleteCharAt( )

You can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ).

Syntax:

StringBuffer delete(int startIndex, int endIndex)

StringBuffer deleteCharAt(int loc)

The delete( ) method deletes a sequence of characters from the invoking object.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)
source diginotes.in

Here, startIndex specifies the index of the first character to remove, and endIndex specifies an index one past the last character to remove.

Thus, the substring deleted runs from startIndex to endIndex–1. The resulting StringBuffer object is returned.

The deleteCharAt( ) method deletes the character at the index specified by loc. It returns the resulting StringBuffer object.

```
// Demonstrate delete() and deleteCharAt()

class deleteDemo { public static void main(String args[])

{ StringBuffer sb = new StringBuffer("This is a test.");

sb.delete(4, 7);

System.out.println("After delete: " + sb);

sb.deleteCharAt(0);

 System.out.println("After deleteCharAt: " + sb);

}

}
```

Output

After delete: This a test.

After deleteCharAt: his a test.

J. **replace( )**

a. You can replace one set of characters with another set inside a StringBuffer object by calling replace( ).
b. Syntax

StringBuffer replace(int startIndex, int endIndex, String str)

The substring being replaced is specified by the indexes startIndex and endIndex.

c. Thus, the substringatstartIndexthroughendIndex–1 is replaced.The replacement string is passed in str.

The resulting StringBuffer object is returned.

```
class replaceDemo {

public static void main(String args[])

{ StringBuffer sb = new StringBuffer("This is a test.");

sb.replace(5, 7, "was");

System.out.println("After replace: " + sb);

}
```


source diginotes.in

}

Here is the output:

After replace: This was a test.

K. **substring( )**

1. It has the following two forms:
   Syntax
   String substring(int startIndex)

   String substring(int startIndex, int endIndex)

2. The first form returns the substring that starts at startIndex and runs to the end of the invoking StringBuffer object.

3.The second form returns the substring that starts at startIndex and runs through endIndex–1.

These methods work just like those defined for String that were described earlier.

Difference between StringBuffer and StringBuilder.

1. J2SE 5 adds a new string class to Java's already powerful string handling capabilities. This new class is called StringBuilder.
2. It is identical to StringBuffer except for one important difference: it is not synchronized, which means that it is not thread-safe.
3. The advantage of StringBuilder is faster performance. However, in cases in which you are using multithreading, you must use StringBuffer rather than StringBuilder.

Additional Methods in String which was included in Java 5

1. **int codePointAt(int i)**

Returns the Unicode code point at the location specified by i.

2. **int codePointBefore(int i)**
   Returns the Unicode code point at the location that precedes that specified by i.
3. **int codePointCount(int start , int end)**
   Returns the number of code points in the portion of the invoking String that are between start and end– 1.
4. **boolean contains(CharSequence str)**
   Returns true if the invoking object contains the string specified by str . Returns false, otherwise.
5. **boolean contentEquals(CharSequence str)**
   Returns true if the invoking string contains the same string as str. Otherwise, returns false.
6. **boolean contentEquals(StringBuffer str)**
   Returns true if the invoking string contains the same string as str. Otherwise, returns false.
7. **static String format(String fmtstr , Object ... args)**

source diginotes.in

Returns a string formatted as specified by fmtstr.

8. **static String format(Locale loc , String fmtstr , Object ... args)**
   Returns a string formatted as specified by fmtstr.
9. **boolean matches(string regExp)**
   Returns true if the invoking string matches the regular expression passed in regExp. Otherwise, returns false.
10. **int offsetByCodePoints(int start , int num)**
    Returns the index with the invoking string that is num code points beyond the starting index specified by start.
11. **String replaceFirst(String regExp , String newStr)**
    Returns a string in which the first substring that matches the regular expression specified by regExp is replaced by newStr.
12. **String replaceAll(String regExp , String newStr)**
    Returns a string in which all substrings that match the regular expression specified by regExp are replaced by newStr
13. **String[ ] split(String regExp)**
    Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp.
14. **String[ ] split(String regExp , int max)**
    Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in regExp. The number of pieces is specified by max. If max is negative, then the invoking string is fully decomposed. Otherwise, if max contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If max is zero, the invoking string is fully decomposed.
15. **CharSequence subSequence(int startIndex , int stopIndex)**
    Returns a substring of the invoking string, beginning at startIndex and stopping at stopIndex . This method is required by the CharSequence interface, which is now implemented by String.

Additional Methods in StringBuffer which was included in Java 5

StringBuffer appendCodePoint(int ch)

Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.

int codePointAt(int i)

Returns the Unicode code point at the location specified by i.

int codePointBefore(int i)

Returns the Unicode code point at the location that precedes that specified by i.

int codePointCount(int start , int end)

Returns the number of code points in the portion of the invoking String that are between start and end– 1.

int indexOf(String str)

Searches the invoking StringBuffer for the first occurrence of str. Returns the index of the match, or –1 if no match is found.

int indexOf(String str , int startIndex)

Searches the invoking StringBuffer for the first occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.

source diginotes.in

int lastIndexOf(String str)
Searches the invoking StringBuffer for the last occurrence of str. Returns the index of the match, or –1 if no match is found.

int lastIndexOf(String str , int startIndex)
Searches the invoking StringBuffer for the last occurrence of str, beginning at startIndex. Returns the index of the match, or –1 if no match is found.

source diginotes.in

Servlet

Introdution to servlet

Servlet is small program that execute on the server side of a web connection. Just as applet extend the functionality of web browser the applet extend the functionality of web server.

In order to understand the advantages of servlet, you must have basic understanding of how web browser communicates with the web server.

Consider a request for static page. A user enters a URL into browser. The browser generates http request to a specific file. The file is returned by http response. Web server map this particular request for this purpose. The http header in the response indicates the content. Source of web page as MIME type of text/html.

1. What are the Advantage of Servlet Over "Traditional" CGI?

Java servlet is more efficient, easier to use, more powerful, more portable, and cheaper than traditional CGI and than many alternative CGI-like technologies. (More importantly, servlet developers get paid more than Perl programmers :-).

- **Efficient.** With traditional CGI, a new process is started for each HTTP request. If the CGI program does a relatively fast operation, the overhead of starting the process can dominate the execution time. With servlets, the Java Virtual Machine stays up, and each request is handled by a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are *N* simultaneous request to the same CGI program, then the code for the CGI program is loaded into memory N times. With servlets, however, there are *N* threads but only a single copy of the servlet class.
- **Convenient.** Hey, you already know Java. Why learn Perl too? Besides the convenience of being able to use a familiar language, servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such utilities.
- **Powerful.** Java servlets let you easily do several things that are difficult or impossible with regular CGI. For one thing, servlets can talk directly to the Web server (regular CGI programs can't). This simplifies operations that need to look up images and other data stored in standard places. Servlets can also share data among each other, making useful things like database connection pools easy to implement. They can also maintain

source: diginotes.in

information from request to request, simplifying things like session tracking and caching of previous computations.

- **Portable.** Servlets are written in Java and follow a well-standardized API. Consequently, servlets written for, say I-Planet Enterprise Server can run virtually unchanged on Apache, Microsoft IIS, or Web Star. Servlets are supported directly or via a plug in on almost every major Web server.
- **Inexpensive.** There are a number of free or very inexpensive Web servers available that are good for "personal" use or low-volume Web sites. However, with the major exception of Apache, which is free, most commercial-quality Web servers are relatively expensive. Nevertheless, once you have a Web server, no matter the cost of that server, adding servlet support to it (if it doesn't come preconfigured to support servlets) is generally free or cheap.

2. What is servlet? What are the phases of servlet life cycle? Give an example.

Servlets are small programs that execute on the server side of a web connection. Just as applet extend the functionality of web browser the applet extend the functionality of web server.

Servlet class is loaded.

Servlet class is loaded when first request to web container.

servlet instance is created:

Web container creates the instance of servlet class only once.

init method is invoked:

It class the init method when it loads the instance. It is used to intialise servlet.

Syntax of init method is

public void init(ServletConfig config) throws ServletException

Service method is invoked:

Web container calls service method each time when request for the servlet is received. If servlet is not initialized it calls init then it calls the service method. Syntax of service method is as follows

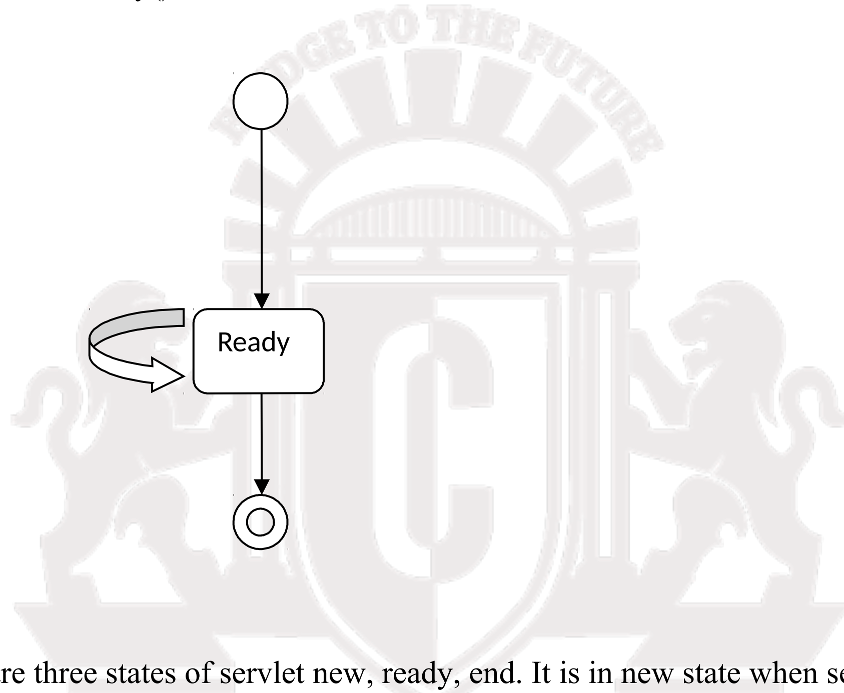
source: diginotes.in

public void service(Servlet request, ServletResponse response) throws ServletException, IOException

Destroy method is invoked.

The web container calls the destroy method before it removes the servlet from service. It gives servlet an opportunity to clean up memory, resources etc. Servlet destroy method has following syntax.

public void destroy().



There are three states of servlet new, ready, end. It is in new state when servlet is created. The servlet instance is created when it is in new state. After invoking the init () method servlet comes to ready state. In ready state servlet invokes destroy method it comes to end state.

3. Explain about deployment descriptor

Deployment descriptor is a file located in the WEB-INF directory that controls the behavior of a java servlet and java server pages. The file is called the web.xml file and contains the header, DOCTYPE, and web app element. The web app element should contain a servlet element with three elements. These are servlet name, servlet class, and init-param.

The servlet name elements contain the name used to access the java servlet. The servlet class is the name of the java servlet class. init-param is the name of an initialization parameter that is used when request is made to the java servlet.

Example file:

<?xml version=”1.0” encoding-“ISO-8859=1”?>…..XML header

source: diginotes.in

```
< !DOCTYPE web-app PUBLIC "~//Sun Microsystems, Inc.?? DTD Web Application2.2//EN"> ..doctype

<web-app>

        <servlet>

        <servlet-name>MyJavaservlet</servlet-name>

        <servlet-class>myPackage.MyJavaservletClass</servlet-class>

        <init-param><param-name>parameter1</param-name>

                <param-value>735</param-value>

        </init-param>

        </servlet>

</web-app>
```

4. How to read data from client in servlet?

- A client uses either the GET or POST method to pass information to a java servlet. Depending on the method used by the client either doGet() or doPost() method is called in servlet.
- Data sent by a client is read into java servlet by calling getParameter() method of HttpservletRequest() object that instantiated in the argument list of doGet() method and doPost() method.
- getParameter() requires one argument, which is the name of parameter that contains the data sent by the client. getParameter() returns the String object.
- String object contains the value assigned by the client. An empty string object is returned when it does not assign a value to the parameter. Also a null is retuned when parameter is not returned in the client.
- getParameterValues() used to return the array of string objects.

Example code

Html code that calls a servlet:

```
<FORM  ACTION="/servlet/myservlets.js2">
Enter Email Address :< INPUT TYPE="TEXT" NAME="email">
<INPUT TYPE="SUBMIT">
</FORM>
```


source: diginotes.in

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class js2 extends Httpservlet {
public void doGet(HttpservletRequset request,HttpservletResponse response)
throws servletException , IOException {
//String email;
//Email=request.getParameter("email");
Respose.setContentType("text/html");
PrinterWriter  pw=response.getWriter();
pw.println("<HTML>\n" +
"HEAD><TITLE> Java Servlet</TITLE></HEAD>\n" +
"<BODY>\n"+
//"<p>MY Email Address :" +email +"</p>\n" +
<h1> My First Servlet
"</BODY>\n" +
</HTML>");
}
}
```

5. How to read HTTP Request Headers?

A request from client contains two components these are implicit data, such as email address explicit data at HTTP request header. Servlet can read these request headers to process the data component of the request.

Example of HTTP header:
Accept: image.jpg, image.gif,*/*
Accept- Encoding: Zip
Cookie: CustNum-12345
Host:www.mywebsite.com
Referer: http://www.mywebsite.com/index.html

The uses of HTTP header:
Accept: Identifies the mail extension
Accept-Charset : Identifies the character set that can be used by browser.
Cookie returns the cookies to server.
Host: contains host portal.
Referrer: Contains the URL of the web page that is currently displayed in the browser.

A java servlet can read an HTTP request header by calling the getHeader() method of the HttpservletRequest object. getHeader() requires one argument which is the name of the http request header.

getHeader()

source: diginotes.in

6. **How to send data to client and writing the HTTP Response Header?**

A java servlet responds to a client's request by reading client data and HTTP request headers, and then processing information based on the nature of the request.

For example, a client request for information about merchandise in an online product catalog requires the java servlet to search the product database to retrieve product information and then format the product information into a web page, which is returned to client.

There are two ways in which java servlet replies to client request. These are sent by sending information to the response stream and sending information in http response header. The http response header is similar to the http request header.

Explicit data are sent by creating an instance of the PrintWriter object and then using println() method to transmit the information to the client.

Implcit data example: HTTP/1.1 200 OK
Content-Type:text/plain

My Response

Java servlet can write to the HTTP response header by calling setStatus() method requires one argument which is an integer that represent the status code.

```
Response.setStatus(100);
```

7. **Explain about Cookies in servlet.**

Cookies are text files stored on the client computer and they are kept for various information tracking purpose. Java Servlets transparently supports HTTP cookies.

There are three steps involved in identifying returning users:

- Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

Setting Cookies with Servlet:

Setting cookies with servlet involves three steps:

(1) Creating a Cookie object: You call the Cookie constructor with a cookie name and a cookie value, both of which are strings.

```
Cookie cookie = new Cookie("key","value");
```


source: diginotes.in

(2) Setting the maximum age: You use setMaxAge to specify how long (in seconds) the cookie should be valid. Following would set up a cookie for 24 hours.

cookie.setMaxAge(60*60*24);

(3) Sending the Cookie into the HTTP response headers: You use **response.addCookie** to add cookies in the HTTP response header as follows:

response.addCookie(cookie);

Writing Cookie

```
import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

public class HelloForm extends HttpServlet {

  public void doGet(HttpServletRequest request,

                   HttpServletResponse response)

            throws ServletException, IOException

  {

     Cookie myCookie = new Cookie("user id", 123);

    myCookie.setMaxAge(60*60);

     response.addCookie(myCookie );

     response.setContentType("text/html");

     PrintWriter out = response.getWriter();

         out.println( "<html>\n" +

              "<head><title>" + My Cookie + "</title></head>\n" +

           "<nody>\n" +

           "<h1>+ My Cookie +"<h1>\n" +

           "<p> Cookie Written + </p>\n" +

           "</body></HTML>");

}

 }
```


source: diginotes.in

Reading Cookies with Servlet:

To read cookies, you need to create an array of *javax.servlet.http.Cookie* objects by calling the **getCookies( )** method of *HttpServletRequest*. Then cycle through the array, and use getName() and getValue() methods to access each cookie and associated value.

Example:Let us read cookies which we have set in previous example:

```
import java.io.*;import javax.servlet.*;import javax.servlet.http.*;
 public class ReadCookies extends HttpServlet {
 public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
  {
     Cookie cookie;
        Cookie[]  cookies;
     cookies = request.getCookies();
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
     String title = "Reading Cookies Example";
     out.println( "<html>\n" +
          "<head><title>" + title + "</title></head>\n" );
             if( cookies != null ){
         out.println("<h2> Found Cookies Name and Value</h2>");
          for (int i = 0; i < cookies.length; i++){
            cookie = cookies[i];
           out.print("Name : " + cookie.getName( ) + ",  ");
        out.print("Value: " + cookie.getValue( )+" <br/>");
       }
```


source: diginotes.in

```
    }else{    out.println( "<h2>No cookies founds</h2>");
    }      out.println("</body>");      out.println("</html>");
  }}
```

8. Explain Session Tracking:

1. A session is created each time a client requests service from a java servlet. The java. The java servlet processes the request and response accordingly, after which the session is terminated. Many times the same client follows with another request to the same client follows with another request to the same java servlet, java servlet requires information regarding the previous session to process request.
2. However , HTTP is stateless protocol, meaning that there is not hold over from the previous sessions.
3. Java servlet is capable of tracking sessions by using Httpsession API. It determines if the request is a continuation from an existing session or new session.
4. A java servlet calls a getSession() method of HttpservletRequset object, which returns a session object if it is a new session. The getSession() method requires one argument which is Boolean true. Returns session object.

Syntax :

```
HttpSession s1=request.getSession(true);
```

<u>JSP program</u>

A jsp is java server page is server side program that is similar in design and functionality to a java servlet.

A JSP is called by a client to provide web services, the nature of which depends on client application.

A jsp is simpler to create than a java servlet because a jsp is written in HTML rather than with the java programming language. . There are three methods that are automatically called when jsp is requested and when jsp terminates normally. These are the jspInit() method , the jspDestroy()m ethod and service() method.

A jspInit() is identical ti init() method of java servlet. It is called when first time jsp is called.

source: diginotes.in

A jspDestroy() is identical to destroy() method of servlet. The destroy() method is automatically called when jsp erminates normally.It is not called when jsp terminates abruptly. It is used for placing clean up codes.

1. **Explain JSP tags(repeated question)**

A jsp tag consists of a combination of HTML tags and JSP tags. JSP tags define java code that is to be executed before the output of jsp program is sent to the browser.

A jsp tag begin with a <%, which is followed by java code , and wnds with %>,

There ia an XML version of jsp tag <jap:TagId></jsp:TagId>

A jsp tags are embedded into the HTML component of a jsp program and are processed by Jsp virtual engine such as Tomcat.

Java code associated with jsp tag are executed and sent to browser.

There are five types of jsp tags :

Comment tag :A comment tag opens with <%-- and close with --%> and is follwed by a comment that usually describes the functionality of statements that follow a comment tag.

Declaration statement tags: A declartion statement tag opens with <%! And is followed by declaration statements that define the variables, object, and methods that are avilabe to other component of jsp program.

Directive tags: A directive tag opens with <%@ and commands the jsp virtual engine to perform a specific task, such as importing java package required by objects and methods used in a declaration statement. The directive tag closes with %> . There are commonly used in directives import, include , and taglib. The import tag is used to import java packages into the jsp program. Include is used for importing file. Taglib is used for including file.

(SOURCE DIGINOTES)

Example:

```
<%@ page import="import java.sql.*" ; %>

<%@  include file="keogh\books.html"  %>

<%@ taglib url="myTags.tld" ; %>
```

Expression tags: An expression tag opens with <%= and is used for an expression statement whose result page replaces the expression tag when the jsp virtual engine resolves JSP tags. An expression tag closes with %>

source: diginotes.in

Scriptlet tag: A sciptlet tag opens with <% and contains commonly used java control statements and loops. And Scriptlet tag closes with %>

2. **How variables and objcts declared in JSP program?**

   You can declare java variables and objects that are used in a JSP program by using the same codin technique used to declare them in java. JSP declaration statements must appear as jsp tag

   Ex:

```
<html>
<head>
        <title> Jsp Programming </title>
<.head>
<body>
        <%! Int age=29; %><p> Your age is : <%=age%> </p>
</body>
</html>
```

3. **How method are declared and used in jsp programs?**

   Methods are defined same way as it is defined in jsp program, except these are placed in JSP tag.methods are declared in JSP decalration tag. The jsp calls method in side the expression tag.

   Example:

```
<html>
<head>
        <title> Jsp programming</title>
</head>
<body>
<%! int add(int n1, int n2)
        {
        int c;
        c=a+b;
        return c;
```


(SOURCE DIGINOTES)
source: diginotes.in

```
        }
    %>
    <p> Addition of two numbers : <%= add(45,46)%> </p>
    </body></html>
```

4. Explain the control statements of JSP with example program:

1. One of the most powerful features avilable in JSP is the ability to change the flow of the program to truly create dynamic content for a web based on conditions received from the browser.
2. Ther are two control statements used to change the flow of program are “if “ and “switch” statement , both of which are also used to direct the flow of a java program.

   Ex:

```
<html>
<head>
        <title> JSP Programming </title>
</head>
<body>
        <%! int grade=26; %>
        </body>
<% if(grade >69) { %>
        <p> You Passed !</p>
<% } else { %>
<p> Better Luck Next Time</p>
<% } %>
</body>
</html>
```

5. Looping Statement of JSP

Jsp loops are nearly identical to loops that you use in your java program except you can repeat the html tags

There are three kind of jsp loop that are commonly used in jsp program.

Ex: for loop , while loop , do while .

Loop plays an important role in JSP database program. The following program is example for “FOR LOOP”.

```
<html><head><title>For Loop Example</title></head>
<body>
```


source: diginotes.in

```
<%
   for (int i = 0; i < 10;i++) {
%>
<p> Hello World</p>
<%  }   %> </body>
</html>
```

6. Explain Requset String generated by browser. how to read a request string in jsp?

1. A browser generate requst string whenever the submit button is selected. The user requests the string consists of URL and the query the string.
   Example of request string:
   http://www.jimkeogh.com/jsp/?fname=" Bob" & lname ="Smith"
2. Your jsp programneeds to parse the query string to extract the values of fields that are to be processed by your program. You can parse the query string by using the methods of the request object.
3. getParameter(Name) method used to parse a value of a specific field that are to be processed by your program
4. code to process the requset string
   ```
   <%! String FirstName =requst.getParameter(fname);
       String  LastName =requst.getParameter(lname); %>
   ```
5. Copying from multivalued field such as selection list field can be tricky multivalued fields are handled by using getParameterValues()
6. Other than requset string url has protocols, port no, the host name
7. **Write the JSP program to create and read cookie called "EMPID" and that has value "AN2536".**

Cookie is small piece of information created b a JSP program that is stored on the clien's hard dik by the browser. Cookie isare used to store various kind of information, such as user preference. The cookies are created by using Cookie class.

Create cookie:

```
<html>
<head>
<title> creating cookie</title>
</head>
<body>
<%! String MyCookieName="EMPID";
```


source: diginotes.in

```
        String UserValue="AN2536";
%>
</body>
</html>
```

Reading Cookie:

```
<html>
<head>
<title>reading cookie </title>
</head>
<body>
<% String myCookieName="EMPID";
String myCookieValue;
String CName, CValue;
int found=0;
Cookie[] cookies=request.getCoookies();
        for( int i=0;i<cookies.length;i++) {
        CName= cookies[i].getName();
        CValue =cookies[i].getValue();
        If(myCookieName.equals(CName)) {
        found=1;
        myCookieValue=Cvalue; } }
        If(found== 1)  { %>
        <p> Cookie Name =   < %= CName %> </p>
        <p> Cookie Value =   < %= CValue %> </p>
        <%  } %> </body></html>
```

8. Explain steps to configure tomcat.

i. Jsp program programs are executed by a JSP virtual machine that run on a web server.
ii. We can download and install JSP virtual machine.
iii. Installation Steps

Connect to Jakarta.apache.org.
Select down load
Select Binaries to display the binary Download Page.
Create a folder from the root directory called tomcat.
Download latest release.
Unzip Jakarta-tomcat.zip.
The extraction process creates the following folder in the tomcat directory: bin, conf, doc, lib, src, and webapps
Modify the batch file , which is located in the \tomcat\bin folder. Change the JAVA_HOME variable is assigned the pathe where JDK is installed on your computer.
Open dos window and type \tomcat\bin\tomcat to start Tomcat.
Open your browser. Enter http://localhost:8080.

source: diginotes.in

Tomcat home page is displayed on the screen verifying that Tomcat is running.

9. Explain how session objects are created.

A JSP database system ia able to share information among JSP programs within a session by using a session object. Each time a session is created , a unique ID is assigned to the session and stored as acookie.

A unique ID enables JSP program to track multiple session session simultaneously while maintaining data integrity of each session. The session ID is used to prevent the intermingling of each session.

Create session Object:

```
<html><head><title> Jsp Session</title></head>
<body>
<% ! String AtName="Product";
      String AtValue ="1234";
      Session.setAttributes(AtName, AtValue);
%></body></html>
```

In session object we can store information about purchases as session attributes can be retrived and modified each time the jsp program runs. setAttributes() used for creating attributes.

Read Session Object:

getAttributeNames() methos returns names of all the attribbutes as Enumeration , the attributes are processsed.

```
<html><head><title> Jsp Session</title></head>
<body><% !
      Enumaration purchases=session.getAttributeNames();
      String AtName=(String) attributeNames.nextElement();
      String AtValu=(String) session.getAttribute(AtName); %>
      <p> Attirbute Name <%= AtName %>  </p>
      <p> Attribute Value <% = Atvalue %> </p>
      <% } %> %></body></html>
```


source: diginotes.in

III Internal Questions

1. What are different types of JSP tags describe the JSP tags with example.(Dec 2011)
2. Define JSP. Explain two types of control statements with example.(Dec 2012)
3. Write the JSP program to create and read cookie called "EMPID" and that has value "AN2536"(Dec 2012)
4. What is RMI? Briefly explain working of RMI in java.(Dec 2012)
5. Department has set the grade for the subject Java as follows:

   Above 90: A, 80 - 89: B , 70 -79 : C
   Below 70 = fail. Sham enters his marks for the subject Java in the interface provided.
   Write a JSP program to accept the mark and display the grade.(Jun 2011)

6. Briefly explain the RMI in Java (June 2011)

7. Discuss different types of JSP tags (Jun 2011)

8. Write a program using RMI such as client and server program in which client sends hello message to server and replies to client (June 2012)

9. Develop simple java servlet that handle HTTP Request and Response (June 2012)

10. Explain javax.servlet packages(June 2012)

11. What is difference between JSP and Servlet? (june 2012)

12. What are the advantages of JSP program?(jun 2010)

13. What are servlets? Briefly explain the application of servlets in web programming (dec 2010)

14. Explain the life cycle of a servlet. (dec 2010)
15. Write a java servlet which reads two parameters from the webpage, say value 1 and value 2 , which are type integer and finds the sum of the two value and return back the result as a webpage.(dec 2010)

16. Provide java syntax for the following: (dec 2010)
    i) Handling HTTP requests and responses
    ii) Using cookies
    iii) Session tracking
17. List out difference between CGI and servlet.
18. What is cookie list out methods defined by cookie. Write a servlet program to read cookie.
19. Write a jsp program to add cookie name "User Id" and value"JB007"
20. Describe in detail how tomcat web server is configured in develop of servlet life cycle.

source: diginotes.in

Unit -5

The Concept of JDBC:

1. Java was not considered industrial strength programming language since java was unable to access the DBMS.
2. Each dbms has its own way to access the data storage. low level code required to access oracle data storage need to be rewritten to access db2.

3. JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases

4. JDBC drivers has to do the following

    - Open connection between DBMS and J2EE environment.

    - Translate low level equivalents of sql statements sent by J2EE component into messages that can be processed by the DBMS.

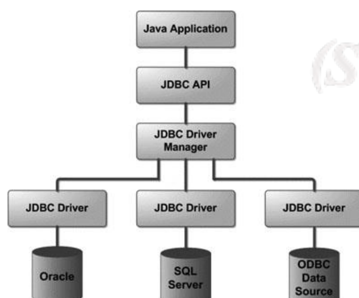
    - Return the data that conforms to JDBC specifications to the JDBC driver

    - Return the error messages that conforms to JDBC specifications to the JDBC driver

    - Provides transaction management routines that conforms to JDBC specifications to the JDBC driver
    - Close connection between the DBMS and the J2EE component.

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

JDBC Architecture



June 2012 (Briefly discuss the various JDBC driver types 10 M)

JDBC Driver Types

Type 1 driver JDBC to ODBC Driver

1. It is also called JDBC/ODBC Bridge , developed by MicroSoft.
2. It receives messages from a J2EE component that conforms to the JDBC specifications
3. Then it translates into the messages understood by the DBMS.
4. This is DBMS independent database program that is ODBC open database connectivity.

Type 2 JAVA / Native Code Driver

1. Generates platform specific code that is code understood by platform specific code only understood by specific databases.
2. Manufacturer of DBMS provides both java/ Native code driver.
3. Using this provides lost of portability of code.
4. It won't work for another DBMS manufacturer

Type 3 JDBC Driver

1. Most commonly used JDBC driver.
2. Coverts SQL queries into JDBC Formatted statements.
3. Then JDBC Formatted statements are translated into the format required by the DBMS.
4. Referred as Java protocol

Type 4 JDBC Driver

1. Referred as Type 4 database protocol
2. SQL statements are transferred into the format required by the DBMS.
3. This is the fastest communication protocol.

JDBC Packages

JDBC API contains two packages. First package is called java.sql, second package is called javax.sql which extends java.sql for advanced JDBC features.

Explain the various steps of the JDBC process with code snippets.

1. Loading the JDBC driver

    - The jdbc driver must be loaded before the J2EE compnet can be connected to the database.
    - Driver is loaded by calling the method and passing it the name of driver

    Class.forName("sun:jdbc.odbc.JdbcOdbcDriver");

2. Connecting to the DBMS.

    - Once the driver is loaded , J2EE component must connect to the DBMS using DriverManager.getConnection() method.
    - It is highest class in hierarchy and is responsible for managing driver information.

- It takes three arguments URL, User, Password
- It returns connection interface that is used through out the process to reference a database

```
String url="jdbc:odbc:JdbcOdbcDriver";
String userId="jim"
String password="Keogh";
Statement DatRequest;
Private Connection db;

try{
Class.forName("sun:jdbc.odbc.JdbcOdbcDriver");
Db=DriverManager.getConnection(url,userId,password);
}
```

3. Creating and Executing a statement.
   - The next step after the JDBC is loaded and connection is successfully made with a particular database managed by the dbms, is to end a particular query to the DBMS for processing.
   - SQL query consists series of SQL command that direct DBMS to do something example Return rows.
   - Connect.createStatement() method is used to create a statement Object.
   - The statement object is then used to execute a query and return result object that contain response from the DBMS

```
Statement DataRequest;
ResultSet Results;
try {
        String query="select * from Customers";
        DataRequest=Database.createStatement();
        Results= DataRequests.executeQuery(query);
    }
```

4. Processing data returned by the DBMS
   - **java.sql.ResultSet** object is assigned the result received from the DBMS after the query is processed.
   - **java.sql.ResultSet** contain method to interct with data that is returned by the DBMS to the J2EE Component.

```
Results= DataRequests.executeQuery(query);
do
{
Fname=Results.getString(Fname)
}
While(Results.next())
```

In the above code it return result from the query and executes the query. And getString is used to process the String retrived from the database.

5. Terminating the connection with the DBMS.
   To terminate the connection Database.close() method is used.

With proper syntax, explain three types of getConnection() method.

1 After the JDBC driver is successfully loaded and registered, the J2EE component must connect to the database. The database must be associated with the JDBC driver.

2 The datasource that JDBC component will connect to is identified using the URL format. The URL consists of three format.
- These are jdbc which indicate jdbc protocol is used to read the URL.
- <subprotocol> which is JDBC driver name.
- <subname> which is the name of database.

3 Connection to the database is achieved by using one of three getConnection() methods. It returns connection object otherwise returns SQLException

4 Three getConnection() method
- getConnection(String url)
- getConnection(String url, String pass, String user)
- getConnection(String url, Properties prop)

5 getConnection(String url)
- Sometimes the DBMS grant access to a database to anyone that time J2EE component uses getConnection(url) method is used.

```
String url=" jdbc:odbc:JdbcOdbcDriver ";
try{
Class.forName("sun:jdbc.odbc.JdbcOdbcDriver");
Db=DriverManager.getConnection(url);
}
```

6 getConnection(String url, String pass, String user)

- Database has limited access to the database to authorized user and require J2EE to supply user id and password with request access to the database. The this method is used.

```
try{
Class.forName("sun:jdbc.odbc.JdbcOdbcDriver");
Db=DriverManager.getConnection(url,userId,password);
}
```

7 getConnection(String url, Properties prop)
- There might be occasions when a DBMS require information besides userid and password before DBMS grant access to the database.

- This additional information is called properties and that must be associated with Properties object.
- The property is stored in text file. And then loaded by load method of Properties class.

```
Connection db;
Properties props=new Properties();
try {
 FileInputStream inputfile=new FileInputStream("text.txt");
Prop.load(inputfile);
}
```

Write short notes on Timeout:

1. Competition to use the same database is a common occurrence in the J2EE environment and can lead to performance degradation of J2EE application
2. Database may not connect immediately delayed response because database may not available.
3. Rather than delayed waiting timeJ2EE component can stop connection. After some time. This time can bet set with the following method: **DriverManager.setLoginTimeout(int sec).**
4. **DriverManager.getLoginTimeout(int sec)** return the current timeout in seconds.

Explain Connection Pool

1. Client needs frequent that needs to frequently interact with database must either open connection and leave open connection during processing or open or close and reconnect each time.
2. Leaving the connection may open might prevent another client from accessing the database when DBS have limited no of connections. Connecting and reconnecting is time consuming.
3. The release of JDBC 2.1 Standard extension API introduced concept on connection pooling
4. A connection pool is a collection of database connection that are opened and loaded into memory so these connection can be reused with out reconnecting to the database.
5. DataSource interface to connect to the connection pool. connection pool is implemented in application server.
6. There are two types of connection to the database 1.) Logical 2.) Physical
7. The following is code to connect to connection pool.

```
Context ctext= new IntialContext()
DataSource pool =(DataSource) ctext.lookup("java:comp/env/jdbc/pool");
Connection db=pool.getConnection();
```

Briefly explain the Statement object. Write program to call a stored procedure.(10)

1. Statement object executes query immediately with out precompiling.
2. The statement object contains the **excuteQuery()** method , which accept query as argument then query is transmitted for processing.It returns ResultSet as object.

Example Program

```
String url=”jdbc:odbc:JdbcOdbcDriver”;
String userId=”jim”
String password=”Keogh”;
Statement DatRequest;
Private Connection db;
ResultSet rs;

// code to load driver
//code to connect to the database
 try{
String query=”SELECT * FROM Customers;
DatRequest=Db.createStaement();
rs=DatRequest.executeQuery(query);// return result set object
}catch(SQLException  err)
{
System.err.println(“Error”);
System.exit(1);
}
```

3. Another method is used when DML and DDL operations are used for processing query is executeUpdate(). This returns no of rows as integer.

```
try{
String query=”UPDATE Customer set PAID=’Y’ where BALANCE =’0’;
DatRequest=Db.createStaement();
int  n=DatRequest.executeUpdate(query);// returns no of rows updated
}catch(SQLException  err)
{
System.err.println(“Error”);
System.exit(1);
}
```

Briefly explain the prepared statement object. Write program to call a stored procedure.(10)

1. A SQL query must be compiled before DBMS processes the query. Query is precompiled and executed using Prepared statements.
2. Question mark is placed as the value of the customer number. The value will be inserted into the precompiled query later in the code.

3. Setxxx() is used to replace the question mark with the value passed to the setxxx() method . xxx represents data type of the field.
   Example if it is string then setString() is used.
4. It takes two arguments on is position of question mark and other is value to the filed.
5. This is referred as late binding.

```
String url=”jdbc:odbc:JdbcOdbcDriver”;
String userId=”jim”
String password=”Keogh”;
ResultSet rs;

// code to load driver
//code to connect to the database
 try{

String query=”SELECT * FROM Customers where cno=?”;
PreparedStatement    pstatement=db.preparedStatement(query);
pstatement.setString( 1,”123”); // 1 represents first place holder, 123 is value
rs= pstatement.executeQuery();


}catch(SQLException  err)
{
System.err.println(“Error”);
System.exit(1);
}
```

Briefly explain the callable statement object. Write program to call a stored procedure.(10)

1. The callableStatement object is used to call a stored procedure from with in J2EE object. A stored procedure is block of code and is identified by unique name. the code can bewritten in Transact-C ,PL/SQL.

2. Stored procedure is executed by invoking by the name of procedure.

3. The callableStatement uses three types of parameter when calling stored procedure. The parameters are IN ,OUT,INOUT.

4. IN parameter contains data that needs to be passed to the stored procedure whose value is assigned using setxxx() method.

5. OUT parameter contains value returned by stored procedure.the OUT parameter should be registers by using registerOutParameter() method and then later retrieved by the J2EE component using getxxx() method.

6. INOUT parameter is used to both pass information to the stored procedure and retrieve the information from the procedure.

7. Suppose, you need to execute the following Oracle stored procedure:

```
CREATE OR REPLACE PROCEDURE getEmpName
    (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
    BEGIN
     SELECT first INTO EMP_FIRST   FROM Employees   WHERE ID = EMP_ID;
    END;
```

8. The following code snippets is used

```
CallableStatement cstmt = null;
    try {    String SQL = "{call getEmpName (?, ?)}";
             cstmt = conn.prepareCall (SQL);catch (SQLException e) { }
```

9. Using CallableStatement objects is much like using PreparedStatement objects. You must bind values to all parameters before executing the statement, or you will receive an SQLException.
10. If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.
11. When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type to the data type the stored procedure is expected to return.
12. Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

ResultSet

1. ResultSet object contain the methods that are used to copy data from ResultSet into java collection object or variable for further processing.

2. Data in the ResultSet is logically organized into the virtual table for further processing. Result set along with row and column it also contains meta data.

3. ResultSet uses virtual cursor to point to a row of the table.

4. J2EE component should use the virtual cursor to each row and the use other methods of the ResultSet to object to interact with the data stored in column of the row.

5. The virtual cursor is positioned above the first row of data when the ResultSet is returned by executeQuery () method.

6. The virtual cursor is moved to the first row with help of next() method of ResultSet

7. Once virtual cursor is positioned getxxx() is used to return the data. Data type of data is represents by xxx. It should match with column data type.

8. getString(fname) …..fname is column name.

9. setString(1)……….. in this 1 indicates first column selected by query.

```
stmt = conn.createStatement();

String sql;

sql = "SELECT id, first, last, age FROM Employees";

ResultSet rs = stmt.executeQuery(sql);

while(rs.next()){

  int id  = rs.getInt("id");/ / rs.getInt(1);

  int age = rs.getInt("age");

  String first = rs.getString("first");

  String last = rs.getString("last");

  System.out.print("ID: " + id);

  System.out.print(", Age: " + age);
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
            System.out.print(", First: " + first);

            System.out.println(", Last: " + last);

        }
```

Explain the with an example Scrollable Result Set (6 Marks)

1. Until the release of JDBC 2.1 API , the virtual cursor can move only in forward directions. But today the virtual cursor can be positioned at a specific row.
2. There are six methods to position the cursor at specific location in addition to next() in scrollable result set. firs() ,last(), absolute(), relative(), previous(), and getRow().
3. first() ………… position at first row.
4. last()…………position at last row.
5. previous()……….position at previous row.
6. absolute()………. To the row specified in the absolute function
7. relative()………… move relative to current row. Positive and negative no can be given.
   Ex. relative(-4) … 4 position backward direction.
8. getRow() ……… returns the no of current row.
9. There are three constants can be passed to the createStatement()
10. Default is TYPE_FORWARD_ONLY. Otherwise three constant can be passed to the create statement 1.) TYPE_SCROLL_INSENSITIVE

    2.) TYPE_SCROLL_SENSITIVE

11. TYPE_SCROLL makes cursor to move both direction. INSENSITIVE makes changes made by J2EE component will not reflect. SENSITIVE means changes by J2EE will reflect in the result set.
    Example code.

    ```
    String sql=" select * from emp";
    DR=Db.createStatement(TYPE_SCROLL_INSENSITIVE);
    RS= DR.executeQuery(sql);
    ```
12. Now we can use all the methods of ResultSet.

Explain the with an example updatable Result Set.

1. Rows contained in the result set is updatable similar to how rows in the table can be updated. This is possible by sending CONCUR_UPDATABLE.
2. There are three ways in which result set can be changed. These are updating row , deleting a row, inserting a new row.
3. **Update ResultSet**

- Once the executeQuery() method of the statement object returns a result set. updatexxx() method is used to change the value of column in the current row of result set.
- It requires two parameters, position of the column in query. Second parameter is value
- updateRow() method is called after all the updatexxx() methods are called.

  Example:

```
try{
        String query= "select Fname, Lname from Customers
        where Fname= 'Mary' and Lanme='Smith';
        DataRequest= Db.
        createStatement(ResultSet.CONCUR_UPDATABLE);
        Rs= DataRequest.executeQuery(query);
        Rs.updateString("LastName","Smith");
        Rs.updateRow();
 }
```

4. **Delete row in result set**
   - ❖ By using absolute method positioning the virtual cursor and calling deleteRow(int n) n is the number of rows to be deleted.
   - ❖ Rs.deleteRow(0) current row is deleted.
5. **Insert Row in result set**
   - ❖ Once the executeQuery() method of the statement object returns a result set. updatexxx() method is used to insert the new row of result set.
   - ❖ It requires two parameters, position of the column in query. Second parameter is value
   - ❖ insertRow() method is called after all the updatexxx() methods are called.

```
try{
        String query= "select Fname, Lname from Customers
        where Fname= 'Mary' and Lanme='Smith';
        DataRequest= Db.
        createStatement(ResultSet.CONCUR_UPDATABLE);
        Rs= DataRequest.executeQuery(query);
        Rs.updateString(1 ,"Jon");
        Rs.updateString(2 ,"Smith");
        Rs.insertRow();
 }
```

6. Whatever the changes making will affect only in the result set not in the table. To update in the table have to execute the DML(update, insert, delete) statements.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

Explain the Transaction processing with example

1. A transaction may consists of a set of SQL statements, each of which must be successfully completed for the transaction to be completed. If one fails SQL statements successfully completed must be rolled back.
2. Transaction is not completed until the J2EE component calls the commit() method of the connection object. All SQL statements executed prior to the call to commit() method can be rolled back.
3. Commit() method was automatically called in the program. DBMS has set AutoCommit feature.
4. If the J2EE component is processing a transaction then it has to deactivate the auto commit() option false.

```
try {
        DataBase.setAutoCommit(false)
        String query="UPDATE Customer set Street ='5  main Street' "+
              "WHERE FirstName ='Bob' ";
        DR= DataBase.createStatement();
        DataRequest=DataBase.createStatement();
DataRequest.executeUpdate(query1);
DataBase.commit();
}
```

5. Transaction can also be rolled back. When not happened. Db.rollback().
6. A transaction may consists of many tasks , some of which no need to roll back . in such situation we can create a savepoints, in between transactions. It was introduced in JDBC 3.0. save points are created and then passed as parameters to rollback() methods.
7. releseSavepint() is used to remove the savepoint from the transaction.
8. Savepoint s1=DataBase.setSavePoint("sp1");to create the savepoint.
9. Database.rollback(sp1); to rollback the transaction.

Batch Execution of transaction

10. Another way to combine sql statements into a single into a single transaction and then execute the entire transaction .
11. To do this the addBatch() method of statement object. The addBatch() method receives a SQL statement as a parameter and places the SQL statement in the batch.
12. executeBatch() method is called to execute the entire batch at the same time. It returns an array that contains no of SQL statement that execute successfully.

```
String query1="UPDATE Customers SET street =' 5 th Main'" +
        "Where Fname='BoB' ";
String query2="UPDATE Customers SET street =' 10 th Main'" +
        "Where Fname='Tom' ";
Statement DR=DB.createStatement();
```

```
DR.addBatch(query1);
DR.addBatch(query2);
int [] updated= DR.executeBatch();
```

Write notes on metadata interface Metadata

1. Metadata is data about data. MetaData is accessed by using the DatabaseMetaData interface.
2. This interface is used to return the meta data information about database.
3. Meta data is retrieved by using getMetaData() method of connection object.
   Database metadata
   The method used to retrieve meta data informations are
4. getDatabaseProductNAme()…returns the product name of database.
5. getUserNAme() returns the usernamr()
6. getURL() returns the URL of the databse.
7. getSchemas() returns all the schema name
8. getPrimaryKey() returns primary key
9. getTables() returns names of tables in the database
   ResultSet Metadata
   ResultSetMetaData rm=Result.getMeatData()

   The method used to retrieve meta data information about result set are
10. getColumnCount() returns the number of columns contained in result set

Data types of Sql used in setXXX() and getXXX() methods.

SQL	JDBC/Java
VARCHAR	java.lang.String
CHAR	java.lang.String
LONGVARCHAR	java.lang.String
BIT	boolean
NUMERIC	java.math.BigDecimal
TINYINT	byte

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	float
DOUBLE	double
VARBINARY	byte[ ]
BINARY	byte[ ]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	java.sql.Clob
BLOB	java.sql.Blob
ARRAY	java.sql.Array
REF	java.sql.Ref

Exceptions handling with jdbc

Exception handling allows you to handle exceptional conditions such as program-defined errors in a controlled fashion.

1. When an exception condition occurs, an exception is thrown. The term thrown means that current program execution stops, and control is redirected to the nearest applicable catch clause. If no applicable catch clause exists, then the program's execution ends.

2. JDBC Exception handling is very similar to Java Excpetion handling but for JDBC.

3. There are three kind of exception thrown by jdbc methods.

4. SQLException ,SQLWarnings, DataTruncation

   SQLException

5. The most common exception you'll deal with is **java.sql.SQLException which result in SQL sysntax errors.**

6. getNextException() method returns details about the error.

7. getErrorCode() method retrieves vendor specific error codes.

   SQLWarnings

8. it throws warnings related to connection from DBMS. getWarnings() method of connecction object retrieves t **warnings.** getNextWarnings() **returns** subsequent warnings.

   DataTruncation

9. Whenever data is lost due to truncation of the data value , a truncation exception is thrown.

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

Differentiate between a Statement and a PreparedStatement.

• A standard Statement is used for creating a Java representation for a literal SQL statement and for executing it on the database.

• A PreparedStatement is a precompiled Statement.

• A Statement has to verify its metadata in the database every time.

• But ,the prepared statement has to verify its metadata in the database only once.

• If we execute the SQL statement, it will go to the STATEMENT.

• But, if we want to execute a single SQL statement for the multiple number of times, it'll go to the PreparedStatement.

Explain the classes, interface , methods available in java.sql.* package.

Java.sql.package include classes and interface to perform almost all JDBC operation such as creating and executing SQL queries

1. java.sql.BLOB --------provide support to BLOB SQL data type.
2. java.sql.Connection----- creates connection with specific data type
   Methods in Connection
   setSavePoint()
   rollback()
   commit()
   setAutoCommit()
3. java.sql.CallableStatement-------- Executes stored procedures
   Methods in CallableStatement
   execute()
   registerOutParameter()
4. java.sql.CLOB ------------ support for CLOB data type.
5. java.sql.Date------------ support for Date SQL type.
6. Java.sql.Driver -----create instance of driver with the DriverManager
7. java.sql.DriverManager---- manages the data base driver
   getConnection()
   setLoginTimeout()
   getLoginTimeout()
8. java.sql.PreparedStatement—create parameterized query
   executeQuery()
   executeUpdate()
9. java.sql.ResultSet------------ it is interface to access result row by row
   rs.next()
   rs.last()
   rs.first()
10. java.sql.Savepoint------- Specify savepoint in transaction.
11. java.sql.SQLException--------- Encapsulates JDBC related exception.
12. java.sql.Statement……….. interface used to execute SQL statement.

13. java.sql.DataBaseMetaData….. returns mata data

University Questions

1. Write a program to display current content of table in database.
2. Exceptions handling with jdbc program.
3. Write notes on metadata interface Metadata
4. Explain the with an example updatable Result Set.
5. Explain the Transaction processing with example
6. Explain the with an example updatable Result Set.
7. Explain the with an example Scrollable Result Set (6 Marks)
8. Explain the various steps of the JDBC process with code snippets.
9. Briefly explain the callable statement object. Write program to call a stored procedure
10. (Briefly discuss the various JDBC driver types 10 M)
11. With proper syntax, explain three types of getConnection() method.
12. Explain J2ee multitier architecture.
13. Explain the classes, interface available in java.sql.* package.

ADVANCED JAVA and J2EE

Subject Code: 15CS553 **IA Marks: 20**
Hours/Week: 3 **Exam Hours: 3**
Total Hours: 40 **Exam Marks: 80**

MODULE-I	**8 Hours**
Enumerations, Autoboxing and Annotations(metadata): Enumerations, Enumeration fundamentals, the values() and valueOf() Methods, java enumerations are class types, enumerations Inherits Enum, example, type wrappers, Autoboxing, Autoboxing and Methods, Autoboxing/Unboxing occurs in Expressions, Autoboxing/Unboxing, Boolean and character values, Autoboxing/Unboxing helps prevent errors, A word of Warning. Annotations, Annotation basics, specifying retention policy, Obtaining Annotations at runtime by use of reflection, Annotated element Interface, Using Default values, Marker Annotations, Single Member annotations, Built-In annotations.	
MODULE-II	**8 Hours**
The collections and Framework: Collections Overview, Recent Changes to Collections, The Collection Interfaces, The Collection Classes, Accessing a collection Via an Iterator, Storing User Defined Classes in Collections, The Random Access Interface, Working With Maps, Comparators, The Collection Algorithms, Why Generic Collections?, The legacy Classes and Interfaces, Parting Thoughts on Collections.	
MODULE-III	**8 Hours**
String Handling : The String Constructors, String Length, Special String Operations, String Literals, String Concatenation, String Concatenation with Other Data Types, String Conversion and toString( ) Character Extraction, charAt( ), getChars( ), getBytes( ) toCharArray(), String Comparison, equals( ) and equalsIgnoreCase( ), regionMatches( ) startsWith( ) and endsWith( ), equals() Versus == , compareTo( ) Searching Strings, Modifying a String, substring( ), concat( ), replace( ), trim( ), Data Conversion Using valueOf( ), Changing the Case of Characters Within a String, Additional String Methods, StringBuffer , StringBuffer Constructors, length( ) and capacity( ), ensureCapacity( ), setLength( ), charAt( ) and setCharAt( ), getChars( ),append( ), insert( ), reverse(), delete( ) and deleteCharAt( ), replace( ), substring( ), Additional StringBuffer Methods, StringBuilder	
MODULE-IV	**8 Hours**
Servlet: Background; The Life Cycle of a Servlet; Using Tomcat for Servlet Development; A simple Servlet; The Servlet API; The Javax.servlet Package; Reading Servlet Parameter; The Javax.servlet.http package; Handling HTTP Requests and Responses; Using Cookies; Session Tracking. **Java Server Pages (JSP):** JSP, JSP Tags, Tomcat, Request String, User Sessions, Cookies, Session Objects	
MODULE-V	**8 Hours**

The Concept of JDBC; JDBC Driver Types; JDBC Packages; A Brief Overview of the JDBC process; Database Connection; Associating the JDBC/ODBC Bridge with the Database; Statement Objects; ResultSet; Transaction Processing; Metadata, Data types; Exceptions.

Reference / Text Book Details

Sl.No.	Title of Book	Author	Publication	Edition
1	**JAVA The Complete Reference**	Herbert Schildt	Tata McGraw Hill	7th/9th
2	**J2EE The Complete Reference**	Jim Keogh	Tata McGraw Hill	2007
3	**Introduction to JAVA Programming**	Y. Daniel Liang	Pearson Education, 2007	7th
4	**The J2EE Tutorial**	Stephanie Bodoff	Pearson Education, 2004	2nd
5	**Advanced JAVA Programming**	Uttam K Roy	Oxford University Press	2015

.

Table of Contents

SL No	Module Description	Page No
1	**Module 1 – Enumeration, AutoBoxing and Annotations**	**1-23**
2	**Module 2 – Collection Framework**	**24-57**
3	**Module 3 – String Methods**	**58-119**
4	**Module 4 – Servlets and JSP**	**120-149**
5	**Module 5 – JDBC Methods**	**150-166**

Module – 1
Enumerations, Autoboxing and Annotations

Enumerations

Enumerations was added to Java language in JDK5. **Enumeration** means a list of named constant. In Java, enumeration defines a class type. An Enumeration can have constructors, methods and instance variables. It is created using **enum** keyword. Each enumeration constant is *public*, *static* and *final* by default. Even though enumeration defines a class type and have constructors, you do not instantiate an **enum** using **new**. Enumeration variables are used and declared in much a same way as you do a primitive variable.

How to Define and Use an Enumeration

1. An enumeration can be defined simply by creating a list of enum variable. Let us take an example for list of Subject variable, with different subjects in the list.

```
enum Subject          //Enumeration defined
{
  Java, Cpp, C, Dbms
}
```

2. Identifiers Java, Cpp, C and Dbms are called **enumeration constants**. These are public, static and final by default.
3. Variables of Enumeration can be defined directly without any **new** keyword.

```
  Subject sub;
```

4. Variables of Enumeration type can have only enumeration constants as value. We define an enum variable as enum_variable = enum_type.enum_constant;

```
  sub = Subject.Java;
```

5. Two enumeration constants can be compared for equality by using the = = relational operator.

Example:

```
if(sub == Subject.Java) {
  ...
}
```

Example of Enumeration

```
enum WeekDays
{ sun, mon, tues, wed, thurs, fri, sat }

class Test
{
 public static void main(String args[])
 {
  WeekDays wk; //wk is an enumeration variable of type WeekDays
  wk = WeekDays.sun; //wk can be assigned only the constants defined under
                     //enum type Weekdays
  System.out.println("Today is "+wk);
 }
}
```

Output :

Today is sun

The enum can be defined within or outside the class because it is similar to a class.

```
enum Season { WINTER, SPRING, SUMMER, FALL }
class EnumExample2{
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s);
}}
```

Output:WINTER

```
class EnumExample3{
enum Season { WINTER, SPRING, SUMMER, FALL;}//semicolon; is optional here
public static void main(String[] args) {
Season s=Season.WINTER;//enum type is required to access WINTER
System.out.println(s);
}}
```

Output:WINTER

Example of Enumeration using switch statement

```
class EnumExample5{
enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY}
public static void main(String args[]){
Day day=Day.MONDAY;
```

```
switch(day){
case SUNDAY:
 System.out.println("sunday");
 break;
case MONDAY:
 System.out.println("monday");
 break;
default:
System.out.println("other day");
}
}}
```

Output:monday

Values( ) and ValueOf( ) method

All the enumerations has predefined methods **values()** and **valueOf()**. values() method returns an array of enum-type containing all the enumeration constants in it. Its general form is,

public **static** *enum-type[ ]* **values()**

valueOf() method is used to return the enumeration constant whose value is equal to the string passed in as argument while calling this method. It's general form is,

public **static** *enum-type* **valueOf** (String *str*)

Example of enumeration using values() and valueOf() methods:

```
enum Restaurants {
dominos, kfc, pizzahut, paninos, burgerking
}
class Test {
public static void main(String args[])
{
Restaurants r;
System.out.println("All constants of enum type Restaurants are:");
Restaurants  rArray[] = Restaurants.values(); //returns an array of constants of type Restaurants
for(Restaurants  a : rArray) //using foreach loop
System.out.println(a);

r = Restaurants.valueOf("dominos");
System.out.println("I AM " + r);
}
}
```

Output:

All constants of enum type Restaurants are:
dominos

kfc
pizzahut
paninos
burgerking
I AM dominos

Instead of creating array we can directly obtain all values.

```
class EnumExample1{
                public enum Season { WINTER, SPRING, SUMMER, FALL }
                public static void main(String[] args) {
                        for (Season s : Season.values())
                        System.out.println(s);
                        }
                }
```

Output:WINTER
SPRING
SUMMER
FALL

Points to remember about Enumerations

1. Enumerations are of class type, and have all the capabilities that a Java class has.
2. Enumerations can have Constructors, instance Variables, methods and can even implement Interfaces.
3. Enumerations are not instantiated using **new** keyword.
4. All Enumerations by default inherit **java.lang.Enum** class.
5. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.
6. enum may implement many interfaces but cannot extend any class because it internally extends Enum class

---

Java Enum are class type and can contain Constructor, instance variable and Method & implement interface.

```
enum Student
{
 John(11), Jonny(10), Sam(13), Viraaj(9);
 private int age;                 //variable defined in enum Student
 int getage() { return age; }  //method defined in enum Student
 Student(int age)  //constructor defined in enum Student
 {
  this.age= age;
 }
}
```

```
class EnumDemo
{
 public static void main( String args[] )
 {
  Student S; //is enum variable or enum object. Constructor is called when
            //each enum object is created.

  System.out.println("Age of Viraaj is " +Student.Viraaj.getage()+ "years");

//display all student and ages
System.out.println("All students age is:");
for (Student a : Student.values())
System.out.println(a +" age is " +a.getage());

 }

}
```

Output :

Age of Viraaj is 9 years

All students age is:
John age is 11
Jonny age is 10
Sam age is 13
Viraaj age is 9

In this example as soon as we declare an enum variable(*Student S*), the constructor is called, and it initializes age for every enumeration constant with values specified with them in parenthesis.

Each enum constant(John, Jonny…) has its own copy of value(age…)

Student.Viraaj.getage()returns age of Viraaj.

Java Enum containg overloaded constructor

```
class EnumExamp{
enum Season{
WINTER(5), SPRING(10), SUMMER(15), FALL;

private int value;
 Season(int v){
  value=v;
}
//default constructor initializes value to -1
Season(){
  value= -1;
 }
```

```
}
public static void main(String args[]){
        //printing enum constant and its value
        System.out.println("Season is "+ Season.SUMMER+ " value is "+ Season.SUMMER.value);
//printing all enum constant and its value
for (Season s : Season.values())
System.out.println(s+" "+s.value);

}}
```

output

Season is SUMMER value is 15
WINTER 5
SPRING 10
SUMMER 15
FALL -1

Enumerations Inherits Enum

All enumerations automatically inherit **java.lang.Enum**. The **Enum** class defines several methods such as **ordinal( )**, **compareTo( )**, **equals( )** and so on, that are available for use by all enumerations. You can obtain a value that indicates an enumeration constant's position in the list of constants. This is called its **'ordinal value'** and it is retrieved by calling the **ordinal( )** method. This method returns the ordinal value of the invoking constant. Ordinal values begin at **'0'**.

final int ordinal()

//example using **ordinal() method**

```
enum Season { WINTER, SPRING, SUMMER, FALL }
class EnumExample2{
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s.ordinal());
}}
```

Output:

0

//example using **compareTo( ) method**

You can compare the ordinal value of two contents of the same enumeration by using the **compareTo( )** method. This method returns a negative integer, zero, or a positive integer as this object ordinal value less than, equal to, or greater than the specified object ordinal value.

```
enum Tutorials {
  topic1, topic2, topic3;
}

public class EnumDemo {
```

```
  public static void main(String args[]) {

    Tutorials t1, t2, t3;

    t1 = Tutorials.topic1;
    t2 = Tutorials.topic2;
    t3 = Tutorials.topic3;

    if(t1.compareTo(t2) > 0) {
      System.out.println(t2 + " completed before " + t1);
    }

    if(t1.compareTo(t2) < 0) {
      System.out.println(t1 + " completed before " + t2);
    }

    if(t1.compareTo(t3) == 0) {
      System.out.println(t1 + " completed with " + t3);
    }
  }
}
topic1 completed before topic2
```

You can compare an enumeration constant with any other object by using **equal( )**, which overrides the **equals( )** method defined by **Object**. Although **equals( )** can compare an enumeration constant to **any other object**, those two objects will only be equal if they both refer to the same constant, within the same enumeration.

```
enum Tutorials {
  topic1, topic2, topic3;
}

public class EnumDemo {

  public static void main(String args[]) {

    Tutorials t1, t2, t3;

    t1 = Tutorials.topic1;
    t2 = Tutorials.topic2;
    t3 = Tutorials.topic1;

    if(t1.equals(t2)) {
      System.out.println("Error");
    }

    if (t1.equals(t3)) {
```

```
        System.out.println(t1 + " Equals " + t3);
      }
   }
}
```

topic1 Equals topic1

We can compare 2 enumerations references for equality using ==(operator).

```
enum Apple {
  shimla, ooty, wood, green, red
}

public class EnumDemo4567 {
  public static void main(String args[])
  {
    Apple ap, ap2, ap3;

    // Obtain all ordinal values using ordinal().
    System.out.println("Here are all apple constants" +
                    " and their ordinal values: ");
    for(Apple a : Apple.values())
      System.out.println(a + " " + a.ordinal());

    ap =  Apple.wood;
    ap2 = Apple.ooty;
    ap3 = Apple.wood;

    System.out.println();

    // Demonstrate compareTo() and equals()
    if(ap.compareTo(ap2) < 0)
      System.out.println(ap + " comes before " + ap2);

    if(ap.compareTo(ap2) > 0)
      System.out.println(ap2 + " comes before " + ap);

    if(ap.compareTo(ap3) == 0)
      System.out.println(ap + " equals " + ap3);

    System.out.println();

    if(ap.equals(ap2))
      System.out.println("Error!");

    if(ap.equals(ap3))
      System.out.println(ap + " equals " + ap3);

    if(ap == ap3)
```

```
      System.out.println(ap + " == " + ap3);

  }
}
```

Here are all apple constants and their ordinal values:
shimla 0
ooty 1
wood 2
green 3
red 4

ooty comes before wood
wood equals wood

wood equals wood
wood == wood

Type Wrapper

Java uses primitive data types such as int, double, float etc. to hold the basic data types.
Eg. Int a =10;
Float f=24.7;
Char ch='c';
Despite the performance benefits offered by the primitive data types, there are situations when you will need an object representation of the primitive data type. For example, many data structures in Java operate on objects. So you cannot use primitive data types with those data structures. To handle such type of situations, Java provides **type Wrappers** which provide classes that encapsulate a primitive type within an object.

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as **ArrayList** and **Vector**, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Character : It encapsulates primitive type char within object.
Character (char ch) //constructor for Character class
ch-specifies character that will be wrapped by Character object being created.
To obtain primitive char value contained in **Character** object call
char charValue()

Boolean : It encapsulates primitive type boolean within object.
Boolean (boolean b) //constructor for Boolean class
To obtain primitive bool value contained in **Boolean** object call
boolean booleanValue()

Likewise for below wrapper classes
Numeric type wrappers : It is the most commonly used type wrapper.
Byte Short Integer Long Float Double

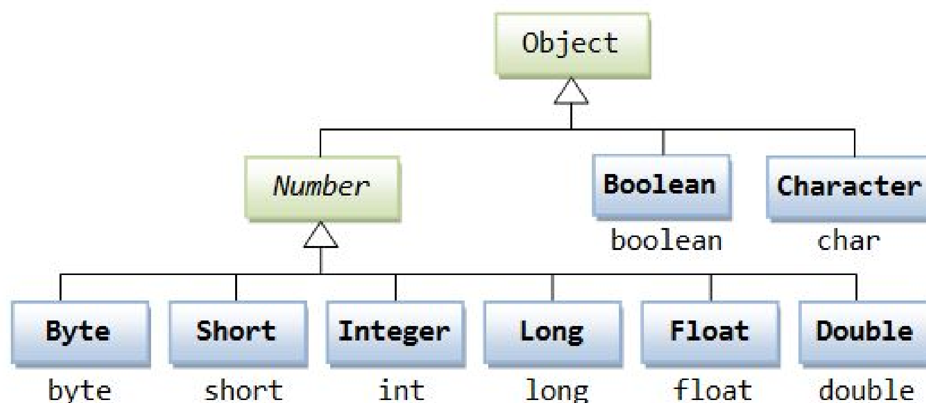
Primitive	Wrapper Class	Constructor Argument	Methods to get primitive values
boolean	Boolean	Boolean (boolean b) or String	booleanValue()
byte	Byte	Byte (byte b) or String	byte Value()
char	Character	Character (char ch)	char Value()
int	Integer	Integer(int a) or String	int Value()
float	Float	Float(float f) or double or String	float Value()
double	Double	Double (double d) or String	double Value()
long	Long	Long (long l)or String	long Value()
short	Short	Short (short) or String	short Value()

Following example shows constructors in wrapper classes.

```
public class WrapperClasses
{
   public static void main(String[] args)
   {
      Byte B1 = new Byte((byte) 10);     //Constructor which takes byte value as an argument
      Byte B2 = new Byte("10");          //Constructor which takes String as an argument

      //Byte B3 = new Byte("abc");       //Run Time Error : NumberFormatException

      //Because, String abc can not be parse-able to byte

      Short S1 = new Short((short) 20);   //Constructor which takes short value as an argument
      Short S2 = new Short("10");         //Constructor which takes String as an argument

      Integer I1 = new Integer(30);     //Constructor which takes int value as an argument
      Integer I2 = new Integer("30");   //Constructor which takes String as an argument

      Long L1 = new Long(40);      //Constructor which takes long value as an argument
      Long L2 = new Long("40");    //Constructor which takes String as an argument

      Float F1 = new Float(12.2f);      //Constructor which takes float value as an argument
      Float F2 = new Float("15.6");   //Constructor which takes String as an argument
```

```
        Float F3 = new Float(15.6d);    //Constructor which takes double value as an argument

        Double D1 = new Double(17.8d);       //Constructor which takes double value as an argument
        Double D2 = new Double("17.8");     //Constructor which takes String as an argument

        Boolean BLN1 = new Boolean(false);      //Constructor which takes boolean value as an argument
        Boolean BLN2 = new Boolean("true");      //Constructor which takes String as an argument

        Character C1 = new Character('D');      //Constructor which takes char value as an argument
        Character C2 = new Character("abc");    //Compile time error : String abc can not be converted to
character
    }
}
```

Type Wrapper Hierarchy



Boxing : Process of converting primitive type to corresponding wrapper.

Eg. Integer i = new Integer(10);
Integer j = 20;

UnBoxing : Process of extracting value for type wrapper.

int a = i.intValue(i);

Autoboxing and Unboxing

- Autoboxing and Unboxing features was added in Java5.
- **Autoboxing** is a process by which primitive type is automatically encapsulated(boxed) into its equivalent type wrapper
- **Auto-Unboxing** is a process by which the value of an object is automatically extracted from a type Wrapper class.

Benefits of Autoboxing / Unboxing

1. Autoboxing / Unboxing lets us use primitive types and Wrapper class objects interchangeably.

2. We don't have to perform Explicit **typecasting**.
3. It helps prevent errors, but may lead to unexpected results sometimes. Hence must be used with care.
4. Auto-unboxing also allows you to mix different types of numeric objects in an expression. When the values are unboxed, the standard type conversions can be applied.

Simple Example of Autoboxing in java:

```
class BoxingExample1{
  public static void main(String args[]){
    int a=50;
        Integer a2=new Integer(a);//Boxing

        Integer a3=5;//Boxing

        System.out.println(a2+" "+a3);
 }
}
Output:50 5
```

Simple Example of Unboxing in java:

```
class UnboxingExample1{
  public static void main(String args[]){
    Integer i=new Integer(50);
        int a=i;

        System.out.println(a);
 }
}
Output:50
```

Autoboxing / Unboxing in Expressions

Whenever we use object of Wrapper class in an expression, automatic unboxing and boxing is done by JVM.

```
Integer iOb;
iOb = 100;        //Autoboxing of int
++iOb;
```

When we perform increment operation on Integer object, it is first unboxed, then incremented and then again reboxed into Integer type object.

This will happen always, when we will use Wrapper class objects in expressions or conditions etc.

Example 2

```
class Test {
public static void main(String args[]) {
Integer i = 35;
Double d = 33.3;
d = d + i;
System.out.println("Value of d is " + d);
}
}
```

Ouput:
Value of d is 68.3

Note: When the statement **d = d + i;** was executed, i was auto-unboxed into int, d was auto-unboxed into double, addition was performed and then finally, auto-boxing of d was done into Double type Wrapper class.

Autoboxing / Unboxing in Methods

```
class Boxing1{
  static void m(int i)
          {System.out.println("int");}
  public static void main(String args[ ]){
   Integer s=30;
   m(s);
  }
}
```

Output:int

Autoboxing / Unboxing Boolean

```
class UnboxingExample2{
  public static void main(String args[ ]){
    Integer i=new Integer(50);
      if(i<100){           //unboxing internally
        System.out.println(i);
        }
 }
}
```

Output:50

Autoboxing / Unboxing Boolean and character values

```
// Autoboxing/unboxing a Boolean and Character.
class AutoBox5 {
```

```
public static void main(String args[]) {
// Autobox/unbox a boolean.
Boolean b = true;
// Below, b is auto-unboxed when used in
// a conditional expression, such as an if.
if(b) System.out.println("b is true");
// Autobox/unbox a char.
Character ch = 'x'; // box a char
char ch2 = ch; // unbox a char
System.out.println("ch2 is " + ch2);
}
}
```

The output is shown here:

```
b is true
ch2 is x
```

Autoboxing / Unboxing helps preventing errors

```
// An error produced by manual unboxing.
class UnboxingError {
public static void main(String args[]) {
Integer iOb = 1000; // autobox the value 1000
int i = iOb.byteValue(); // manually unbox as byte !!!
System.out.println(i); // does not display 1000 !
}
}
```

Annotations (Metadata)

Java Annotations allow us to add metadata information into our source code,
Annotations were added to the java from JDK 5.
Annotations, does not change the actions of a program.
Thus, an annotation leaves the semantics of a program unchanged.
However, this information can be used by various tools during both development and deployment.

- Annotations start with ‘@’.
- Annotations do not change action of a compiled program.
- Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- Annotations are not pure comments as they can change the way a program is treated by compiler.

Annotations basics

An annotation always starts with the symbol @ followed by the annotation name. The symbol @ indicates to the compiler that this is an annotation.

Where we can use annotations?

Annotations can be applied to the classes, interfaces, methods and fields.

Built-In Java Annotations

There are **7** built-in annotations in java. Some annotations are applied to java code and some to other annotations.

Built-In Java Annotations used in java code imported from java.lang

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations

4 Annotations imported from java.lang.annotation

- @Target
- @Retention
- @Inherited
- @Documented

1. **@Override** It is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result . It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

```
class Base
{
    public void Display()
    {
       System.out.println("Base display()");
    }
     public static void main(String args[])
    {
       Base t1 = new Derived();
       t1.Display();
    }
}
class Derived extends Base
{
    @Override
    public void Display()
    {
       System.out.println("Derived display()");
    }
}
```

Output:
Derived display()

2. **@SuppressWarnings**
   It is used to inform the compiler to suppress specified compiler warnings. The warnings to suppress are specified by name, in string form. This type of annotation can be applied to any type of declaration.

Java groups warnings under two categories. They are : **deprecation** and **unchecked.**. Any unchecked warning is generated when a legacy code interfaces with a code that use generics.

```
class DeprecatedTest
{
   @Deprecated
   public void Display()
   {
      System.out.println("Deprecatedtest display()");
   }
}

public class SuppressWarningTest
{
   // If we comment below annotation, program generates
   // warning
   @SuppressWarnings({"checked", "deprecation"})
   public static void main(String args[])
   {
      DeprecatedTest d1 = new DeprecatedTest();
      d1.Display();
   }
}
```

Output:
Deprecatedtest display()

3. **@Deprecated** It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.The Javadoc @deprecated tag should be used when an element has been deprecated.

```
public class DeprecatedTest
{
   @Deprecated
   public void Display()
   {
      System.out.println("Deprecatedtest display()");
   }

   public static void main(String args[])
   {
      DeprecatedTest d1 = new DeprecatedTest();
```

```
        d1.Display();
    }
}
```

Output:
Deprecatedtest display()

Types of Annotation

There are 3 categories of Annotations:-
1. Marker Annotations:
The only purpose is to mark a declaration. These annotations contain no members and do not consist any data. Thus, its presence as an annotation is sufficient. Since, marker interface contains no members, simply determining whether it is present or absent is sufficient. **@Override** , @Deprecated is an example of Marker Annotation.

Example: - @TestAnnotation()

2. Single value Annotations:
These annotations contain only one member and allow a shorthand form of specifying the value of the member. We only need to specify the value for that member when the annotation is applied and don't need to specify the name of the member. However in order to use this shorthand, the name of the member must be **value.**

Example: - @TestAnnotation("testing");

3. Multivalue Annotations:
These annotations consist of multiple data members/ name, value, pairs.

Example:- @TestAnnotation(owner="Umesh", value="Tutor")

Java Custom Annotation

Java Custom annotations or Java User-defined annotations are easy to create and use. The *@interface* element is used to declare an annotation. For example:

```
@interface MyAnnotation{ }
```

Here, MyAnnotation is the custom annotation name.

Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

1. Annotations are created by using @interface, followed by annotation name as shown in the below example.
2. An annotation can have elements as well. They look like methods. For example in the below code, we have four elements. We should not provide implementation for these elements.

3. Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
4. Method should not have any parameter.
5. We should attach @ just before interface keyword to define annotation.
6. It may assign a default value to the method.

Declaring custom Annotation

```
@interface MyAnnotation{
   int value1( ) default 1;
   String value2( ) default "";
   String value3( ) default "xyz";
   }
```

How to apply custom Annotation

```
@MyAnnotation(value1=10,value2="Umesh",value3="SJBIT")
```

4. **@Documented** It is a marker interface that tells a tool that an annotation is to be documented. Annotations are not included by Javadoc comments. Use of @Documented annotation in the code enables tools like Javadoc to process it and include the annotation type information in the generated document.

```
java.lang.annotation.Documented
@Documented
public @interface MyCustomAnnotation {
  //Annotation body
}

@MyCustomAnnotation
public class MyClass {
     //Class body
}
```

While generating the javadoc for class MyClass, the annotation @MyCustomAnnotation would be included in that.

5. **@Inherited**

The @Inherited annotation signals that a custom annotation used in a class should be inherited by all of its sub classes. For example:

```
java.lang.annotation.Inherited
```

```
@Inherited
public @interface MyCustomAnnotation {

}
@MyCustomAnnotation
public class MyParentClass {
  ...
}
public class MyChildClass extends MyParentClass {
   ...
}
```

Here the class MyParentClass is using annotation @MyCustomAnnotation which is marked with @inherited annotation. It means the sub class MyChildClass inherits the @MyCustomAnnotation.

6. @Target

It specifies where we can use the annotation. For example: In the below code, we have defined the target type as METHOD which means the below annotation can only be used on methods.

The java.lang.annotation.**ElementType** enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc.

Element Types	Where the annotation can be applied
TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target({ElementType.METHOD})
// u can also target multiple elements
//@Target({ ElementType.FIELD, ElementType.METHOD})
public @interface MyCustomAnnotation {

}
public class MyClass {
   @MyCustomAnnotation
```

```
   public void myMethod()
   {
      //Doing something
   }
}
```

7. **@Retention**

It indicates how long annotations with the annotated type are to be retained.

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
@interface MyCustomAnnotation {

}
```

Here we have used RetentionPolicy.RUNTIME. There are two other options as well. Lets see what do they mean:

RetentionPolicy.RUNTIME: The annotation should be available at runtime, for inspection via java reflection.

RetentionPolicy.CLASS: The annotation would be in the .class file but it would not be available at runtime.

RetentionPolicy.SOURCE: The annotation would be available in the source code of the program, it would neither be in the .class file nor be available at the runtime.

Complete in one example

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Target(ElementType.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation{
    int studentAge() default 18;
    String studentName();
    String stuAddress();
    String stuStream() default "CSE";
}
@MyCustomAnnotation( studentName="umesh", stuAddress="India")
public class MyClass {
...
```

```
}
```

Obtaining annotation at runtime using reflection

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

- The required classes for reflection are provided under java.lang.reflect package.

Reflection can be used to get information about –

- **Class** The getClass() method is used to get the name of the class to which an object belongs.
- **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
- **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.

```
import java.lang.annotation.*;
import java.lang.reflect.*;
// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
String str();
int val();
}
class Meta {
// Annotate a method.
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth() {
Meta ob = new Meta();
// Obtain the annotation for this method
// and display the values of the members.

try {

// First, get a Class object that represents

// this class.
Class c = ob.getClass();
// Now, get a Method object that represents
// this method.
Method m = c.getMethod("myMeth");
// Next, get the annotation for this class.
MyAnno anno = m.getAnnotation(MyAnno.class);
// Finally, display the values.
System.out.println(anno.str() + " " + anno.val());
} catch (NoSuchMethodException exc) {
```

```
System.out.println("Method Not Found.");
}
}
public static void main(String args[]) {
myMeth();
}
}
```

The output from the program is shown here:

Annotation Example 100

Obtaining All Annotations

You can obtain all annotations that have **RUNTIME** retention that are associated with an item by calling **getAnnotations( )** on that item. It has this general form:

Annotation[ ] getAnnotations( )

It returns an array of the annotations. **getAnnotations( )** can be called on objects of type **Class**, **Method**, **Constructor**, and **Field**.

```
// Show all annotations for a class and a method.
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
	String str();
	int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
	String description();
}
@What(description = "An annotation test class")

@MyAnno(str = "Meta2", val = 99)

class Meta2 {
@What(description = "An annotation test method")
@MyAnno(str = "Testing", val = 100)
	public static void myMeth() {
		Meta2 ob = new Meta2();
		try {
			Annotation annos[] = ob.getClass().getAnnotations();
			// Display all annotations for Meta2.
			System.out.println("All annotations for Meta2:");
```

```
                    for(Annotation a : annos)
                    System.out.println(a);
                    System.out.println();
                    // Display all annotations for myMeth.
                    Method m = ob.getClass( ).getMethod("myMeth");
                    annos = m.getAnnotations();
                    System.out.println("All annotations for myMeth:");
                    for(Annotation a : annos)
                        System.out.println(a);
                    } catch (NoSuchMethodException exc) {
                        System.out.println("Method Not Found.");
                    }
}
public static void main(String args[]) {
    myMeth();
    }
}
```

The output is shown here:

```
All annotations for Meta2:
@What(description=An annotation test class)
@MyAnno(str=Meta2, val=99)
All annotations for myMeth:
@What(description=An annotation test method)
@MyAnno(str=Testing, val=100)
```

The AnnotatedElement Interface

The methods **getAnnotation( )** and **getAnnotations( )** used by the preceding examples are defined by the **AnnotatedElement** interface, which is defined in **java.lang.reflect**. This interface supports reflection for annotations and is implemented by the classes Method, Field,Constructor, Class, and Package.

In addition to getAnnotation( ) and getAnnotations( ), **AnnotatedElemen**t defines two other methods. The first **is getDeclaredAnnotations( ),** which has this general form:

Annotation[ ] getDeclaredAnnotations( )

It returns all non-inherited annotations present in the invoking object. The second is **isAnnotationPresent( ),** which has this general form:

boolean isAnnotationPresent(Class annoType)

It returns true if the annotation specified by annoType is associated with the invokingobject. It returns false otherwise.

MODULE – 2

COLLECTION FRAMEWORK

Introduction to Collections

A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers). If you have used the Java programming language — or just about any other programming language — you are already familiar with collections.

What Is a Collections Framework?

A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve. We believe that the Java Collections Framework breaks with this tradition, as you will learn for yourself in this chapter.

Benefits of the Java Collections Framework

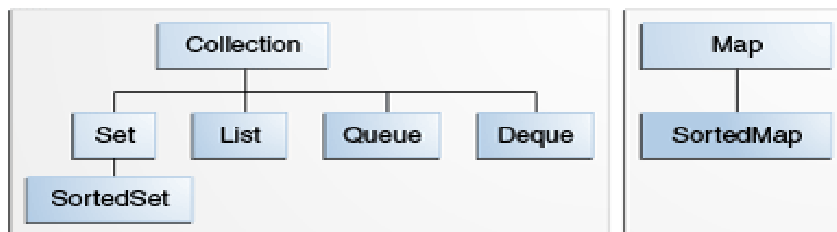
The Java Collections Framework provides the following benefits:

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.

- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.
- **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Interfaces

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



The core collection interfaces.

A `Set` is a special kind of `Collection`, a `SortedSet` is a special kind of `Set`, and so forth. Note also that the hierarchy consists of two distinct trees — a `Map` is not a true `Collection`.

Note that all the core collection interfaces are generic. For example, this is the declaration of the `Collection` interface.

```
public interface Collection<E>...
```

The `<E>` syntax tells you that the interface is generic. When you declare a `Collection` instance you can *and should* specify the type of object contained in the collection. Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime. For information on generic types, see the Generics (Updated) lesson.

When you understand how to use these interfaces, you will know most of what there is to know about the Java Collections Framework. This chapter discusses general guidelines for effective use of the interfaces, including when to use which interface. You'll also learn programming idioms for each interface to help you get the most out of it.

To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type. (Such variants might include immutable, fixed-size, and append-only.) Instead, the modification operations in each interface are designated *optional* — a given implementation may elect not to support all operations. If an unsupported operation is invoked, a collection throws an `UnsupportedOperationException`. Implementations are responsible for documenting which of the optional operations they support. All of the Java platform's general-purpose implementations support all of the optional operations.

The following list describes the core collection interfaces:

- `Collection` — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The `Collection` interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as `Set` and `List`. Also see The Collection Interface section.
- `Set` — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine. See also The Set Interface section.
- `List` — an ordered collection (sometimes called a *sequence*). `List`s can contain duplicate elements. The user of a `List` generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used `Vector`, you're familiar with the general flavor of `List`. Also see The List Interface section.
- `Queue` — a collection used to hold multiple elements prior to processing. Besides basic `Collection` operations, a `Queue` provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to `remove` or `poll`. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties. Also see The Queue Interface section.

- `Deque` — a collection used to hold multiple elements prior to processing. Besides basic `Collection` operations, a `Deque` provides additional insertion, extraction, and inspection operations.

  Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends. Also see The Deque Interface section.

- `Map` — an object that maps keys to values. A `Map` cannot contain duplicate keys; each key can map to at most one value. If you've used `Hashtable`, you're already familiar with the basics of `Map`. Also see The Map Interface section.

The last two core collection interfaces are merely sorted versions of `Set` and `Map`:

- `SortedSet` — a `Set` that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls. Also see The SortedSet Interface section.
- `SortedMap` — a `Map` that maintains its mappings in ascending key order. This is the `Map` analog of `SortedSet`. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories. Also see The SortedMap Interface section.

To understand how the sorted interfaces maintain the order of their elements, see the Object Ordering section.

The Collection Interface

A `Collection` represents a group of objects known as its elements. The `Collection` interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a `Collection` argument. This constructor, known as a *conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a Collection<String> c, which may be a List, a Set, or another kind of Collection. This idiom creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c.

```
List<String> list = new ArrayList<String>(c);
```

Or — if you are using JDK 7 or later — you can use the diamond operator:

```
List<String> list = new ArrayList<>(c);
```

The Collection interface contains methods that perform basic operations, such as int size(), boolean isEmpty(), boolean contains(Object element), boolean add(E element), boolean remove(Object element), and Iterator<E> iterator().

It also contains methods that operate on entire collections, such as boolean containsAll(Collection<?> c), boolean addAll(Collection<? extends E> c), boolean removeAll(Collection<?> c), boolean retainAll(Collection<?> c), and void clear().

Additional methods for array operations (such as Object[] toArray() and <T> T[] toArray(T[] a) exist as well.

In JDK 8 and later, the Collection interface also exposes methods Stream<E> stream() and Stream<E> parallelStream(), for obtaining sequential or parallel streams from the underlying collection. (See the lesson entitled Aggregate Operations for more information about using streams.)

The Collection interface does about what you'd expect given that a Collection represents a group of objects. It has methods that tell you how many elements are in the collection (size, isEmpty), methods that check whether a given object is in the collection (contains), methods that add and remove an element from the collection (add, remove), and methods that provide an iterator over the collection (iterator).

The add method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call. Similarly, the remove method is designed to remove a single instance of the specified element from the Collection, assuming that it contains the element to start with, and to return true if the Collection was modified as a result.

Traversing Collections

There are three ways to traverse collections: (1) using aggregate operations (2) with the for-each construct and (3) by using Iterators.

Aggregate Operations

In JDK 8 and later, the preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

```
myShapesCollection.stream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));
```

Likewise, you could easily request a parallel stream, which might make sense if the collection is large enough and your computer has enough cores:

```
myShapesCollection.parallelStream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));
```

There are many different ways to collect data with this API. For example, you might want to convert the elements of a `Collection` to `String` objects, then join them, separated by commas:

```
    String joined = elements.stream()
    .map(Object::toString)
    .collect(Collectors.joining(", "));
```

Or perhaps sum the salaries of all employees:

```
int total = employees.stream()
.collect(Collectors.summingInt(Employee::getSalary)));
```

These are but a few examples of what you can do with streams and aggregate operations. For more information and examples, see the lesson entitled Aggregate Operations.

The Collections framework has always provided a number of so-called "bulk operations" as part of its API. These include methods that operate on entire collections, such as `containsAll`, `addAll`, `removeAll`, etc. Do not confuse those methods with the aggregate operations that were introduced in JDK 8. The key difference between the new aggregate operations and the existing bulk operations (`containsAll`, `addAll`, etc.) is that the old versions are all *mutative*, meaning that they all modify the underlying collection. In contrast, the new aggregate operations *do not* modify the underlying collection. When using the new aggregate operations and lambda expressions, you must take care to avoid mutation so as not to introduce problems in the future, should your code be run later from a parallel stream.

for-each Construct

The `for-each` construct allows you to concisely traverse a collection or array using a `for` loop — see The for Statement. The following code uses the `for-each` construct to print out each element of a collection on a separate line.

```
for (Object o : collection)
    System.out.println(o);
```

Iterators

An `Iterator` is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

The `hasNext` method returns `true` if the iteration has more elements, and the `next` method returns the next element in the iteration. The `remove` method removes the last element that was returned by `next` from the underlying `Collection`. The `remove` method may be called only once per call to `next` and throws an exception if this rule is violated.

Note that `Iterator.remove` is the *only* safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

Use `Iterator` instead of the `for-each` construct when you need to:

- Remove the current element. The `for-each` construct hides the iterator, so you cannot call `remove`. Therefore, the `for-each` construct is not usable for filtering.
- Iterate over multiple collections in parallel.

The following method shows you how to use an `Iterator` to filter an arbitrary `Collection` — that is, traverse the collection removing specific elements.

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

This simple piece of code is polymorphic, which means that it works for *any* `Collection` regardless of implementation. This example demonstrates how easy it is to write a polymorphic algorithm using the Java Collections Framework.

Collection Interface Bulk Operations

Bulk operations perform an operation on an entire `Collection`. You could implement these shorthand operations using the basic operations, though in most cases such implementations would be less efficient. The following are the bulk operations:

- `containsAll` — returns `true` if the target `Collection` contains all of the elements in the specified `Collection`.
- `addAll` — adds all of the elements in the specified `Collection` to the target `Collection`.
- `removeAll` — removes from the target `Collection` all of its elements that are also contained in the specified `Collection`.
- `retainAll` — removes from the target `Collection` all its elements that are *not* also contained in the specified `Collection`. That is, it retains only those elements in the target `Collection` that are also contained in the specified `Collection`.
- `clear` — removes all elements from the `Collection`.

The `addAll`, `removeAll`, and `retainAll` methods all return `true` if the target `Collection` was modified in the process of executing the operation.

As a simple example of the power of bulk operations, consider the following idiom to remove *all* instances of a specified element, `e`, from a `Collection`, `c`.

```
c.removeAll(Collections.singleton(e));
```

More specifically, suppose you want to remove all of the `null` elements from a `Collection`.

```
c.removeAll(Collections.singleton(null));
```

This idiom uses `Collections.singleton`, which is a static factory method that returns an immutable `Set` containing only the specified element.

Collection Interface Array Operations

The `toArray` methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a `Collection` to be translated into an array. The simple form with no arguments creates a new array of `Object`. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

For example, suppose that `c` is a `Collection`. The following snippet dumps the contents of `c` into a newly allocated array of `Object` whose length is identical to the number of elements in `c`.

```
Object[] a = c.toArray();
```

Suppose that `c` is known to contain only strings (perhaps because `c` is of type `Collection<String>`). The following snippet dumps the contents of `c` into a newly allocated array of `String` whose length is identical to the number of elements in `c`.

```
String[] a = c.toArray(new String[0]);
```

The Set Interface

A `Set` is a `Collection` that cannot contain duplicate elements. It models the mathematical set abstraction. The `Set` interface contains *only* methods inherited from `Collection` and adds the restriction that duplicate elements are prohibited. `Set` also adds a stronger contract on the behavior of the `equals` and `hashCode` operations, allowing `Set` instances to be compared meaningfully even if their implementation types differ. Two `Set` instances are equal if they contain the same elements.

The Java platform contains three general-purpose `Set` implementations: `HashSet`, `TreeSet`, and `LinkedHashSet`. `HashSet`, which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. `TreeSet`, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than `HashSet`. `LinkedHashSet`, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). `LinkedHashSet` spares its clients from the unspecified, generally chaotic ordering provided by `HashSet` at a cost that is only slightly higher.

Here's a simple but useful `Set` idiom. Suppose you have a `Collection`, `c`, and you want to create another `Collection` containing the same elements but with all duplicates eliminated. The following one-liner does the trick.

```
Collection<Type> noDups = new HashSet<Type>(c);
```

It works by creating a `Set` (which, by definition, cannot contain duplicates), initially containing all the elements in `c`. It uses the standard conversion constructor described in the The Collection Interface section.

Or, if using JDK 8 or later, you could easily collect into a `Set` using aggregate operations:

```
c.stream()
.collect(Collectors.toSet()); // no duplicates
```

Here's a slightly longer example that accumulates a `Collection` of names into a `TreeSet`:

```
Set<String> set = people.stream()
.map(Person::getName)
.collect(Collectors.toCollection(TreeSet::new));
```

And the following is a minor variant of the first idiom that preserves the order of the original collection while removing duplicate elements:

```
Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

The following is a generic method that encapsulates the preceding idiom, returning a `Set` of the same generic type as the one passed.

```
public static <E> Set<E> removeDups(Collection<E> c) {
    return new LinkedHashSet<E>(c);
}
```

Set Interface Basic Operations

The `size` operation returns the number of elements in the `Set` (its *cardinality*). The `isEmpty` method does exactly what you think it would. The `add` method adds the specified element to the `Set` if it is not already present and returns a boolean indicating whether the element was added. Similarly, the `remove` method removes the specified element from the `Set` if it is present and returns a boolean indicating whether the element was present. The `iterator` method returns an `Iterator` over the `Set`.

The following `program` prints out all distinct words in its argument list. Two versions of this program are provided. The first uses JDK 8 aggregate operations. The second uses the for-each construct.

Using JDK 8 Aggregate Operations:

```
import java.util.*;
import java.util.stream.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> distinctWords = Arrays.asList(args).stream()
                .collect(Collectors.toSet());
        System.out.println(distinctWords.size()+
                           " distinct words: " +
                           distinctWords);
    }
}
```

Using the `for-each` Construct:

```
import java.util.*;

public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
               s.add(a);
               System.out.println(s.size() + " distinct words: " + s);
    }
}
```

Now run either version of the program.

```
java FindDups i came i saw i left
```

The following output is produced:

```
4 distinct words: [left, came, saw, i]
```

Note that the code always refers to the `Collection` by its interface type (`Set`) rather than by its implementation type. This is a *strongly* recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor. If either of the variables used to store a collection or the parameters used to pass it around are declared to be of the `Collection`'s implementation type rather than its interface type, *all* such variables and parameters must be changed in order to change its implementation type.

Furthermore, there's no guarantee that the resulting program will work. If the program uses any nonstandard operations present in the original implementation type but not in the new one, the program will fail. Referring to collections only by their interface prevents you from using any nonstandard operations.

The implementation type of the `Set` in the preceding example is `HashSet`, which makes no guarantees as to the order of the elements in the `Set`. If you want the program to print the word list in alphabetical order, merely change the `Set`'s implementation type from `HashSet` to `TreeSet`. Making this trivial one-line change causes the command line in the previous example to generate the following output.

```
java FindDups i came i saw i left

4 distinct words: [came, i, left, saw]
```

Set Interface Bulk Operations

Bulk operations are particularly well suited to `Set`s; when applied, they perform standard set-algebraic operations. Suppose `s1` and `s2` are sets. Here's what bulk operations do:

- `s1.containsAll(s2)` — returns `true` if `s2` is a **subset** of `s1`. (`s2` is a subset of `s1` if set `s1` contains all of the elements in `s2`.)
- `s1.addAll(s2)` — transforms `s1` into the **union** of `s1` and `s2`. (The union of two sets is the set containing all of the elements contained in either set.)
- `s1.retainAll(s2)` — transforms `s1` into the intersection of `s1` and `s2`. (The intersection of two sets is the set containing only the elements common to both sets.)
- `s1.removeAll(s2)` — transforms `s1` into the (asymmetric) set difference of `s1` and `s2`. (For example, the set difference of `s1` minus `s2` is the set containing all of the elements found in `s1` but not in `s2`.)

To calculate the union, intersection, or set difference of two sets *nondestructively* (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2);

Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2);
```

```
Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2);
```

The implementation type of the result `Set` in the preceding idioms is `HashSet`, which is, as already mentioned, the best all-around `Set` implementation in the Java platform. However, any general-purpose `Set` implementation could be substituted.

Let's revisit the `FindDups` program. Suppose you want to know which words in the argument list occur only once and which occur more than once, but you do not want any duplicates printed out repeatedly. This effect can be achieved by generating two sets — one containing every word in the argument list and the other containing only the duplicates. The words that occur only once are the set difference of these two sets, which we know how to compute. Here's how `the resulting program` looks.

```
import java.util.*;

public class FindDups2 {
    public static void main(String[] args) {
        Set<String> uniques = new HashSet<String>();
        Set<String> dups    = new HashSet<String>();

        for (String a : args)
            if (!uniques.add(a))
                dups.add(a);

        // Destructive set-difference
        uniques.removeAll(dups);

        System.out.println("Unique words:    " + uniques);
        System.out.println("Duplicate words: " + dups);
    }
}
```

When run with the same argument list used earlier (`i came i saw i left`), the program yields the following output.

```
Unique words:    [left, saw, came]
Duplicate words: [i]
```

A less common set-algebraic operation is the *symmetric set difference* — the set of elements contained in either of two specified sets but not in both. The following code calculates the symmetric set difference of two sets nondestructively.

```
Set<Type> symmetricDiff = new HashSet<Type>(s1);
symmetricDiff.addAll(s2);
Set<Type> tmp = new HashSet<Type>(s1);
tmp.retainAll(s2);
symmetricDiff.removeAll(tmp);
```

The List Interface

A `List` is an ordered `Collection` (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from `Collection`, the `List` interface includes operations for the following:

- `Positional access` — manipulates elements based on their numerical position in the list. This includes methods such as `get`, `set`, `add`, `addAll`, and `remove`.
- `Search` — searches for a specified object in the list and returns its numerical position. Search methods include `indexOf` and `lastIndexOf`.
- `Iteration` — extends `Iterator` semantics to take advantage of the list's sequential nature. The `listIterator` methods provide this behavior.
- `Range-view` — The `sublist` method performs arbitrary *range operations* on the list.

The Java platform contains two general-purpose `List` implementations. `ArrayList`, which is usually the better-performing implementation, and `LinkedList` which offers better performance under certain circumstances.

Collection Operations

The operations inherited from `Collection` all do about what you'd expect them to do, assuming you're already familiar with them. If you're not familiar with them from `Collection`, now would be a good time to read The Collection Interface section. The `remove` operation always removes *the first* occurrence of the specified element from the list. The `add` and `addAll` operations always append the new element(s) to the *end* of the list. Thus, the following idiom concatenates one list to another.

```
list1.addAll(list2);
```

Here's a nondestructive form of this idiom, which produces a third `List` consisting of the second list appended to the first.

```
List<Type> list3 = new ArrayList<Type>(list1);
list3.addAll(list2);
```

Note that the idiom, in its nondestructive form, takes advantage of `ArrayList`'s standard conversion constructor.

And here's an example (JDK 8 and later) that aggregates some names into a `List`:

```
List<String> list = people.stream()
.map(Person::getName)
.collect(Collectors.toList());
```

Like the Set interface, List strengthens the requirements on the equals and hashCode methods so that two List objects can be compared for logical equality without regard to their implementation classes. Two List objects are equal if they contain the same elements in the same order.

Positional Access and Search Operations

The basic positional access operations are get, set, add and remove. (The set and remove operations return the old value that is being overwritten or removed.) Other operations (indexOf and lastIndexOf) return the first or last index of the specified element in the list.

The addAll operation inserts all the elements of the specified Collection starting at the specified position. The elements are inserted in the order they are returned by the specified Collection's iterator. This call is the positional access analog of Collection's addAll operation.

Here's a little method to swap two indexed values in a List.

```
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

Of course, there's one big difference. This is a polymorphic algorithm: It swaps two elements in any List, regardless of its implementation type. Here's another polymorphic algorithm that uses the preceding swap method.

```
public static void shuffle(List<?> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i));
}
```

This algorithm, which is included in the Java platform's Collections class, randomly permutes the specified list using the specified source of randomness. It's a bit subtle: It runs up the list from the bottom, repeatedly swapping a randomly selected element into the current position. Unlike most naive attempts at shuffling, it's *fair* (all permutations occur with equal likelihood, assuming an unbiased source of randomness) and *fast* (requiring exactly list.size()-1 swaps). The following program uses this algorithm to print the words in its argument list in random order.

```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        for (String a : args)
            list.add(a);
```

```
        Collections.shuffle(list, new Random());
        System.out.println(list);
    }
}
```

In fact, this program can be made even shorter and faster. The `Arrays` class has a static factory method called `asList`, which allows an array to be viewed as a `List`. This method does not copy the array. Changes in the `List` write through to the array and vice versa. The resulting List is not a general-purpose `List` implementation, because it doesn't implement the (optional) `add` and `remove` operations: Arrays are not resizable. Taking advantage of `Arrays.asList` and calling the library version of `shuffle`, which uses a default source of randomness, you get the following `tiny program` whose behavior is identical to the previous program.

```
import java.util.*;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }
}
```

Iterators

As you'd expect, the `Iterator` returned by `List`'s `iterator` operation returns the elements of the list in proper sequence. `List` also provides a richer iterator, called a `ListIterator`, which allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator.

The three methods that `ListIterator` inherits from `Iterator` (`hasNext`, `next`, and `remove`) do exactly the same thing in both interfaces. The `hasPrevious` and the `previous` operations are exact analogues of `hasNext` and `next`. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The `previous` operation moves the cursor backward, whereas `next` moves it forward.

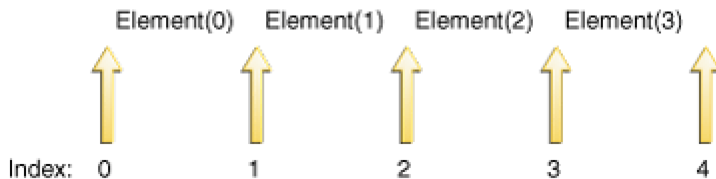
Here's the standard idiom for iterating backward through a list.

```
for      (ListIterator<Type>      it      =      list.listIterator(list.size());
it.hasPrevious(); ) {
    Type t = it.previous();
    ...
}
```

Note the argument to `listIterator` in the preceding idiom. The `List` interface has two forms of the `listIterator` method. The form with no arguments returns a `ListIterator` positioned at the beginning of the list; the form with an `int` argument returns a `ListIterator` positioned at the specified index. The index refers to the element that would be returned by an initial call to

next. An initial call to previous would return the element whose index was index-1. In a list of length n, there are n+1 valid values for index, from 0 to n, inclusive.

Intuitively speaking, the cursor is always between two elements — the one that would be returned by a call to previous and the one that would be returned by a call to next. The n+1 valid index values correspond to the n+1 gaps between elements, from the gap before the first element to the gap after the last one. The following figure shows the five possible cursor positions in a list containing four elements.



The five possible cursor positions.

Calls to next and previous can be intermixed, but you have to be a bit careful. The first call to previous returns the same element as the last call to next. Similarly, the first call to next after a sequence of calls to previous returns the same element as the last call to previous.

It should come as no surprise that the nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous. These calls are typically used either to report the position where something was found or to record the position of the ListIterator so that another ListIterator with identical position can be created.

It should also come as no surprise that the number returned by nextIndex is always one greater than the number returned by previousIndex. This implies the behavior of the two boundary cases: (1) a call to previousIndex when the cursor is before the initial element returns -1 and (2) a call to nextIndex when the cursor is after the final element returns list.size(). To make all this concrete, the following is a possible implementation of List.indexOf.

```
public int indexOf(E e) {
    for (ListIterator<E> it = listIterator(); it.hasNext(); )
        if (e == null ? it.next() == null : e.equals(it.next()))
            return it.previousIndex();
    // Element not found
    return -1;
}
```

Note that the indexOf method returns it.previousIndex() even though it is traversing the list in the forward direction. The reason is that it.nextIndex() would return the index of the element we are about to examine, and we want to return the index of the element we just examined.

The `Iterator` interface provides the `remove` operation to remove the last element returned by `next` from the `Collection`. For `ListIterator`, this operation removes the last element returned by `next` or `previous`. The `ListIterator` interface provides two additional operations to modify the list — `set` and `add`. The `set` method overwrites the last element returned by `next` or `previous` with the specified element. The following polymorphic algorithm uses `set` to replace all occurrences of one specified value with another.

```
public static <E> void replace(List<E> list, E val, E newVal) {
    for (ListIterator<E> it = list.listIterator(); it.hasNext(); )
        if (val == null ? it.next() == null : val.equals(it.next()))
            it.set(newVal);
}
```

The only bit of trickiness in this example is the equality test between `val` and `it.next`. You need to special-case a `val` value of `null` to prevent a `NullPointerException`.

The `add` method inserts a new element into the list immediately before the current cursor position. This method is illustrated in the following polymorphic algorithm to replace all occurrences of a specified value with the sequence of values contained in the specified list.

```
public static <E>
    void replace(List<E> list, E val, List<? extends E> newVals) {
    for (ListIterator<E> it = list.listIterator(); it.hasNext(); ){
        if (val == null ? it.next() == null : val.equals(it.next())) {
            it.remove();
            for (E e : newVals)
                it.add(e);
        }
    }
}
```

Range-View Operation

The `range-view` operation, `subList(int fromIndex, int toIndex)`, returns a `List` view of the portion of this list whose indices range from `fromIndex`, inclusive, to `toIndex`, exclusive. This *half-open range* mirrors the typical `for` loop.

```
for (int i = fromIndex; i < toIndex; i++) {
    ...
}
```

As the term *view* implies, the returned `List` is backed up by the `List` on which `subList` was called, so changes in the former are reflected in the latter.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a `List` can be used as a range operation by passing a `subList` view instead of a whole `List`. For example, the following idiom removes a range of elements from a `List`.

```
list.subList(fromIndex, toIndex).clear();
```

Similar idioms can be constructed to search for an element in a range.

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

Note that the preceding idioms return the index of the found element in the `subList`, not the index in the backing `List`.

Any polymorphic algorithm that operates on a `List`, such as the `replace` and `shuffle` examples, works with the `List` returned by `subList`.

Here's a polymorphic algorithm whose implementation uses `subList` to deal a hand from a deck. That is, it returns a new `List` (the "hand") containing the specified number of elements taken from the end of the specified `List` (the "deck"). The elements returned in the hand are removed from the deck.

```
public static <E> List<E> dealHand(List<E> deck, int n) {
    int deckSize = deck.size();
    List<E> handView = deck.subList(deckSize - n, deckSize);
    List<E> hand = new ArrayList<E>(handView);
    handView.clear();
    return hand;
}
```

Note that this algorithm removes the hand from the *end* of the deck. For many common `List` implementations, such as `ArrayList`, the performance of removing elements from the end of the list is substantially better than that of removing elements from the beginning.

The following is a program that uses the `dealHand` method in combination with `Collections.shuffle` to generate hands from a normal 52-card deck. The program takes two command-line arguments: (1) the number of hands to deal and (2) the number of cards in each hand.

```
import java.util.*;

public class Deal {
    public static void main(String[] args) {
        if (args.length < 2) {
            System.out.println("Usage: Deal hands cards");
            return;
        }
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);

        // Make a normal 52-card deck.
        String[] suit = new String[] {
            "spades", "hearts",
            "diamonds", "clubs"
        };
```

```
        String[] rank = new String[] {
            "ace", "2", "3", "4",
            "5", "6", "7", "8", "9", "10",
            "jack", "queen", "king"
        };

        List<String> deck = new ArrayList<String>();
        for (int i = 0; i < suit.length; i++)
            for (int j = 0; j < rank.length; j++)
                deck.add(rank[j] + " of " + suit[i]);

        // Shuffle the deck.
        Collections.shuffle(deck);

        if (numHands * cardsPerHand > deck.size()) {
            System.out.println("Not enough cards.");
            return;
        }

        for (int i = 0; i < numHands; i++)
            System.out.println(dealHand(deck, cardsPerHand));
    }

    public static <E> List<E> dealHand(List<E> deck, int n) {
        int deckSize = deck.size();
        List<E> handView = deck.subList(deckSize - n, deckSize);
        List<E> hand = new ArrayList<E>(handView);
        handView.clear();
        return hand;
    }
}
```

Running the program produces output like the following.

```
% java Deal 4 5

[8 of hearts, jack of spades, 3 of spades, 4 of spades,
    king of diamonds]
[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts,
    queen of hearts]
[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds,
    9 of clubs]
[8 of spades, 6 of diamonds, ace of spades, 3 of hearts,
    ace of hearts]
```

Although the `subList` operation is extremely powerful, some care must be exercised when using it. The semantics of the `List` returned by `subList` become undefined if elements are added to or removed from the backing `List` in any way other than via the returned `List`. Thus, it's highly recommended that you use the `List` returned by `subList` only as a transient object — to perform one or a sequence of range operations on the backing `List`. The longer you use the `subList` instance, the greater the probability that you'll compromise it by modifying the backing `List` directly or through another `subList` object. Note that it is legal to modify a sublist of a sublist and to continue using the original sublist (though not concurrently).

List Algorithms

Most polymorphic algorithms in the `Collections` class apply specifically to `List`. Having all these algorithms at your disposal makes it very easy to manipulate lists. Here's a summary of these algorithms, which are described in more detail in the Algorithms section.

- `sort` — sorts a `List` using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
- `shuffle` — randomly permutes the elements in a `List`.
- `reverse` — reverses the order of the elements in a `List`.
- `rotate` — rotates all the elements in a `List` by a specified distance.
- `swap` — swaps the elements at specified positions in a `List`.
- `replaceAll` — replaces all occurrences of one specified value with another.
- `fill` — overwrites every element in a `List` with the specified value.
- `copy` — copies the source `List` into the destination `List`.
- `binarySearch` — searches for an element in an ordered `List` using the binary search algorithm.
- `indexOfSubList` — returns the index of the first sublist of one `List` that is equal to another.
- `lastIndexOfSubList` — returns the index of the last sublist of one `List` that is equal to another.

The Queue Interface

- A `Queue` is a collection for holding elements prior to processing. Besides basic `Collection` operations, queues provide additional insertion, removal, and inspection operations. The `Queue` interface follows.

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
    E poll();
    E remove();
}
```

- Each `Queue` method exists in two forms: (1) one throws an exception if the operation fails, and (2) the other returns a special value if the operation fails (either `null` or `false`, depending on the operation). The regular structure of the interface is illustrated in the following table.

Queue Interface Structure

Type of Operation	Throws exception	Returns special value
Insert	`add(e)`	`offer(e)`
Remove	`remove()`	`poll()`
Examine	`element()`	`peek()`

- Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to their values — see the Object Ordering section for details). Whatever ordering is used, the head of the queue is the element that would be removed by a call to `remove` or `poll`. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every `Queue` implementation must specify its ordering properties.
- It is possible for a `Queue` implementation to restrict the number of elements that it holds; such queues are known as *bounded*. Some `Queue` implementations in `java.util.concurrent` are bounded, but the implementations in `java.util` are not.
- The `add` method, which `Queue` inherits from `Collection`, inserts an element unless it would violate the queue's capacity restrictions, in which case it throws `IllegalStateException`. The `offer` method, which is intended solely for use on bounded queues, differs from `add` only in that it indicates failure to insert an element by returning `false`.
- The `remove` and `poll` methods both remove and return the head of the queue. Exactly which element gets removed is a function of the queue's ordering policy. The `remove` and `poll` methods differ in their behavior only when the queue is empty. Under these circumstances, `remove` throws `NoSuchElementException`, while `poll` returns `null`.
- The `element` and `peek` methods return, but do not remove, the head of the queue. They differ from one another in precisely the same fashion as `remove` and `poll`: If the queue is empty, `element` throws `NoSuchElementException`, while `peek` returns `null`.
- `Queue` implementations generally do not allow insertion of `null` elements. The `LinkedList` implementation, which was retrofitted to implement `Queue`, is an exception. For historical reasons, it permits `null` elements, but you should refrain from taking advantage of this, because `null` is used as a special return value by the `poll` and `peek` methods.
- Queue implementations generally do not define element-based versions of the `equals` and `hashCode` methods but instead inherit the identity-based versions from `Object`.
- The `Queue` interface does not define the blocking queue methods, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the interface `java.util.concurrent.BlockingQueue`, which extends `Queue`.
- In the following example program, a queue is used to implement a countdown timer. The queue is preloaded with all the integer values from a number specified on the command line to zero, in descending order. Then, the values are removed from the queue and printed at one-second intervals. The program is artificial in that it would be more natural to do the same thing without using a queue, but it illustrates the use of a queue to store elements prior to subsequent processing.

```
import java.util.*;

public class Countdown {
    public static void main(String[] args) throws InterruptedException {
        int time = Integer.parseInt(args[0]);
        Queue<Integer> queue = new LinkedList<Integer>();
```

```
        for (int i = time; i >= 0; i--)
            queue.add(i);

        while (!queue.isEmpty()) {
            System.out.println(queue.remove());
            Thread.sleep(1000);
        }
    }
}
```

In the following example, a priority queue is used to sort a collection of elements. Again this program is artificial in that there is no reason to use it in favor of the `sort` method provided in `Collections`, but it illustrates the behavior of priority queues.

```
static <E> List<E> heapSort(Collection<E> c) {
    Queue<E> queue = new PriorityQueue<E>(c);
    List<E> result = new ArrayList<E>();

    while (!queue.isEmpty())
        result.add(queue.remove());

    return result;
}
```

The Map Interface

A `Map` is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction. The `Map` interface includes methods for basic operations (such as `put`, `get`, `remove`, `containsKey`, `containsValue`, `size`, and `empty`), bulk operations (such as `putAll` and `clear`), and collection views (such as `keySet`, `entrySet`, and `values`).

The Java platform contains three general-purpose `Map` implementations: `HashMap`, `TreeMap`, and `LinkedHashMap`. Their behavior and performance are precisely analogous to `HashSet`, `TreeSet`, and `LinkedHashSet`, as described in The Set Interface section.

The remainder of this page discusses the `Map` interface in detail. But first, here are some more examples of collecting to `Map`s using JDK 8 aggregate operations. Modeling real-world objects is a common task in object-oriented programming, so it is reasonable to think that some programs might, for example, group employees by department:

```
// Group employees by department
Map<Department, List<Employee>> byDept = employees.stream()
.collect(Collectors.groupingBy(Employee::getDepartment));
```

Or compute the sum of all salaries by department:

```
// Compute sum of salaries by department
```

```
Map<Department, Integer> totalByDept = employees.stream()
.collect(Collectors.groupingBy(Employee::getDepartment,
Collectors.summingInt(Employee::getSalary)));
```

Or perhaps group students by passing or failing grades:

```
// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing = students.stream()
.collect(Collectors.partitioningBy(s -> s.getGrade()>= PASS_THRESHOLD));
```

You could also group people by city:

```
// Classify Person objects by city
Map<String, List<Person>> peopleByCity
         = personStream.collect(Collectors.groupingBy(Person::getCity));
```

Or even cascade two collectors to classify people by state and city:

```
// Cascade Collectors
Map<String, Map<String, List<Person>>> peopleByStateAndCity
  = personStream.collect(Collectors.groupingBy(Person::getState,
  Collectors.groupingBy(Person::getCity)))
```

Again, these are but a few examples of how to use the new JDK 8 APIs. For in-depth coverage of lambda expressions and aggregate operations see the lesson entitled Aggregate Operations.

Map Interface Basic Operations

The basic operations of `Map` (`put`, `get`, `containsKey`, `containsValue`, `size`, and `isEmpty`) behave exactly like their counterparts in `Hashtable`. The `following program` generates a frequency table of the words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list.

```
import java.util.*;

public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();

        // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }

        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

The only tricky thing about this program is the second argument of the `put` statement. That argument is a conditional expression that has the effect of setting the frequency to one if the word has never been seen before or one more than its current value if the word has already been seen. Try running this program with the command:

```
java Freq if it is to be it is up to me to delegate
```

The program yields the following output.

```
8 distinct words:
{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

Suppose you'd prefer to see the frequency table in alphabetical order. All you have to do is change the implementation type of the `Map` from `HashMap` to `TreeMap`. Making this four-character change causes the program to generate the following output from the same command line.

```
8 distinct words:
{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

Similarly, you could make the program print the frequency table in the order the words first appear on the command line simply by changing the implementation type of the map to `LinkedHashMap`. Doing so results in the following output.

```
8 distinct words:
{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}
```

This flexibility provides a potent illustration of the power of an interface-based framework.

Like the `Set`and `List`interfaces, `Map` strengthens the requirements on the `equals` and `hashCode` methods so that two `Map` objects can be compared for logical equality without regard to their implementation types. Two `Map` instances are equal if they represent the same key-value mappings.

By convention, all general-purpose `Map` implementations provide constructors that take a `Map` object and initialize the new `Map` to contain all the key-value mappings in the specified `Map`. This standard `Map` conversion constructor is entirely analogous to the standard `Collection` constructor: It allows the caller to create a `Map` of a desired implementation type that initially contains all of the mappings in another `Map`, regardless of the other `Map`'s implementation type. For example, suppose you have a `Map`, named `m`. The following one-liner creates a new `HashMap` initially containing all of the same key-value mappings as `m`.

```
Map<K, V> copy = new HashMap<K, V>(m);
```

Map Interface Bulk Operations

The `clear` operation does exactly what you would think it could do: It removes all the mappings from the `Map`. The `putAll` operation is the `Map` analogue of the `Collection` interface's `addAll` operation. In addition to its obvious use of dumping one `Map` into another, it has a second, more subtle use. Suppose a `Map` is used to represent a collection of attribute-value pairs; the `putAll` operation, in combination with the `Map` conversion constructor, provides a neat way to implement attribute map creation with default values. The following is a static factory method that demonstrates this technique.

```
static <K, V> Map<K, V> newAttributeMap(Map<K, V>defaults, Map<K, V>
overrides) {
    Map<K, V> result = new HashMap<K, V>(defaults);
    result.putAll(overrides);
    return result;
}
```

Collection Views

The `Collection` view methods allow a `Map` to be viewed as a `Collection` in these three ways:

- `keySet` — the `Set` of keys contained in the `Map`.
- `values` — The `Collection` of values contained in the `Map`. This `Collection` is not a `Set`, because multiple keys can map to the same value.
- `entrySet` — the `Set` of key-value pairs contained in the `Map`. The `Map` interface provides a small nested interface called `Map.Entry`, the type of the elements in this `Set`.

The `Collection` views provide the *only* means to iterate over a `Map`. This example illustrates the standard idiom for iterating over the keys in a `Map` with a `for-each` construct:

```
for (KeyType key : m.keySet())
    System.out.println(key);
```

and with an `iterator`:

```
// Filter a map based on some
// property of its keys.
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
    if (it.next().isBogus())
        it.remove();
```

The idiom for iterating over values is analogous. Following is the idiom for iterating over key-value pairs.

```
for (Map.Entry<KeyType, ValType> e : m.entrySet())
    System.out.println(e.getKey() + ": " + e.getValue());
```

At first, many people worry that these idioms may be slow because the `Map` has to create a new `Collection` instance each time a `Collection` view operation is called. Rest easy: There's no

reason that a `Map` cannot always return the same object each time it is asked for a given `Collection` view. This is precisely what all the `Map` implementations in `java.util` do.

With all three `Collection` views, calling an `Iterator`'s `remove` operation removes the associated entry from the backing `Map`, assuming that the backing `Map` supports element removal to begin with. This is illustrated by the preceding filtering idiom.

With the `entrySet` view, it is also possible to change the value associated with a key by calling a `Map.Entry`'s `setValue` method during iteration (again, assuming the `Map` supports value modification to begin with). Note that these are the *only* safe ways to modify a `Map` during iteration; the behavior is unspecified if the underlying `Map` is modified in any other way while the iteration is in progress.

The `Collection` views support element removal in all its many forms — `remove`, `removeAll`, `retainAll`, and `clear` operations, as well as the `Iterator.remove` operation. (Yet again, this assumes that the backing `Map` supports element removal.)

The `Collection` views *do not* support element addition under any circumstances. It would make no sense for the `keySet` and `values` views, and it's unnecessary for the `entrySet` view, because the backing `Map`'s `put` and `putAll` methods provide the same functionality.

Fancy Uses of Collection Views: Map Algebra

When applied to the `Collection` views, bulk operations (`containsAll`, `removeAll`, and `retainAll`) are surprisingly potent tools. For starters, suppose you want to know whether one `Map` is a submap of another — that is, whether the first `Map` contains all the key-value mappings in the second. The following idiom does the trick.

```
if (m1.entrySet().containsAll(m2.entrySet())) {
    ...
}
```

Along similar lines, suppose you want to know whether two `Map` objects contain mappings for all of the same keys.

```
if (m1.keySet().equals(m2.keySet())) {
    ...
}
```

Suppose you have a `Map` that represents a collection of attribute-value pairs, and two `Set`s representing required attributes and permissible attributes. (The permissible attributes include the required attributes.) The following snippet determines whether the attribute map conforms to these constraints and prints a detailed error message if it doesn't.

```
static <K, V> boolean validate(Map<K, V> attrMap, Set<K> requiredAttrs,
Set<K>permittedAttrs) {
    boolean valid = true;
```

```
    Set<K> attrs = attrMap.keySet();

    if (! attrs.containsAll(requiredAttrs)) {
        Set<K> missing = new HashSet<K>(requiredAttrs);
        missing.removeAll(attrs);
        System.out.println("Missing attributes: " + missing);
        valid = false;
    }
    if (! permittedAttrs.containsAll(attrs)) {
        Set<K> illegal = new HashSet<K>(attrs);
        illegal.removeAll(permittedAttrs);
        System.out.println("Illegal attributes: " + illegal);
        valid = false;
    }
    return valid;
}
```

Suppose you want to know all the keys common to two `Map` objects.

```
Set<KeyType>commonKeys = new HashSet<KeyType>(m1.keySet());
commonKeys.retainAll(m2.keySet());
```

A similar idiom gets you the common values.

All the idioms presented thus far have been nondestructive; that is, they don't modify the backing `Map`. Here are a few that do. Suppose you want to remove all of the key-value pairs that one `Map` has in common with another.

```
m1.entrySet().removeAll(m2.entrySet());
```

Suppose you want to remove from one `Map` all of the keys that have mappings in another.

```
m1.keySet().removeAll(m2.keySet());
```

What happens when you start mixing keys and values in the same bulk operation? Suppose you have a `Map`, `managers`, that maps each employee in a company to the employee's manager. We'll be deliberately vague about the types of the key and the value objects. It doesn't matter, as long as they're the same. Now suppose you want to know who all the "individual contributors" (or nonmanagers) are. The following snippet tells you exactly what you want to know.

```
Set<Employee>          individualContributors          =          new
HashSet<Employee>(managers.keySet());
individualContributors.removeAll(managers.values());
```

Suppose you want to fire all the employees who report directly to some manager, Simon.

```
Employee simon = ... ;
managers.values().removeAll(Collections.singleton(simon));
```

Note that this idiom makes use of `Collections.singleton`, a static factory method that returns an immutable `Set` with the single, specified element.

Once you've done this, you may have a bunch of employees whose managers no longer work for the company (if any of Simon's direct-reports were themselves managers). The following code will tell you which employees have managers who no longer works for the company.

```
Map<Employee, Employee> m = new HashMap<Employee, Employee>(managers);
m.values().removeAll(managers.keySet());
Set<Employee> slackers = m.keySet();
```

This example is a bit tricky. First, it makes a temporary copy of the `Map`, and it removes from the temporary copy all entries whose (manager) value is a key in the original `Map`. Remember that the original `Map` has an entry for each employee. Thus, the remaining entries in the temporary `Map` comprise all the entries from the original `Map` whose (manager) values are no longer employees. The keys in the temporary copy, then, represent precisely the employees that we're looking for.

There are many more idioms like the ones contained in this section, but it would be impractical and tedious to list them all. Once you get the hang of it, it's not that difficult to come up with the right one when you need it.

Multimaps

A *multimap* is like a `Map` but it can map each key to multiple values. The Java Collections Framework doesn't include an interface for multimaps because they aren't used all that commonly. It's a fairly simple matter to use a `Map` whose values are `List` instances as a multimap. This technique is demonstrated in the next code example, which reads a word list containing one word per line (all lowercase) and prints out all the anagram groups that meet a size criterion. An *anagram group* is a bunch of words, all of which contain exactly the same letters but in a different order. The program takes two arguments on the command line: (1) the name of the dictionary file and (2) the minimum size of anagram group to print out. Anagram groups containing fewer words than the specified minimum are not printed.

There is a standard trick for finding anagram groups: For each word in the dictionary, alphabetize the letters in the word (that is, reorder the word's letters into alphabetical order) and put an entry into a multimap, mapping the alphabetized word to the original word. For example, the word *bad* causes an entry mapping *abd* into *bad* to be put into the multimap. A moment's reflection will show that all the words to which any given key maps form an anagram group. It's a simple matter to iterate over the keys in the multimap, printing out each anagram group that meets the size constraint.

`The following program` is a straightforward implementation of this technique.

```
import java.util.*;
import java.io.*;

public class Anagrams {
    public static void main(String[] args) {
        int minGroupSize = Integer.parseInt(args[1]);
```

```
        // Read words from file and put into a simulated multimap
        Map<String, List<String>> m = new HashMap<String, List<String>>();

        try {
            Scanner s = new Scanner(new File(args[0]));
            while (s.hasNext()) {
                String word = s.next();
                String alpha = alphabetize(word);
                List<String> l = m.get(alpha);
                if (l == null)
                    m.put(alpha, l=new ArrayList<String>());
                l.add(word);
            }
        } catch (IOException e) {
            System.err.println(e);
            System.exit(1);
        }

        // Print all permutation groups above size threshold
        for (List<String> l : m.values())
            if (l.size() >= minGroupSize)
                System.out.println(l.size() + ": " + l);
    }

    private static String alphabetize(String s) {
        char[] a = s.toCharArray();
        Arrays.sort(a);
        return new String(a);
    }
}
```

Running this program on a 173,000-word dictionary file with a minimum anagram group size of eight produces the following output.

```
9: [estrin, inerts, insert, inters, niters, nitres, sinter,
     triens, trines]
8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]
8: [aspers, parses, passer, prases, repass, spares, sparse,
     spears]
10: [least, setal, slate, stale, steal, stela, taels, tales,
      teals, tesla]
8: [enters, nester, renest, rentes, resent, tenser, ternes,
     treens]
8: [arles, earls, lares, laser, lears, rales, reals, seral]
8: [earings, erasing, gainers, reagins, regains, reginas,
     searing, seringa]
8: [peris, piers, pries, prise, ripes, speir, spier, spire]
12: [apers, apres, asper, pares, parse, pears, prase, presa,
      rapes, reaps, spare, spear]
11: [alerts, alters, artels, estral, laster, ratels, salter,
      slater, staler, stelar, talers]
9: [capers, crapes, escarp, pacers, parsec, recaps, scrape,
     secpar, spacer]
9: [palest, palets, pastel, petals, plates, pleats, septal,
     staple, tepals]
```

```
9: [anestri, antsier, nastier, ratines, retains, retinas,
     retsina, stainer, stearin]
8: [ates, east, eats, etas, sate, seat, seta, teas]
8: [carets, cartes, caster, caters, crates, reacts, recast,
     traces]
```

The SortedSet Interface

A `SortedSet` is a `Set` that maintains its elements in ascending order, sorted according to the elements' natural ordering or according to a `Comparator` provided at `SortedSet` creation time. In addition to the normal `Set` operations, the `SortedSet` interface provides operations for the following:

- `Range view` — allows arbitrary range operations on the sorted set
- `Endpoints` — returns the first or last element in the sorted set
- `Comparator access` — returns the `Comparator`, if any, used to sort the set

The code for the `SortedSet` interface follows.

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

Set Operations

The operations that `SortedSet` inherits from `Set` behave identically on sorted sets and normal sets with two exceptions:

- The `Iterator` returned by the `iterator` operation traverses the sorted set in order.
- The array returned by `toArray` contains the sorted set's elements in order.

Although the interface doesn't guarantee it, the `toString` method of the Java platform's `SortedSet` implementations returns a string containing all the elements of the sorted set, in order.

Standard Constructors

By convention, all general-purpose `Collection` implementations provide a standard conversion constructor that takes a `Collection`; `SortedSet` implementations are no exception. In `TreeSet`, this constructor creates an instance that sorts its elements according to their natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified collection was a `SortedSet` instance and, if so, to sort the new `TreeSet` according to the same criterion (comparator or natural ordering). Because `TreeSet` took the approach that it did, it also provides a constructor that takes a `SortedSet` and returns a new `TreeSet` containing the same elements sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines which of these two constructors is invoked (and whether the sorting criterion is preserved).

`SortedSet` implementations also provide, by convention, a constructor that takes a `Comparator` and returns an empty set sorted according to the specified `Comparator`. If `null` is passed to this constructor, it returns a set that sorts its elements according to their natural ordering.

Range-view Operations

The `range-view` operations are somewhat analogous to those provided by the `List` interface, but there is one big difference. Range views of a sorted set remain valid even if the backing sorted set is modified directly. This is feasible because the endpoints of a range view of a sorted set are absolute points in the element space rather than specific elements in the backing collection, as is the case for lists. A `range-view` of a sorted set is really just a window onto whatever portion of the set lies in the designated part of the element space. Changes to the `range-view` write back to the backing sorted set and vice versa. Thus, it's okay to use `range-view`s on sorted sets for long periods of time, unlike `range-view`s on lists.

Sorted sets provide three `range-view` operations. The first, `subSet`, takes two endpoints, like `subList`. Rather than indices, the endpoints are objects and must be comparable to the elements in the sorted set, using the `Set`'s `Comparator` or the natural ordering of its elements, whichever the `Set` uses to order itself. Like `subList`, the range is half open, including its low endpoint but excluding the high one.

Thus, the following line of code tells you how many words between `"doorbell"` and `"pickle"`, including `"doorbell"` but excluding `"pickle"`, are contained in a `SortedSet` of strings called `dictionary`:

```
int count = dictionary.subSet("doorbell", "pickle").size();
```

In like manner, the following one-liner removes all the elements beginning with the letter `f`.

```
dictionary.subSet("f", "g").clear();
```

A similar trick can be used to print a table telling you how many words begin with each letter.

```
for (char ch = 'a'; ch <= 'z'; ) {
    String from = String.valueOf(ch++);
```

```
    String to = String.valueOf(ch);
    System.out.println(from + ": " + dictionary.subSet(from, to).size());
}
```

Suppose you want to view a *closed interval*, which contains both of its endpoints, instead of an open interval. If the element type allows for the calculation of the successor of a given value in the element space, merely request the `subSet` from `lowEndpoint` to `successor(highEndpoint)`. Although it isn't entirely obvious, the successor of a string `s` in `String`'s natural ordering is `s + "\0"` — that is, `s` with a `null` character appended.

Thus, the following one-liner tells you how many words between `"doorbell"` and `"pickle"`, including doorbell *and* pickle, are contained in the dictionary.

```
count = dictionary.subSet("doorbell", "pickle\0").size();
```

A similar technique can be used to view an *open interval*, which contains neither endpoint. The open-interval view from `lowEndpoint` to `highEndpoint` is the half-open interval from `successor(lowEndpoint)` to `highEndpoint`. Use the following to calculate the number of words between `"doorbell"` and `"pickle"`, excluding both.

```
count = dictionary.subSet("doorbell\0", "pickle").size();
```

The `SortedSet` interface contains two more `range-view` operations — `headSet` and `tailSet`, both of which take a single `Object` argument. The former returns a view of the initial portion of the backing `SortedSet`, up to but not including the specified object. The latter returns a view of the final portion of the backing `SortedSet`, beginning with the specified object and continuing to the end of the backing `SortedSet`. Thus, the following code allows you to view the dictionary as two disjoint `volumes` (`a-m` and `n-z`).

```
SortedSet<String> volume1 = dictionary.headSet("n");
SortedSet<String> volume2 = dictionary.tailSet("n");
```

Endpoint Operations

The `SortedSet` interface contains operations to return the first and last elements in the sorted set, not surprisingly called `first` and `last`. In addition to their obvious uses, `last` allows a workaround for a deficiency in the `SortedSet` interface. One thing you'd like to do with a `SortedSet` is to go into the interior of the `Set` and iterate forward or backward. It's easy enough to go forward from the interior: Just get a `tailSet` and iterate over it. Unfortunately, there's no easy way to go backward.

The following idiom obtains the first element that is less than a specified object `o` in the element space.

```
Object predecessor = ss.headSet(o).last();
```

This is a fine way to go one element backward from a point in the interior of a sorted set. It could be applied repeatedly to iterate backward, but this is very inefficient, requiring a lookup for each element returned.

Comparator Accessor

The `SortedSet` interface contains an accessor method called `comparator` that returns the `Comparator` used to sort the set, or `null` if the set is sorted according to the *natural ordering* of its elements. This method is provided so that sorted sets can be copied into new sorted sets with the same ordering.

The SortedMap Interface

A `SortedMap` is a `Map` that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a `Comparator` provided at the time of the `SortedMap` creation. Natural ordering and `Comparator`s are discussed in the Object Ordering section. The `SortedMap` interface provides operations for normal `Map` operations and for the following:

- `Range view` — performs arbitrary range operations on the sorted map
- `Endpoints` — returns the first or the last key in the sorted map
- `Comparator access` — returns the `Comparator`, if any, used to sort the map

The following interface is the `Map` analog of `SortedSet`.

```
public interface SortedMap<K, V> extends Map<K, V>{
    Comparator<? super K> comparator();
    SortedMap<K, V> subMap(K fromKey, K toKey);
    SortedMap<K, V> headMap(K toKey);
    SortedMap<K, V> tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

Map Operations

The operations `SortedMap` inherits from `Map` behave identically on sorted maps and normal maps with two exceptions:

- The `Iterator` returned by the `iterator` operation on any of the sorted map's `Collection` views traverse the collections in order.
- The arrays returned by the `Collection` views' `toArray` operations contain the keys, values, or entries in order.

Although it isn't guaranteed by the interface, the `toString` method of the `Collection` views in all the Java platform's `SortedMap` implementations returns a string containing all the elements of the view, in order.

Standard Constructors

By convention, all general-purpose `Map` implementations provide a standard conversion constructor that takes a `Map`; `SortedMap` implementations are no exception. In `TreeMap`, this constructor creates an instance that orders its entries according to their keys' natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified `Map` instance was a `SortedMap` and, if so, to sort the new map according to the same criterion (comparator or natural ordering). Because `TreeMap` took the approach it did, it also provides a constructor that takes a `SortedMap` and returns a new `TreeMap` containing the same mappings as the given `SortedMap`, sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines whether the `SortedMap` constructor is invoked in preference to the ordinary `map` constructor.

`SortedMap` implementations also provide, by convention, a constructor that takes a `Comparator` and returns an empty map sorted according to the specified `Comparator`. If `null` is passed to this constructor, it returns a `Map` that sorts its mappings according to their keys' natural ordering.

Comparison to SortedSet

Because this interface is a precise `Map` analog of `SortedSet`, all the idioms and code examples in The SortedSet Interface section apply to `SortedMap` with only trivial modifications.

MODULE – 3
STRING METHODS

public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the Character class.

The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. String concatenation is implemented through the StringBuilder(or StringBuffer) class and its append method. String conversions are implemented through the method toString, defined by Object and inherited by all classes in Java. For additional information on string concatenation and conversion, see Gosling, Joy, and Steele, *The Java Language Specification*.

Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a NullPointerException to be thrown.

A String represents a string in the UTF-16 format in which *supplementary characters* are represented by *surrogate pairs* (see the section Unicode Character Representations in the Character class for more information). Index values refer to char code units, so a supplementary character uses two positions in a String.

The String class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., char values).

Constructors

Constructor and Description

String()
Initializes a newly created String object so that it represents an empty character sequence.

String(byte[] bytes)
Constructs a new String by decoding the specified array of bytes using the platform's default charset.

String(byte[] bytes, Charset charset)
Constructs a new String by decoding the specified array of bytes using the specified charset.

String(byte[] ascii, int hibyte)
Deprecated.
This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.

String(byte[] bytes, int offset, int length)
Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.

String(byte[] bytes, int offset, int length, Charset charset)
Constructs a new String by decoding the specified subarray of bytes using the specified charset.

String(byte[] ascii, int hibyte, int offset, int count)
Deprecated.
This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.

String(byte[] bytes, int offset, int length, String charsetName)
Constructs a new String by decoding the specified subarray of bytes using the specified charset.

String(byte[] bytes, String charsetName)
Constructs a new String by decoding the specified array of bytes using the specified charset.

String(char[] value)
Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

String(char[] value, int offset, int count)
Allocates a new String that contains characters from a subarray of the character array argument.

String(int[] codePoints, int offset, int count)

Allocates a new String that contains characters from a subarray of the Unicode code point array argument.

String(String original)
Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

String(StringBuffer buffer)
Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.

String(StringBuilder builder)
Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

public static final Comparator<String> CASE_INSENSITIVE_ORDER
A Comparator that orders String objects as by compareToIgnoreCase. This comparator is serializable.

Note that this Comparator does *not* take locale into account, and will result in an unsatisfactory ordering for certain locales. The java.text package provides *Collators* to allow locale-sensitive ordering.

-
  - **String**

public String()

Initializes a newly created String object so that it represents an empty character sequence. Note that use of this constructor is unnecessary since Strings are immutable.

- **String**

public String(String original)

Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string. Unless an explicit copy of original is needed, use of this constructor is unnecessary since Strings are immutable.

Parameters:
original - A String

- **String**

public String(char[] value)

Allocates a new String so that it represents the sequence of characters currently contained in the character array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:
value - The initial value of the string

- **String**
- public String(char[] value,
- int offset,
  int count)

Allocates a new String that contains characters from a subarray of the character array argument. The offset argument is the index of the first character of the subarray and the count argument specifies the length of the subarray. The contents of the subarray are copied; subsequent modification of the character array does not affect the newly created string.

Parameters:
value - Array that is the source of characters
offset - The initial offset
count - The length
Throws:
IndexOutOfBoundsException - If the offset and count arguments index characters outside the bounds of the value array

- **String**
- public String(int[] codePoints,
- int offset,
  int count)

Allocates a new String that contains characters from a subarray of the Unicode code point array argument. The offset argument is the index of the first code point of the subarray and the count argument specifies the length of the subarray. The contents of the subarray are converted to chars; subsequent modification of the int array does not affect the newly created string.

Parameters:
codePoints - Array that is the source of Unicode code points
offset - The initial offset
count - The length
Throws:
IllegalArgumentException - If any invalid Unicode code point is found in codePoints
IndexOutOfBoundsException - If the offset and count arguments index characters outside the bounds of the codePoints array
Since: 1.5

- **String**
- @Deprecated
- public String(byte[] ascii,
-                 int hibyte,
-                 int offset,

                int count)

Deprecated. This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.

Allocates a new String constructed from a subarray of an array of 8-bit integer values.

The offset argument is the index of the first byte of the subarray, and the count argument specifies the length of the subarray.

Each byte in the subarray is converted to a char as specified in the method above.

Parameters:
ascii - The bytes to be converted to characters
hibyte - The top 8 bits of each 16-bit Unicode code unit
offset - The initial offset
count - The length
Throws:
IndexOutOfBoundsException - If the offset or count argument is invalid
See Also:
String(byte[], int), String(byte[], int, int, java.lang.String), String(byte[], int, int, java.nio.charset.Charset), String(byte[], int, int), String(byte[], java.lang.String), String(byte[], java.nio.charset.Charset), String(byte[])

- **String**
- @Deprecated
- public String(byte[] ascii,

                int hibyte)

Deprecated. This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the String constructors that take a Charset, charset name, or that use the platform's default charset.

Allocates a new String containing characters constructed from an array of 8-bit integer values. Each character *c*in the resulting string is constructed from the corresponding component *b* in the byte array such that:

```
c == (char)(((hibyte & 0xff) << 8)
            | (b & 0xff))
```

Parameters:
ascii - The bytes to be converted to characters
hibyte - The top 8 bits of each 16-bit Unicode code unit
See Also:
String(byte[], int, int, java.lang.String), String(byte[], int, int, java.nio.charset.Charset), String(byte[], int, int), String(byte[], java.lang.String), String(byte[], java.nio.charset.Charset), String(byte[])

- **String**
- public String(byte[] bytes,
- int offset,
- int length,
- String charsetName)
  throws UnsupportedEncodingException

Constructs a new String by decoding the specified subarray of bytes using the specified charset. The length of the new String is a function of the charset, and hence may not be equal to the length of the subarray.

The behavior of this constructor when the given bytes are not valid in the given charset is unspecified. The CharsetDecoder class should be used when more control over the decoding process is required.

Parameters:
bytes - The bytes to be decoded into characters
offset - The index of the first byte to decode
length - The number of bytes to decode
charsetName - The name of a supported charset
Throws:
UnsupportedEncodingException - If the named charset is not supported
IndexOutOfBoundsException - If the offset and length arguments index characters outside the bounds of the bytes array
Since:
JDK1.1

- **String**
- public String(byte[] bytes,
- int offset,
- int length,
  Charset charset)

Constructs a new String by decoding the specified subarray of bytes using the specified charset. The length of the new String is a function of the charset, and hence may not be equal to the length of the subarray.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The CharsetDecoder class should be used when more control over the decoding process is required.

Parameters:
bytes - The bytes to be decoded into characters
offset - The index of the first byte to decode
length - The number of bytes to decode
charset - The charset to be used to decode the bytes
Throws:
IndexOutOfBoundsException - If the offset and length arguments index characters outside the bounds of the bytes array
Since:
1.6

- **String**
- public String(byte[] bytes,
- String charsetName)
  throws UnsupportedEncodingException

Constructs a new String by decoding the specified array of bytes using the specified charset. The length of the new String is a function of the charset, and hence may not be equal to the length of the byte array.

The behavior of this constructor when the given bytes are not valid in the given charset is unspecified. The CharsetDecoder class should be used when more control over the decoding process is required.

Parameters:
bytes - The bytes to be decoded into characters
charsetName - The name of a supported charset
Throws:
UnsupportedEncodingException - If the named charset is not supported
Since:
JDK1.1

- **String**
- public String(byte[] bytes,
  Charset charset)

Constructs a new String by decoding the specified array of bytes using the specified charset. The length of the new String is a function of the charset, and hence may not be equal to the length of the byte array.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement string. The CharsetDecoder class should be used when more control over the decoding process is required.

Parameters:
bytes - The bytes to be decoded into characters
charset - The charset to be used to decode the bytes
Since:
1.6

- **String**
- public String(byte[] bytes,
- int offset,
  int length)

Constructs a new String by decoding the specified subarray of bytes using the platform's default charset. The length of the new String is a function of the charset, and hence may not be equal to the length of the subarray.

The behavior of this constructor when the given bytes are not valid in the default charset is unspecified. The CharsetDecoder class should be used when more control over the decoding process is required.

Parameters:
bytes - The bytes to be decoded into characters
offset - The index of the first byte to decode
length - The number of bytes to decode
Throws:
IndexOutOfBoundsException - If the offset and the length arguments index characters outside the bounds of the bytes array
Since:
JDK1.1

- **String**

public String(byte[] bytes)

Constructs a new String by decoding the specified array of bytes using the platform's default charset. The length of the new String is a function of the charset, and hence may not be equal to the length of the byte array.

The behavior of this constructor when the given bytes are not valid in the default charset is unspecified. The CharsetDecoder class should be used when more control over the decoding process is required.

Parameters:

bytes - The bytes to be decoded into characters
Since:
JDK1.1

- **String**

public String(StringBuffer buffer)

Allocates a new string that contains the sequence of characters currently contained in the string buffer argument. The contents of the string buffer are copied; subsequent modification of the string buffer does not affect the newly created string.

Parameters:
buffer - A StringBuffer

- **String**

public String(StringBuilder builder)

Allocates a new string that contains the sequence of characters currently contained in the string builder argument. The contents of the string builder are copied; subsequent modification of the string builder does not affect the newly created string.

This constructor is provided to ease migration to StringBuilder. Obtaining a string from a string builder via the toString method is likely to run faster and is generally preferred.

Parameters:
builder - A StringBuilder
Since:
1.5

- **Method Detail**
- **length**

public int length()

Returns the length of this string. The length is equal to the number of Unicode code units in the string.

Specified by:
length in interface CharSequence
Returns:
the length of the sequence of characters represented by this object.

- **isEmpty**

public boolean isEmpty()

Returns true if, and only if, length() is 0.

Returns:
true if length() is 0, otherwise false
Since:
1.6

- **charAt**

public char charAt(int index)

Returns the char value at the specified index. An index ranges from 0 to length() - 1. The first char value of the sequence is at index 0, the next at index 1, and so on, as for array indexing.

If the char value specified by the index is a surrogate, the surrogate value is returned.

Specified by:
charAt in interface CharSequence
Parameters:
index - the index of the char value.
Returns:
the char value at the specified index of this string. The first char value is at index 0.
Throws:
IndexOutOfBoundsException - if the index argument is negative or not less than the length of this string.

- **codePointAt**

public int codePointAt(int index)

Returns the character (Unicode code point) at the specified index. The index refers to char values (Unicode code units) and ranges from 0 to length()- 1.

If the char value specified at the given index is in the high-surrogate range, the following index is less than the length of this String, and the char value at the following index is in the low-surrogate range, then the supplementary code point corresponding to this surrogate pair is returned. Otherwise, the char value at the given index is returned.

Parameters:
index - the index to the char values
Returns:
the code point value of the character at the index
Throws:

IndexOutOfBoundsException - if the index argument is negative or not less than the length of this string.
Since:
1.5

- **codePointBefore**

public int codePointBefore(int index)

Returns the character (Unicode code point) before the specified index. The index refers to char values (Unicode code units) and ranges from 1 to length.

If the char value at (index - 1) is in the low-surrogate range, (index - 2) is not negative, and the char value at (index - 2) is in the high-surrogate range, then the supplementary code point value of the surrogate pair is returned. If the char value at index - 1 is an unpaired low-surrogate or a high-surrogate, the surrogate value is returned.

Parameters:
index - the index following the code point that should be returned
Returns:
the Unicode code point value before the given index.
Throws:
IndexOutOfBoundsException - if the index argument is less than 1 or greater than the length of this string.
Since:
1.5

- **codePointCount**
- public int codePointCount(int beginIndex,
  int endIndex)

Returns the number of Unicode code points in the specified text range of this String. The text range begins at the specified beginIndex and extends to the char at index endIndex - 1. Thus the length (in chars) of the text range is endIndex-beginIndex. Unpaired surrogates within the text range count as one code point each.

Parameters:
beginIndex - the index to the first char of the text range.
endIndex - the index after the last char of the text range.
Returns:
the number of Unicode code points in the specified text range
Throws:
IndexOutOfBoundsException - if the beginIndex is negative, or endIndex is larger than the length of this String, or beginIndex is larger than endIndex.
Since:
1.5

- **offsetByCodePoints**
- public int offsetByCodePoints(int index,
  int codePointOffset)

Returns the index within this String that is offset from the given index by codePointOffset code points. Unpaired surrogates within the text range given by index and codePointOffset count as one code point each.

Parameters:
index - the index to be offset
codePointOffset - the offset in code points
Returns:
the index within this String
Throws:
IndexOutOfBoundsException - if index is negative or larger then the length of this String, or if codePointOffset is positive and the substring starting with index has fewer than codePointOffset code points, or if codePointOffset is negative and the substring before index has fewer than the absolute value of codePointOffset code points.
Since:
1.5

- **getChars**
- public void getChars(int srcBegin,
- int srcEnd,
- char[] dst,
  int dstBegin)

Copies characters from this string into the destination character array.

The first character to be copied is at index srcBegin; the last character to be copied is at index srcEnd-1 (thus the total number of characters to be copied is srcEnd-srcBegin). The characters are copied into the subarray of dst starting at index dstBegin and ending at index:

dstBegin + (srcEnd-srcBegin) - 1

Parameters:
srcBegin - index of the first character in the string to copy.
srcEnd - index after the last character in the string to copy.
dst - the destination array.
dstBegin - the start offset in the destination array.
Throws:
IndexOutOfBoundsException - If any of the following is true:

- srcBegin is negative.
- srcBegin is greater than srcEnd
- srcEnd is greater than the length of this string

- dstBegin is negative
- dstBegin+(srcEnd-srcBegin) is larger than dst.length

- **getBytes**
- @Deprecated
- public void getBytes(int srcBegin,
- int srcEnd,
- byte[] dst,

int dstBegin)

Deprecated. This method does not properly convert characters into bytes. As of JDK 1.1, the preferred way to do this is via the getBytes() method, which uses the platform's default charset.

Copies characters from this string into the destination byte array. Each byte receives the 8 low-order bits of the corresponding character. The eight high-order bits of each character are not copied and do not participate in the transfer in any way.

The first character to be copied is at index srcBegin; the last character to be copied is at index srcEnd-1. The total number of characters to be copied is srcEnd-srcBegin. The characters, converted to bytes, are copied into the subarray of dst starting at index dstBegin and ending at index:

dstBegin + (srcEnd-srcBegin) - 1

Parameters:
srcBegin - Index of the first character in the string to copy
srcEnd - Index after the last character in the string to copy
dst - The destination array
dstBegin - The start offset in the destination array
Throws:
IndexOutOfBoundsException - If any of the following is true:

- srcBegin is negative
- srcBegin is greater than srcEnd
- srcEnd is greater than the length of this String
- dstBegin is negative
- dstBegin+(srcEnd-srcBegin) is larger than dst.length

- **getBytes**
- public byte[] getBytes(String charsetName)
  throws UnsupportedEncodingException

Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.

The behavior of this method when this string cannot be encoded in the given charset is unspecified. The CharsetEncoder class should be used when more control over the encoding process is required.

Parameters:
charsetName - The name of a supported charset
Returns:
The resultant byte array
Throws:
UnsupportedEncodingException - If the named charset is not supported
Since:
JDK1.1

- **getBytes**

public byte[] getBytes(Charset charset)

Encodes this String into a sequence of bytes using the given charset, storing the result into a new byte array.

This method always replaces malformed-input and unmappable-character sequences with this charset's default replacement byte array. The CharsetEncoder class should be used when more control over the encoding process is required.

Parameters:
charset - The Charset to be used to encode the String
Returns:
The resultant byte array
Since:
1.6

- **getBytes**

public byte[] getBytes()

Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.

The behavior of this method when this string cannot be encoded in the default charset is unspecified. The CharsetEncoder class should be used when more control over the encoding process is required.

Returns:
The resultant byte array
Since:
JDK1.1

- **equals**

public boolean equals(Object anObject)

Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

Overrides:
equals in class Object
Parameters:
anObject - The object to compare this String against
Returns:
true if the given object represents a String equivalent to this string, false otherwise
See Also:
compareTo(String), equalsIgnoreCase(String)

- **contentEquals**

public boolean contentEquals(StringBuffer sb)

Compares this string to the specified StringBuffer. The result is true if and only if this String represents the same sequence of characters as the specified StringBuffer. This method synchronizes on the StringBuffer.

Parameters:
sb - The StringBuffer to compare this String against
Returns:
true if this String represents the same sequence of characters as the specified StringBuffer, false otherwise
Since:
1.4

- **contentEquals**

public boolean contentEquals(CharSequence cs)

Compares this string to the specified CharSequence. The result is true if and only if this String represents the same sequence of char values as the specified sequence. Note that if the CharSequence is a StringBuffer then the method synchronizes on it.

Parameters:
cs - The sequence to compare this String against
Returns:
true if this String represents the same sequence of char values as the specified sequence, false otherwise
Since:

1.5

- **equalsIgnoreCase**

public boolean equalsIgnoreCase(String anotherString)

Compares this String to another String, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length and corresponding characters in the two strings are equal ignoring case.

Two characters c1 and c2 are considered the same ignoring case if at least one of the following is true:

- The two characters are the same (as compared by the == operator)
- Applying the method Character.toUpperCase(char) to each character produces the same result
- Applying the method Character.toLowerCase(char) to each character produces the same result

Parameters:
anotherString - The String to compare this String against
Returns:
true if the argument is not null and it represents an equivalent String ignoring case; false otherwise
See Also:
equals(Object)

- **compareTo**

public int compareTo(String anotherString)

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this String object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal; compareTo returns 0 exactly when the equals(Object) method would return true.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let *k* be the smallest such index; then the string whose character at position *k* has the smaller value, as determined by using the < operator, lexicographically precedes the other string. In this case, compareTo returns the difference of the two character values at position k in the two string -- that is, the value:

this.charAt(k)-anotherString.charAt(k)

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, compareTo returns the difference of the lengths of the strings -- that is, the value:

this.length()-anotherString.length()

Specified by:
compareTo in interface Comparable<String>
Parameters:
anotherString - the String to be compared.
Returns:
the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

- **compareToIgnoreCase**

public int compareToIgnoreCase(String str)

Compares two strings lexicographically, ignoring case differences. This method returns an integer whose sign is that of calling compareTo with normalized versions of the strings where case differences have been eliminated by calling Character.toLowerCase(Character.toUpperCase(character)) on each character.

Note that this method does *not* take locale into account, and will result in an unsatisfactory ordering for certain locales. The java.text package provides *collators* to allow locale-sensitive ordering.

Parameters:
str - the String to be compared.
Returns:
a negative integer, zero, or a positive integer as the specified String is greater than, equal to, or less than this String, ignoring case considerations.
Since:
1.2
See Also:
Collator.compare(String, String)

- **regionMatches**
- public boolean regionMatches(int toffset,
- String other,
- int ooffset,

int len)

Tests if two string regions are equal.

A substring of this String object is compared to a substring of the argument other. The result is true if these substrings represent identical character sequences. The substring of this String object to be compared begins at index toffset and has length len. The substring of other to be compared begins at index ooffset and has length len. The result is false if and only if at least one of the following is true:

- toffset is negative.
- ooffset is negative.
- toffset+len is greater than the length of this String object.
- ooffset+len is greater than the length of the other argument.
- There is some nonnegative integer *k* less than len such that: this.charAt(toffset + *k*) != other.charAt(ooffset + *k*)

Parameters:
toffset - the starting offset of the subregion in this string.
other - the string argument.
ooffset - the starting offset of the subregion in the string argument.
len - the number of characters to compare.
Returns:
true if the specified subregion of this string exactly matches the specified subregion of the string argument; false otherwise.

- **regionMatches**
- public boolean regionMatches(boolean ignoreCase,
- int toffset,
- String other,
- int ooffset,

int len)

Tests if two string regions are equal.

A substring of this String object is compared to a substring of the argument other. The result is true if these substrings represent character sequences that are the same, ignoring case if and only if ignoreCase is true. The substring of this String object to be compared begins at index toffset and has length len. The substring of other to be compared begins at index ooffset and has length len. The result is false if and only if at least one of the following is true:

- toffset is negative.
- ooffset is negative.
- toffset+len is greater than the length of this String object.
- ooffset+len is greater than the length of the other argument.
- ignoreCase is false and there is some nonnegative integer *k* less than len such that:
- this.charAt(toffset+k) != other.charAt(ooffset+k)

- ignoreCase is true and there is some nonnegative integer *k* less than len such that:
- Character.toLowerCase(this.charAt(toffset+k)) !=
- Character.toLowerCase(other.charAt(ooffset+k))

and:

Character.toUpperCase(this.charAt(toffset+k)) !=
Character.toUpperCase(other.charAt(ooffset+k))

Parameters:
ignoreCase - if true, ignore case when comparing characters.
toffset - the starting offset of the subregion in this string.
other - the string argument.
ooffset - the starting offset of the subregion in the string argument.
len - the number of characters to compare.
Returns:
true if the specified subregion of this string matches the specified subregion of the string argument; false otherwise. Whether the matching is exact or case insensitive depends on the ignoreCase argument.

- **startsWith**
- public boolean startsWith(String prefix,
  int toffset)

Tests if the substring of this string beginning at the specified index starts with the specified prefix.

Parameters:
prefix - the prefix.
toffset - where to begin looking in this string.
Returns:
true if the character sequence represented by the argument is a prefix of the substring of this object starting at index toffset; false otherwise. The result is false if toffset is negative or greater than the length of this String object; otherwise the result is the same as the result of the expression
this.substring(toffset).startsWith(prefix)

- **startsWith**

public boolean startsWith(String prefix)

Tests if this string starts with the specified prefix.

Parameters:

prefix - the prefix.
Returns:
true if the character sequence represented by the argument is a prefix of the character sequence represented by this string; false otherwise. Note also that true will be returned if the argument is an empty string or is equal to this String object as determined by the equals(Object) method.
Since:
1. 0

- **endsWith**

public boolean endsWith(String suffix)

Tests if this string ends with the specified suffix.

Parameters:
suffix - the suffix.
Returns:
true if the character sequence represented by the argument is a suffix of the character sequence represented by this object; false otherwise. Note that the result will be true if the argument is the empty string or is equal to this String object as determined by the equals(Object) method.

- **hashCode**

public int hashCode()

Returns a hash code for this string. The hash code for a String object is computed as

s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]

using int arithmetic, where s[i] is the *i*th character of the string, n is the length of the string, and ^ indicates exponentiation. (The hash value of the empty string is zero.)

Overrides:
hashCode in class Object
Returns:
a hash code value for this object.
See Also:
Object.equals(java.lang.Object), System.identityHashCode(java.lang.Object)

- **indexOf**

public int indexOf(int ch)

Returns the index within this string of the first occurrence of the specified character. If a character with value ch occurs in the character sequence represented by this String object, then the index (in Unicode code units) of the first such occurrence is returned. For values of ch in the range from 0 to 0xFFFF (inclusive), this is the smallest value *k* such that:

this.charAt(*k*) == ch

is true. For other values of ch, it is the smallest value *k* such that:

this.codePointAt(*k*) == ch

is true. In either case, if no such character occurs in this string, then -1 is returned.

Parameters:
ch - a character (Unicode code point).
Returns:
the index of the first occurrence of the character in the character sequence represented by this object, or -1 if the character does not occur.

- **indexOf**
- public int indexOf(int ch,
  int fromIndex)

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

If a character with value ch occurs in the character sequence represented by this String object at an index no smaller than fromIndex, then the index of the first such occurrence is returned. For values of ch in the range from 0 to 0xFFFF (inclusive), this is the smallest value *k* such that:

(this.charAt(*k*) == ch) && (*k* >= fromIndex)

is true. For other values of ch, it is the smallest value *k* such that:

(this.codePointAt(*k*) == ch) && (*k* >= fromIndex)

is true. In either case, if no such character occurs in this string at or after position fromIndex, then -1 is returned.

There is no restriction on the value of fromIndex. If it is negative, it has the same effect as if it were zero: this entire string may be searched. If it is greater than the length of this string, it has the same effect as if it were equal to the length of this string: -1 is returned.

All indices are specified in char values (Unicode code units).

Parameters:
ch - a character (Unicode code point).
fromIndex - the index to start the search from.
Returns:
the index of the first occurrence of the character in the character sequence represented by this object that is greater than or equal to fromIndex, or -1 if the character does not occur.

- **lastIndexOf**

public int lastIndexOf(int ch)

Returns the index within this string of the last occurrence of the specified character. For values of ch in the range from 0 to 0xFFFF (inclusive), the index (in Unicode code units) returned is the largest value *k* such that:

this.charAt(*k*) == ch

is true. For other values of ch, it is the largest value *k* such that:

this.codePointAt(*k*) == ch

is true. In either case, if no such character occurs in this string, then -1 is returned. The String is searched backwards starting at the last character.

Parameters:
ch - a character (Unicode code point).
Returns:
the index of the last occurrence of the character in the character sequence represented by this object, or -1 if the character does not occur.

- **lastIndexOf**
- public int lastIndexOf(int ch,
  int fromIndex)

Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index. For values of ch in the range from 0 to 0xFFFF (inclusive), the index returned is the largest value *k* such that:

(this.charAt(*k*) == ch) && (*k* <= fromIndex)

is true. For other values of ch, it is the largest value *k* such that:

(this.codePointAt(*k*) == ch) && (*k* <= fromIndex)

is true. In either case, if no such character occurs in this string at or before position fromIndex, then -1 is returned.

All indices are specified in char values (Unicode code units).

Parameters:
ch - a character (Unicode code point).
fromIndex - the index to start the search from. There is no restriction on the value of fromIndex. If it is greater than or equal to the length of this string, it has the same effect as if it were equal to one less than the length of this string: this entire string may be searched. If it is negative, it has the same effect as if it were -1: -1 is returned.
Returns:
the index of the last occurrence of the character in the character sequence represented by this object that is less than or equal to fromIndex, or -1 if the character does not occur before that point.

- **indexOf**

public int indexOf(String str)

Returns the index within this string of the first occurrence of the specified substring.

The returned index is the smallest value *k* for which:

this.startsWith(str, *k*)

If no such value of *k* exists, then -1 is returned.

Parameters:
str - the substring to search for.
Returns:
the index of the first occurrence of the specified substring, or -1 if there is no such occurrence.

- **indexOf**
- public int indexOf(String str,
  int fromIndex)

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

The returned index is the smallest value *k* for which:

k >= fromIndex && this.startsWith(str, *k*)

If no such value of *k* exists, then -1 is returned.

Parameters:
str - the substring to search for.
fromIndex - the index from which to start the search.
Returns:
the index of the first occurrence of the specified substring, starting at the specified index, or -1 if there is no such occurrence.

- **lastIndexOf**

public int lastIndexOf(String str)

Returns the index within this string of the last occurrence of the specified substring. The last occurrence of the empty string "" is considered to occur at the index value this.length().

The returned index is the largest value *k* for which:

this.startsWith(str, *k*)

If no such value of *k* exists, then -1 is returned.

Parameters:
str - the substring to search for.
Returns:
the index of the last occurrence of the specified substring, or -1 if there is no such occurrence.

- **lastIndexOf**
- public int lastIndexOf(String str,
  int fromIndex)

Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

The returned index is the largest value *k* for which:

k <= fromIndex && this.startsWith(str, *k*)

If no such value of *k* exists, then -1 is returned.

Parameters:
str - the substring to search for.
fromIndex - the index to start the search from.
Returns:
the index of the last occurrence of the specified substring, searching backward from the specified index, or -1 if there is no such occurrence.

- **substring**

public String substring(int beginIndex)

Returns a string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

Examples:

```
"unhappy".substring(2) returns "happy"
"Harbison".substring(3) returns "bison"
"emptiness".substring(9) returns "" (an empty string)
```

Parameters:
beginIndex - the beginning index, inclusive.
Returns:
the specified substring.
Throws:
IndexOutOfBoundsException - if beginIndex is negative or larger than the length of this String object.

- **substring**
- public String substring(int beginIndex,
  int endIndex)

Returns a string that is a substring of this string. The substring begins at the specified beginIndex and extends to the character at index endIndex - 1. Thus the length of the substring is endIndex-beginIndex.

Examples:

```
"hamburger".substring(4, 8) returns "urge"
"smiles".substring(1, 5) returns "mile"
```

Parameters:

beginIndex - the beginning index, inclusive.
endIndex - the ending index, exclusive.
Returns:
the specified substring.
Throws:
IndexOutOfBoundsException - if the beginIndex is negative, or endIndex is larger than the length of this String object, or beginIndex is larger than endIndex.

- **subSequence**
- public CharSequence subSequence(int beginIndex,
 int endIndex)

Returns a character sequence that is a subsequence of this sequence.

An invocation of this method of the form

str.subSequence(begin, end)

behaves in exactly the same way as the invocation

str.substring(begin, end)
Specified by:
subSequence in interface CharSequence
API Note:
This method is defined so that the String class can implement the CharSequence interface.
Parameters:
beginIndex - the begin index, inclusive.
endIndex - the end index, exclusive.
Returns:
the specified subsequence.
Throws:
IndexOutOfBoundsException - if beginIndex or endIndex is negative, if endIndex is greater than length(), or if beginIndex is greater than endIndex
Since:
1.4

- **concat**

public String concat(String str)

Concatenates the specified string to the end of this string.

If the length of the argument string is 0, then this String object is returned. Otherwise, a String object is returned that represents a character sequence that is the concatenation of the character sequence represented by this String object and the character sequence represented by the argument string.

Examples:

"cares".concat("s") returns "caress"
"to".concat("get").concat("her") returns "together"

Parameters:
str - the String that is concatenated to the end of this String.
Returns:
a string that represents the concatenation of this object's characters followed by the string argument's characters.

- **replace**
- public String replace(char oldChar,
  char newChar)

Returns a string resulting from replacing all occurrences of oldChar in this string with newChar.

If the character oldChar does not occur in the character sequence represented by this String object, then a reference to this String object is returned. Otherwise, a String object is returned that represents a character sequence identical to the character sequence represented by this String object, except that every occurrence of oldChar is replaced by an occurrence of newChar.

Examples:

"mesquite in your cellar".replace('e', 'o')
    returns "mosquito in your collar"
"the war of baronets".replace('r', 'y')
    returns "the way of bayonets"
"sparring with a purple porpoise".replace('p', 't')
    returns "starring with a turtle tortoise"
"JonL".replace('q', 'x') returns "JonL" (no change)

Parameters:
oldChar - the old character.
newChar - the new character.
Returns:
a string derived from this string by replacing every occurrence of oldChar with newChar.

- **matches**

public boolean matches(String regex)

Tells whether or not this string matches the given regular expression.

An invocation of this method of the form *str*.matches(*regex*) yields exactly the same result as the expression

Pattern.matches(*regex*, *str*)

Parameters:
regex - the regular expression to which this string is to be matched
Returns:
true if, and only if, this string matches the given regular expression
Throws:
PatternSyntaxException - if the regular expression's syntax is invalid
Since:
1.4
See Also:
Pattern

- **contains**

public boolean contains(CharSequence s)

Returns true if and only if this string contains the specified sequence of char values.

Parameters:
s - the sequence to search for
Returns:
true if this string contains s, false otherwise
Since:
1.5

- **replaceFirst**
- public String replaceFirst(String regex,
  String replacement)

Replaces the first substring of this string that matches the given regular expression with the given replacement.

An invocation of this method of the form *str*.replaceFirst(*regex*, *repl*) yields exactly the same result as the expression

Pattern.compile(*regex*).matcher(*str*).replaceFirst(*repl*)

Note that backslashes (\) and dollar signs ($) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string; see Matcher.replaceFirst(java.lang.String). Use Matcher.quoteReplacement(java.lang.String) to suppress the special meaning of these characters, if desired.

Parameters:
regex - the regular expression to which this string is to be matched
replacement - the string to be substituted for the first match
Returns:
The resulting String
Throws:
PatternSyntaxException - if the regular expression's syntax is invalid
Since:
1.4
See Also:
Pattern

- **replaceAll**
- public String replaceAll(String regex,
  String replacement)

Replaces each substring of this string that matches the given regular expression with the given replacement.

An invocation of this method of the form *str*.replaceAll(*regex*, *repl*) yields exactly the same result as the expression

Pattern.compile(*regex*).matcher(*str*).replaceAll(*repl*)

Note that backslashes (\) and dollar signs ($) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string; see Matcher.replaceAll. Use Matcher.quoteReplacement(java.lang.String) to suppress the special meaning of these characters, if desired.

Parameters:
regex - the regular expression to which this string is to be matched
replacement - the string to be substituted for each match
Returns:
The resulting String
Throws:
PatternSyntaxException - if the regular expression's syntax is invalid
Since:
1.4
See Also:
Pattern

- **replace**
- public String replace(CharSequence target,
  CharSequence replacement)

Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence. The replacement proceeds from the beginning of the string to the end, for example, replacing "aa" with "b" in the string "aaa" will result in "ba" rather than "ab".

Parameters:
target - The sequence of char values to be replaced
replacement - The replacement sequence of char values
Returns:
The resulting string
Since:
1.5

- **split**
- public String[] split(String regex,
  int limit)

Splits this string around matches of the given regular expression.

The array returned by this method contains each substring of this string that is terminated by another substring that matches the given expression or is terminated by the end of the string. The substrings in the array are in the order in which they occur in this string. If the expression does not match any part of the input then the resulting array has just one element, namely this string.

When there is a positive-width match at the beginning of this string then an empty leading substring is included at the beginning of the resulting array. A zero-width match at the beginning however never produces such empty leading substring.

The limit parameter controls the number of times the pattern is applied and therefore affects the length of the resulting array. If the limit *n* is greater than zero then the pattern will be applied at most *n* - 1 times, the array's length will be no greater than *n*, and the array's last entry will contain all input beyond the last matched delimiter. If *n* is non-positive then the pattern will be applied as many times as possible and the array can have any length. If *n* is zero then the pattern will be applied as many times as possible, the array can have any length, and trailing empty strings will be discarded.

The string "boo:and:foo", for example, yields the following results with these parameters:

Regex	Limit	Result
:	2	{ "boo", "and:foo" }
:	5	{ "boo", "and", "foo" }
:	-2	{ "boo", "and", "foo" }
o	5	{ "b", "", ":and:f", "", "" }
o	-2	{ "b", "", ":and:f", "", "" }
o	0	{ "b", "", ":and:f" }

An invocation of this method of the form *str*.split(*regex*, *n*) yields the same result as the expression

Pattern.compile(*regex*).split(*str*, *n*)

Parameters:
regex - the delimiting regular expression
limit - the result threshold, as described above
Returns:
the array of strings computed by splitting this string around matches of the given regular expression
Throws:
PatternSyntaxException - if the regular expression's syntax is invalid
Since:
1.4
See Also:
Pattern

- **split**

public String[] split(String regex)

Splits this string around matches of the given regular expression.

This method works as if by invoking the two-argument split method with the given expression and a limit argument of zero. Trailing empty strings are therefore not included in the resulting array.

The string "boo:and:foo", for example, yields the following results with these expressions:

Regex	Result
:	{ "boo", "and", "foo" }
o	{ "b", "", ":and:f" }

Parameters:
regex - the delimiting regular expression
Returns:
the array of strings computed by splitting this string around matches of the given regular expression
Throws:
PatternSyntaxException - if the regular expression's syntax is invalid
Since:
1.4
See Also:
Pattern

- **join**

- public static String join(CharSequence delimiter,
  CharSequence... elements)

Returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.

For example,

```
String message = String.join("-", "Java", "is", "cool");
// message returned is: "Java-is-cool"
```

Note that if an element is null, then "null" is added.

Parameters:
delimiter - the delimiter that separates each element
elements - the elements to join together.
Returns:
a new String that is composed of the elements separated by the delimiter
Throws:
NullPointerException - If delimiter or elements is null
Since:
1.8
See Also:
StringJoiner

- **join**
- public static String join(CharSequence delimiter,
  Iterable<? extends CharSequence> elements)

Returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.

For example,

```
List<String> strings = new LinkedList<>();
strings.add("Java");strings.add("is");
strings.add("cool");
String message = String.join(" ", strings);
//message returned is: "Java is cool"

Set<String> strings = new LinkedHashSet<>();
strings.add("Java"); strings.add("is");
strings.add("very"); strings.add("cool");
```

```
String message = String.join("-", strings);
//message returned is: "Java-is-very-cool"
```

Note that if an individual element is null, then "null" is added.

Parameters:
delimiter - a sequence of characters that is used to separate each of the elements in the resulting String
elements - an Iterable that will have its elements joined together.
Returns:
a new String that is composed from the elements argument
Throws:
NullPointerException - If delimiter or elements is null
Since:
1.8
See Also:
join(CharSequence,CharSequence...), StringJoiner

- **toLowerCase**

public String toLowerCase(Locale locale)

Converts all of the characters in this String to lower case using the rules of the given Locale. Case mapping is based on the Unicode Standard version specified by the Character class. Since case mappings are not always 1:1 char mappings, the resulting String may be a different length than the original String.

Examples of lowercase mappings are in the following table:

Language Code of Locale	Upper Case	Lower Case	Description
tr (Turkish)	\u0130	\u0069	capital letter I with dot above -> small letter i
tr (Turkish)	\u0049	\u0131	capital letter I -> small letter dotless i
(all)	French Fries	french fries	lowercased all chars in String
(all)	ΙΧΘΥΣ	ιχθυς	lowercased all chars in String

Parameters:
locale - use the case transformation rules for this locale
Returns:
the String, converted to lowercase.
Since:

1.1
See Also:
toLowerCase(), toUpperCase(), toUpperCase(Locale)

- **toLowerCase**

public String toLowerCase()

Converts all of the characters in this String to lower case using the rules of the default locale. This is equivalent to calling toLowerCase(Locale.getDefault()).

Note: This method is locale sensitive, and may produce unexpected results if used for strings that are intended to be interpreted locale independently. Examples are programming language identifiers, protocol keys, and HTML tags. For instance, "TITLE".toLowerCase() in a Turkish locale returns "t\u0131tle", where '\u0131' is the LATIN SMALL LETTER DOTLESS I character. To obtain correct results for locale insensitive strings, use toLowerCase(Locale.ROOT).

Returns:
the String, converted to lowercase.
See Also:
toLowerCase(Locale)

- **toUpperCase**

public String toUpperCase(Locale locale)

Converts all of the characters in this String to upper case using the rules of the given Locale. Case mapping is based on the Unicode Standard version specified by the Character class. Since case mappings are not always 1:1 char mappings, the resulting String may be a different length than the original String.

Examples of locale-sensitive and 1:M case mappings are in the following table.

Language Code of Locale	Lower Case	Upper Case	Description
tr (Turkish)	\u0069	\u0130	small letter i -> capital letter I with dot above
tr (Turkish)	\u0131	\u0049	small letter dotless i -> capital letter I
(all)	\u00df	\u0053 \u0053	small letter sharp s -> two letters: SS
(all)	Fahrvergnügen	FAHRVERGNÜGEN	

Parameters:

locale - use the case transformation rules for this locale
Returns:
the String, converted to uppercase.
Since:
1.1
See Also:
toUpperCase(), toLowerCase(), toLowerCase(Locale)

- **toUpperCase**

public String toUpperCase()

Converts all of the characters in this String to upper case using the rules of the default locale. This method is equivalent to toUpperCase(Locale.getDefault()).

Note: This method is locale sensitive, and may produce unexpected results if used for strings that are intended to be interpreted locale independently. Examples are programming language identifiers, protocol keys, and HTML tags. For instance, "title".toUpperCase() in a Turkish locale returns "T\u0130TLE", where '\u0130' is the LATIN CAPITAL LETTER I WITH DOT ABOVE character. To obtain correct results for locale insensitive strings, use toUpperCase(Locale.ROOT).

Returns:
the String, converted to uppercase.
See Also:
toUpperCase(Locale)

- **trim**

public String trim()

Returns a string whose value is this string, with any leading and trailing whitespace removed.

If this String object represents an empty character sequence, or the first and last characters of character sequence represented by this String object both have codes greater than '\u0020' (the space character), then a reference to this String object is returned.

Otherwise, if there is no character with a code greater than '\u0020' in the string, then a String object representing an empty string is returned.

Otherwise, let *k* be the index of the first character in the string whose code is greater than '\u0020', and let *m* be the index of the last character in the string whose code is greater than '\u0020'. A String object is returned, representing the substring of this string that begins with the character at index *k* and ends with the character at index *m*-that is, the result of this.substring(k, m + 1).

This method may be used to trim whitespace (as defined above) from the beginning and end of a string.

Returns:
A string whose value is this string, with any leading and trailing white space removed, or this string if it has no leading or trailing white space.

- **toString**

public String toString()

This object (which is already a string!) is itself returned.

Specified by:
toString in interface CharSequence
Overrides:
toString in class Object
Returns:
the string itself.

- **toCharArray**

public char[] toCharArray()

Converts this string to a new character array.

Returns:
a newly allocated character array whose length is the length of this string and whose contents are initialized to contain the character sequence represented by this string.

- **format**
- public static String format(String format,
  Object... args)

Returns a formatted string using the specified format string and arguments.

The locale always used is the one returned by Locale.getDefault().

Parameters:
format - A format string
args - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by *The Java™ Virtual Machine Specification*. The behaviour on a null argument depends on the conversion.
Returns:

A formatted string
Throws:
IllegalFormatException - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the Details section of the formatter class specification.
Since:
1.5
See Also:
Formatter

- **format**
- public static String format(Locale l,
- String format,
  Object... args)

Returns a formatted string using the specified locale, format string, and arguments.

Parameters:
l - The locale to apply during formatting. If l is null then no localization is applied.
format - A format string
args - Arguments referenced by the format specifiers in the format string. If there are more arguments than format specifiers, the extra arguments are ignored. The number of arguments is variable and may be zero. The maximum number of arguments is limited by the maximum dimension of a Java array as defined by *The Java™ Virtual Machine Specification*. The behaviour on a null argument depends on the conversion.
Returns:
A formatted string
Throws:
IllegalFormatException - If a format string contains an illegal syntax, a format specifier that is incompatible with the given arguments, insufficient arguments given the format string, or other illegal conditions. For specification of all possible formatting errors, see the Details section of the formatter class specification
Since:
1.5
See Also:
Formatter

- **valueOf**

public static String valueOf(Object obj)

Returns the string representation of the Object argument.

Parameters:
obj - an Object.

Returns:
if the argument is null, then a string equal to "null"; otherwise, the value of obj.toString() is returned.
See Also:
Object.toString()

- **valueOf**

public static String valueOf(char[] data)

Returns the string representation of the char array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the returned string.

Parameters:
data - the character array.
Returns:
a String that contains the characters of the character array.

- **valueOf**
- public static String valueOf(char[] data,
- int offset,
  int count)

Returns the string representation of a specific subarray of the char array argument.

The offset argument is the index of the first character of the subarray. The count argument specifies the length of the subarray. The contents of the subarray are copied; subsequent modification of the character array does not affect the returned string.

Parameters:
data - the character array.
offset - initial offset of the subarray.
count - length of the subarray.
Returns:
a String that contains the characters of the specified subarray of the character array.
Throws:
IndexOutOfBoundsException - if offset is negative, or count is negative, or offset+count is larger than data.length.

- **copyValueOf**
- public static String copyValueOf(char[] data,
- int offset,
  int count)

Equivalent to valueOf(char[], int, int).

Parameters:
data - the character array.
offset - initial offset of the subarray.
count - length of the subarray.
Returns:
a String that contains the characters of the specified subarray of the character array.
Throws:
IndexOutOfBoundsException - if offset is negative, or count is negative, or offset+count is larger than data.length.

- **copyValueOf**

public static String copyValueOf(char[] data)

Equivalent to valueOf(char[]).

Parameters:
data - the character array.
Returns:
a String that contains the characters of the character array.

- **valueOf**

public static String valueOf(boolean b)

Returns the string representation of the boolean argument.

Parameters:
b - a boolean.
Returns:
if the argument is true, a string equal to "true" is returned; otherwise, a string equal to "false" is returned.

- **valueOf**

public static String valueOf(char c)

Returns the string representation of the char argument.

Parameters:
c - a char.
Returns:
a string of length 1 containing as its single character the argument c.

- **valueOf**

public static String valueOf(int i)

Returns the string representation of the int argument.

The representation is exactly the one returned by the Integer.toString method of one argument.

Parameters:
i - an int.
Returns:
a string representation of the int argument.
See Also:
Integer.toString(int, int)

- **valueOf**

public static String valueOf(long l)

Returns the string representation of the long argument.

The representation is exactly the one returned by the Long.toString method of one argument.

Parameters:
l - a long.
Returns:
a string representation of the long argument.
See Also:
Long.toString(long)

- **valueOf**

public static String valueOf(float f)

Returns the string representation of the float argument.

The representation is exactly the one returned by the Float.toString method of one argument.

Parameters:
f - a float.
Returns:
a string representation of the float argument.
See Also:
Float.toString(float)

- **valueOf**

public static String valueOf(double d)

Returns the string representation of the double argument.

The representation is exactly the one returned by the Double.toString method of one argument.

Parameters:
d - a double.
Returns:
a string representation of the double argument.
See Also:
Double.toString(double)

- **intern**

public String intern()

Returns a canonical representation for the string object.

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true.

All literal strings and string-valued constant expressions are interned. String literals are defined in section 3.10.5 of the *The Java™ Language Specification*.

Returns:
a string that has the same contents as this string, but is guaranteed to be from a pool of unique strings.

Class StringBuffer

- java.lang.Object
-
  - java.lang.StringBuffer

- All Implemented Interfaces:

Serializable, Appendable, CharSequence

public final class StringBuffer
extends Object
implements Serializable, CharSequence

A thread-safe, mutable sequence of characters. A string buffer is like a String, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

String buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

The principal operations on a StringBuffer are the append and insert methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string buffer. The append method always adds these characters at the end of the buffer; the insert method adds the characters at a specified point.

For example, if z refers to a string buffer object whose current contents are "start", then the method call z.append("le") would cause the string buffer to contain "startle", whereas z.insert(4, "le") would alter the string buffer to contain "starlet".

In general, if sb refers to an instance of a StringBuffer, then sb.append(x) has the same effect as sb.insert(sb.length(), x).

Whenever an operation occurs involving a source sequence (such as appending or inserting from a source sequence), this class synchronizes only on the string buffer performing the operation, not on the source. Note that while StringBuffer is designed to be safe to use concurrently from multiple threads, if the constructor or the append or insert operation is passed a source sequence that is shared across threads, the calling code must ensure that the operation has a consistent and unchanging view of the source sequence for the duration of the operation. This could be satisfied by the caller holding a lock during the operation's call, by using an immutable source sequence, or by not sharing the source sequence across threads.

Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger.

Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a NullPointerException to be thrown.

As of release JDK 5, this class has been supplemented with an equivalent class designed for use by a single thread, StringBuilder. The StringBuilder class should generally be used in

preference to this one, as it supports all of the same operations but it is faster, as it performs no synchronization.

Constructor and Description

StringBuffer()
Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

StringBuffer(CharSequence seq)
Constructs a string buffer that contains the same characters as the specified CharSequence.

StringBuffer(int capacity)
Constructs a string buffer with no characters in it and the specified initial capacity.

StringBuffer(String str)
Constructs a string buffer initialized to the contents of the specified string.

-

- **StringBuffer**

public StringBuffer()

Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

- **StringBuffer**

public StringBuffer(int capacity)

Constructs a string buffer with no characters in it and the specified initial capacity.

Parameters:

capacity - the initial capacity.

Throws:

NegativeArraySizeException - if the capacity argument is less than 0.

- **StringBuffer**

public StringBuffer(String str)

Constructs a string buffer initialized to the contents of the specified string. The initial capacity of the string buffer is 16 plus the length of the string argument.

Parameters:

str - the initial contents of the buffer.

- **StringBuffer**

public StringBuffer(CharSequence seq)

Constructs a string buffer that contains the same characters as the specified CharSequence. The initial capacity of the string buffer is 16 plus the length of the CharSequence argument.

If the length of the specified CharSequence is less than or equal to zero, then an empty buffer of capacity 16 is returned.

Parameters:

seq - the sequence to copy.

Since:

1.5

- **Method Detail**

- **length**

public int length()

Returns the length (character count).

Specified by:

length in interface CharSequence

Returns:

the length of the sequence of characters currently represented by this object

- **capacity**

public int capacity()

Returns the current capacity. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur.

Returns:

the current capacity

- **ensureCapacity**

public void ensureCapacity(int minimumCapacity)

Ensures that the capacity is at least equal to the specified minimum. If the current capacity is less than the argument, then a new internal array is allocated with greater capacity. The new capacity is the larger of:

- The minimumCapacity argument.
- Twice the old capacity, plus 2.

If the minimumCapacity argument is nonpositive, this method takes no action and simply returns. Note that subsequent operations on this object can reduce the actual capacity below that requested here.

Parameters:

minimumCapacity - the minimum desired capacity.

- **trimToSize**

public void trimToSize()

Attempts to reduce storage used for the character sequence. If the buffer is larger than necessary to hold its current sequence of characters, then it may be resized to become more space efficient. Calling this method may, but is not required to, affect the value returned by a subsequent call to the capacity() method.

Since:

1.5

- **setLength**

public void setLength(int newLength)

Sets the length of the character sequence. The sequence is changed to a new character sequence whose length is specified by the argument. For every nonnegative index *k* less than newLength, the character at index *k* in the new character sequence is the same as the character at index *k* in the old sequence if *k* is less than the length of the old character sequence; otherwise, it is the null character '\u0000'. In other words, if the newLength argument is less than the current length, the length is changed to the specified length.

If the newLength argument is greater than or equal to the current length, sufficient null characters ('\u0000') are appended so that length becomes the newLength argument.

The newLength argument must be greater than or equal to 0.

Parameters:

newLength - the new length

Throws:

IndexOutOfBoundsException - if the newLength argument is negative.

See Also:

length()

- **charAt**

public char charAt(int index)

Returns the char value in this sequence at the specified index. The first char value is at index 0, the next at index 1, and so on, as in array indexing.

The index argument must be greater than or equal to 0, and less than the length of this sequence.

If the char value specified by the index is a surrogate, the surrogate value is returned.

Specified by:

charAt in interface CharSequence

Parameters:

index - the index of the desired char value.

Returns:

the char value at the specified index.

Throws:

IndexOutOfBoundsException - if index is negative or greater than or equal to length().

See Also:

length()

- **codePointAt**

public int codePointAt(int index)

Returns the character (Unicode code point) at the specified index. The index refers to char values (Unicode code units) and ranges from 0 to length()- 1.

If the char value specified at the given index is in the high-surrogate range, the following index is less than the length of this sequence, and the char value at the following index is in the low-surrogate range, then the supplementary code point corresponding to this surrogate pair is returned. Otherwise, the char value at the given index is returned.

Parameters:

index - the index to the char values

Returns:

the code point value of the character at the index

Since:

1.5

- **codePointBefore**

public int codePointBefore(int index)

Returns the character (Unicode code point) before the specified index. The index refers to char values (Unicode code units) and ranges from 1 to length().

If the char value at (index - 1) is in the low-surrogate range, (index - 2) is not negative, and the char value at (index - 2) is in the high-surrogate range, then the supplementary code point value of the surrogate pair is returned. If the char value at index - 1 is an unpaired low-surrogate or a high-surrogate, the surrogate value is returned.

Parameters:

index - the index following the code point that should be returned

Returns:

the Unicode code point value before the given index.

Since:

1.5

- **codePointCount**
- public int codePointCount(int beginIndex,

  int endIndex)

Returns the number of Unicode code points in the specified text range of this sequence. The text range begins at the specified beginIndex and extends to the char at index endIndex - 1. Thus the length (in chars) of the text range is endIndex-beginIndex. Unpaired surrogates within this sequence count as one code point each.

Parameters:

beginIndex - the index to the first char of the text range.

endIndex - the index after the last char of the text range.

Returns:

the number of Unicode code points in the specified text range

Since:

1.5

- **offsetByCodePoints**
- public int offsetByCodePoints(int index,

  int codePointOffset)

Returns the index within this sequence that is offset from the given index by codePointOffset code points. Unpaired surrogates within the text range given by index and codePointOffset count as one code point each.

Parameters:

index - the index to be offset

codePointOffset - the offset in code points

Returns:

the index within this sequence

Since:

1.5

- **getChars**
- public void getChars(int srcBegin,
- int srcEnd,

- char[] dst,

  int dstBegin)

Characters are copied from this sequence into the destination character array dst. The first character to be copied is at index srcBegin; the last character to be copied is at index srcEnd-1. The total number of characters to be copied is srcEnd-srcBegin. The characters are copied into the subarray of dst starting at index dstBegin and ending at index:

dstbegin + (srcEnd-srcBegin) - 1

Parameters:

srcBegin - start copying at this offset.

srcEnd - stop copying at this offset.

dst - the array to copy the data into.

dstBegin - offset into dst.

Throws:

IndexOutOfBoundsException - if any of the following is true:

- srcBegin is negative
- dstBegin is negative
- the srcBegin argument is greater than the srcEnd argument.
- srcEnd is greater than this.length().
- dstBegin+srcEnd-srcBegin is greater than dst.length

- **setCharAt**
- public void setCharAt(int index,

  char ch)

The character at the specified index is set to ch. This sequence is altered to represent a new character sequence that is identical to the old character sequence, except that it contains the character ch at position index.

The index argument must be greater than or equal to 0, and less than the length of this sequence.

Parameters:

index - the index of the character to modify.

ch - the new character.

Throws:

IndexOutOfBoundsException - if index is negative or greater than or equal to length().

See Also:

length()

- **append**

public StringBuffer append(Object obj)

Appends the string representation of the Object argument.

The overall effect is exactly as if the argument were converted to a string by the method String.valueOf(Object), and the characters of that string were then appended to this character sequence.

Parameters:

obj - an Object.

Returns:

a reference to this object.

- **append**

public StringBuffer append(String str)

Appends the specified string to this character sequence.

The characters of the String argument are appended, in order, increasing the length of this sequence by the length of the argument. If str is null, then the four characters "null" are appended.

Let *n* be the length of this character sequence just prior to execution of the append method. Then the character at index *k* in the new character sequence is equal to the character at index *k* in the old character sequence, if *k* is less than *n*; otherwise, it is equal to the character at index *k*-*n* in the argument str.

Parameters:

str - a string.

Returns:

a reference to this object.

- **append**

public StringBuffer append(StringBuffer sb)

Appends the specified StringBuffer to this sequence.

The characters of the StringBuffer argument are appended, in order, to the contents of this StringBuffer, increasing the length of this StringBuffer by the length of the argument. If sb is null, then the four characters "null" are appended to this StringBuffer.

Let *n* be the length of the old character sequence, the one contained in the StringBuffer just prior to execution of the append method. Then the character at index *k* in the new character sequence is equal to the character at index *k* in the old character sequence, if *k* is less than *n*; otherwise, it is equal to the character at index *k-n* in the argument sb.

This method synchronizes on this, the destination object, but does not synchronize on the source (sb).

Parameters:

sb - the StringBuffer to append.

Returns:

a reference to this object.

Since:

1.4

- **append**

public StringBuffer append(CharSequence s)

Appends the specified CharSequence to this sequence.

The characters of the CharSequence argument are appended, in order, increasing the length of this sequence by the length of the argument.

The result of this method is exactly the same as if it were an invocation of this.append(s, 0, s.length());

This method synchronizes on this, the destination object, but does not synchronize on the source (s).

If s is null, then the four characters "null" are appended.

Specified by:

append in interface Appendable

Parameters:

s - the CharSequence to append.

Returns:

a reference to this object.

Since:

1.5

- **append**
- public StringBuffer append(CharSequence s,
- int start,

int end)

Appends a subsequence of the specified CharSequence to this sequence.

Characters of the argument s, starting at index start, are appended, in order, to the contents of this sequence up to the (exclusive) index end. The length of this sequence is increased by the value of end - start.

Let *n* be the length of this character sequence just prior to execution of the append method. Then the character at index *k* in this character sequence becomes equal to the character at index *k* in this sequence, if *k* is less than *n*; otherwise, it is equal to the character at index *k+start-n* in the argument s.

If s is null, then this method appends characters as if the s parameter was a sequence containing the four characters "null".

Specified by:

append in interface Appendable

Parameters:

s - the sequence to append.

start - the starting index of the subsequence to be appended.

end - the end index of the subsequence to be appended.

Returns:

a reference to this object.

Throws:

IndexOutOfBoundsException - if start is negative, or start is greater than end or end is greater than s.length()

Since:

1.5

- **append**

public StringBuffer append(char[] str)

Appends the string representation of the char array argument to this sequence.

The characters of the array argument are appended, in order, to the contents of this sequence. The length of this sequence increases by the length of the argument.

The overall effect is exactly as if the argument were converted to a string by the method String.valueOf(char[]), and the characters of that string were then appended to this character sequence.

Parameters:

str - the characters to be appended.

Returns:

a reference to this object.

- **append**
- public StringBuffer append(char[] str,
- int offset,

int len)

Appends the string representation of a subarray of the char array argument to this sequence.

Characters of the char array str, starting at index offset, are appended, in order, to the contents of this sequence. The length of this sequence increases by the value of len.

The overall effect is exactly as if the arguments were converted to a string by the method String.valueOf(char[],int,int), and the characters of that string were then appended to this character sequence.

Parameters:

str - the characters to be appended.

offset - the index of the first char to append.

len - the number of chars to append.

Returns:

a reference to this object.

Throws:

IndexOutOfBoundsException - if offset < 0 or len < 0 or offset+len > str.length

- **append**

public StringBuffer append(boolean b)

Appends the string representation of the boolean argument to the sequence.

The overall effect is exactly as if the argument were converted to a string by the method String.valueOf(boolean), and the characters of that string were then appended to this character sequence.

Parameters:

b - a boolean.

Returns:

a reference to this object.

- **append**

public StringBuffer append(char c)

Appends the string representation of the char argument to this sequence.

The argument is appended to the contents of this sequence. The length of this sequence increases by 1.

The overall effect is exactly as if the argument were converted to a string by the method String.valueOf(char), and the character in that string were then appended to this character sequence.

Specified by:

append in interface Appendable

Parameters:

c - a char.

Returns:

a reference to this object.

- **append**

public StringBuffer append(int i)

Appends the string representation of the int argument to this sequence.

The overall effect is exactly as if the argument were converted to a string by the method String.valueOf(int), and the characters of that string were then appended to this character sequence.

Parameters:

i - an int.

Returns:

a reference to this object.

- **appendCodePoint**

public StringBuffer appendCodePoint(int codePoint)

Appends the string representation of the codePoint argument to this sequence.

The argument is appended to the contents of this sequence. The length of this sequence increases by Character.charCount(codePoint).

The overall effect is exactly as if the argument were converted to a char array by the method Character.toChars(int) and the character in that array were then appended to this character sequence.

Parameters:

codePoint - a Unicode code point

Returns:

a reference to this object.

Since:

1.5

- **append**

public StringBuffer append(long lng)

Appends the string representation of the long argument to this sequence.

The overall effect is exactly as if the argument were converted to a string by the method String.valueOf(long), and the characters of that string were then appended to this character sequence.

Parameters:

lng - a long.

Returns:

a reference to this object.

- **append**

public StringBuffer append(float f)

Appends the string representation of the float argument to this sequence.

The overall effect is exactly as if the argument were converted to a string by the method String.valueOf(float), and the characters of that string were then appended to this character sequence.

Parameters:

f - a float.

Returns:

a reference to this object.

- **append**

public StringBuffer append(double d)

Appends the string representation of the double argument to this sequence.

The overall effect is exactly as if the argument were converted to a string by the method String.valueOf(double), and the characters of that string were then appended to this character sequence.

Parameters:

d - a double.

Returns:

a reference to this object.

- **delete**
- public StringBuffer delete(int start,

  int end)

Removes the characters in a substring of this sequence. The substring begins at the specified start and extends to the character at index end - 1 or to the end of the sequence if no such character exists. If start is equal to end, no changes are made.

Parameters:

start - The beginning index, inclusive.

end - The ending index, exclusive.

Returns:

This object.

Throws:

StringIndexOutOfBoundsException - if start is negative, greater than length(), or greater than end.

Since:

1.2

- **deleteCharAt**

public StringBuffer deleteCharAt(int index)

Removes the char at the specified position in this sequence. This sequence is shortened by one char.

Note: If the character at the given index is a supplementary character, this method does not remove the entire character. If correct handling of supplementary characters is required, determine the number of chars to remove by calling Character.charCount(thisSequence.codePointAt(index)), where thisSequence is this sequence.

Parameters:

index - Index of char to remove

Returns:

This object.

Throws:

StringIndexOutOfBoundsException - if the index is negative or greater than or equal to length().

Since:

1.2

- **replace**
- public StringBuffer replace(int start,
- int end,

String str)

Replaces the characters in a substring of this sequence with characters in the specified String. The substring begins at the specified start and extends to the character at index end - 1 or to the end of the sequence if no such character exists. First the characters in the substring are removed and then the specified String is inserted at start. (This sequence will be lengthened to accommodate the specified String if necessary.)

Parameters:

start - The beginning index, inclusive.

end - The ending index, exclusive.

str - String that will replace previous contents.

Returns:

This object.

Throws:

StringIndexOutOfBoundsException - if start is negative, greater than length(), or greater than end.

Since:

1.2

- **substring**

public String substring(int start)

Returns a new String that contains a subsequence of characters currently contained in this character sequence. The substring begins at the specified index and extends to the end of this sequence.

Parameters:

start - The beginning index, inclusive.

Returns:

The new string.

Throws:

StringIndexOutOfBoundsException - if start is less than zero, or greater than the length of this object.

Since:

1.2

- **subSequence**
- public CharSequence subSequence(int start,

  int end)

Returns a new character sequence that is a subsequence of this sequence.

An invocation of this method of the form

sb.subSequence(begin, end)

behaves in exactly the same way as the invocation

sb.substring(begin, end)

This method is provided so that this class can implement the CharSequence interface.

Specified by:

subSequence in interface CharSequence

Parameters:

start - the start index, inclusive.

end - the end index, exclusive.

Returns:

the specified subsequence.

Throws:

IndexOutOfBoundsException - if start or end are negative, if end is greater than length(), or if start is greater than end

Since:

1.4

- **substring**
- public String substring(int start,

  int end)

Returns a new String that contains a subsequence of characters currently contained in this sequence. The substring begins at the specified start and extends to the character at index end - 1.

Parameters:

start - The beginning index, inclusive.

end - The ending index, exclusive.

Returns:

The new string.

Throws:

StringIndexOutOfBoundsException - if start or end are negative or greater than length(), or start is greater than end.

Since:

1.2

- **insert**
- public StringBuffer insert(int index,
- char[] str,
- int offset,

int len)

Inserts the string representation of a subarray of the str array argument into this sequence. The subarray begins at the specified offset and extends len chars. The characters of the subarray are inserted into this sequence at the position indicated by index. The length of this sequence increases by len chars.

Parameters:

index - position at which to insert subarray.

str - A char array.

offset - the index of the first char in subarray to be inserted.

len - the number of chars in the subarray to be inserted.

Returns:

This object

Throws:

StringIndexOutOfBoundsException - if index is negative or greater than length(), or offset or len are negative, or (offset+len) is greater than str.length.

Since:

1.2

- **insert**
- public StringBuffer insert(int offset,

  Object obj)

Inserts the string representation of the Object argument into this character sequence.

The overall effect is exactly as if the second argument were converted to a string by the method String.valueOf(Object), and the characters of that string were then inserted into this character sequence at the indicated offset.

The offset argument must be greater than or equal to 0, and less than or equal to the length of this sequence.

Parameters:

offset - the offset.

obj - an Object.

Returns:

a reference to this object.

Throws:

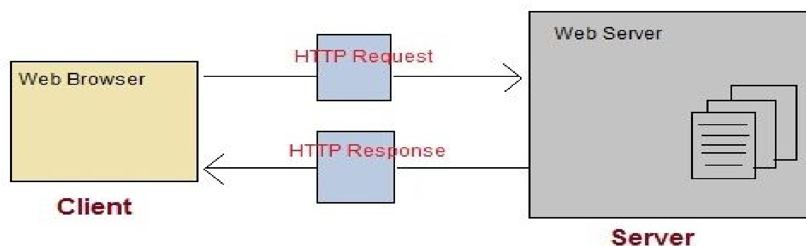
StringIndexOutOfBoundsException - if the offset is invalid.

MODULE - 4

SERVLETS

Introduction to Web

The web clients make requests to web server. When a server answers a request, it usually sends some type of content(**MIME-** Multi purpose Internet Mail Exchange) to the client. The client uses web browser to send request to the server. The server often sends response to the browser with a set of instructions written in HTML(HyperText Markup Language)



Before Servlets, **CGI(Common Gateway Interface)** programming was used to create web applications. Here's how a CGI program works :

Drawbacks of CGI programs

- High resposne time because CGI programs execute in their own OS shell.
- CGI is not scalable.
- CGI programs are not always secure or object-oriented.
- It is Platform dependent.

Because of these disadvantages, developers started looking for better CGI solutions. And then Sun Microsystems developed **Servlet** as a solution.

Servlet

Servlet technology is used to create web application (resides at server side and generates dynamic web page).

Servlet can be described in many ways, depending on the context.

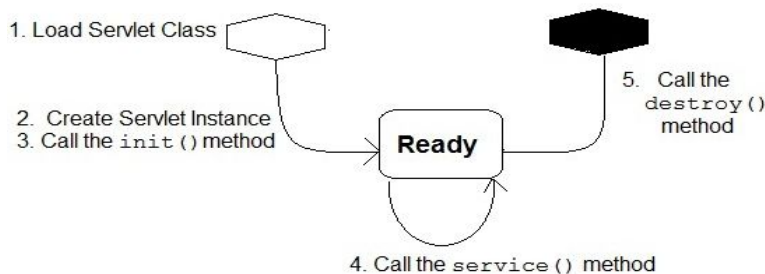
- Servlet is server side program.
- Servlet is an API that provides many interfaces and classes.
- Servlet is a web component that is deployed on the server to create dynamic web page.

Advantages of using Servlets

1. **better performance:** because it creates a thread for each request not process.
2. **Portability:** Servlets are platform independent because it uses java language.
3. **Robust:** Servlets are managed by JVM so we don't need to worry about memory leak, garbage collection etc.
4. **Secure:** servlets are object oriented and runs inside JVM.
5. Servlets are scalable.

Life Cycle of a Servlet

The web container maintains the life cycle of a servlet instance



1. **Loading Servlet Class :** A Servlet class is loaded when first request for the servlet is received by the Web Container.
2. **Servlet instance creation :**After the Servlet class is loaded, Web Container creates the instance of it. Servlet instance is created only once in the life cycle.
3. **Call to the init() method :** init() method is called by the Web Container on servlet instance to initialize the servlet.

   public void **init**(ServletConfig config) throws ServletException

4. **Call to the service() method :** The containers call the service() method each time the request for servlet is received. The service() method will then call the doGet() or doPost() methos based ont eh type of the HTTP request, as explained in previous lessons.
   public void **service**(ServletRequest request, ServletResponse response) throws ServletException, IOException
5. **Call to destroy() method:** The Web Container call the destroy() method before removing servlet instance, giving it a chance for cleanup activity.

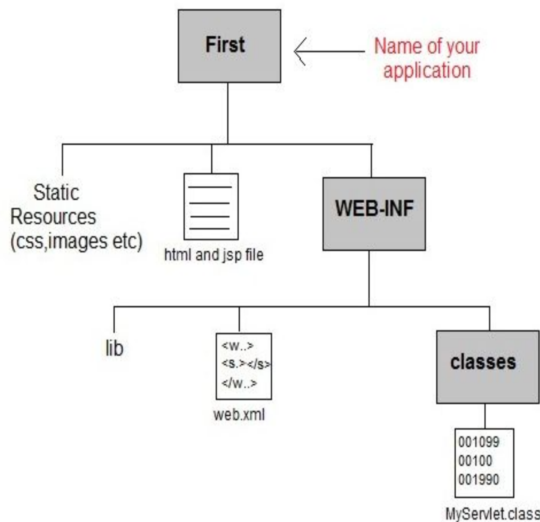
Steps to Create Servlet Application using tomcat server

To create a Servlet application you need to follow the below mentioned steps. These steps are common for all the Web server. In our example we are using Apache Tomcat server. Apache Tomcat is an open source web server for testing servlets and JSP technology. Create directory structure for your application.

1. Create directory structure for your application.
2. Create a Servlet
3. Compile the Servlet
4. Create Deployement Descriptor for your application
5. Start the server and deploy the application
6.

1. Creating the Directory Structure

Sun Microsystem defines a unique directory structure that must be followed to create a servlet application.



Create the above directory structure inside **Apache-Tomcat\webapps** directory.

All HTML, static files(images, css etc) are kept directly under **Web application** folder.

While all the Servlet classes are kept inside `classes` folder.

The `web.xml` (deployement descriptor) file is kept under `WEB-INF` folder.

2. Creating a Servlet

There are three different ways to create a servlet.

- By extending **HttpServlet** class
- By extending **GenericServlet** class
- By implementing **Servlet** interface

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

// extending HttpServlet class

public MyServlet extends HttpServlet
{
 public void doGet(HttpServletRequest request,HttpServletResposne response)
                    throws ServletException
 {
   response.setContentType("text/html"); // set content type
   //get the stream to write the data
```

```
  PrintWriter out = response.getWriter(); // create printwriter object
  out.println("<h1>Hello Readers</h1>"); // print ur content on client web browser
  }
}
```

Write above code and save it as **MyServlet.java** anywhere on your PC. Compile it from there and paste the class file into WEB-INF/classes/ directory that you have to create inside **Tomcat/webapps** directory.

```
import javax.servlet.*;
import java.io.*;

// extending GenericServlet class

public MyServlet extends GenericServlet
{
 public void service(ServletRequest request,ServletResponse response)
   throws IOException,ServletException{
 {
  response.setContentType("text/html"); // set content type
  //get the stream to write the data
  PrintWriter out = response.getWriter(); // create printwriter object
  out.println("<h1>Hello Readers</h1>"); // print ur content on client web browser
  }
}
```

3. Compiling a Servlet

To compile a Servlet a JAR file is required. Different servers require different JAR files. In Apache Tomcat server servlet-api.jar file is required to compile a servlet class.

- Download **servlet-api.jar** file.
- Paste the servlet-api.jar file inside Java\jdk\jre\lib\ext directory.

NOTE: After compiling your Servlet class you will have to paste the class file into WEB-INF/classes/ directory.

4. Create Deployment Descriptor

Deployment Descriptor(DD) is an XML document that is used by Web Container to run Servlets and JSP pages.

web.xml file

```
<web-app>
    <servlet>
        <servlet-name> MyServlet </servlet-name>
        <servlet-class> MyServlet </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name> MyServlet </servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```

5. Starting Tomcat Server for the first time

set JAVA_HOME or JRE_HOME in environment variable (It is required to start server).

Go to My Computer properties -> Click on advanced tab then environment variables -> Click on the new tab of user variable -> Write JAVA_HOME in variable name and paste the path of jdk folder in variable value -> ok

Run Servlet Application

Open Browser and type **http:localhost:8080/First/hello**

Servlet API

Servlet API consists of two important packages that encapsulates all the important classes and interface, namely :

1. **javax.servlet.*;**

INTERFACES	**CLASSES**
Servlet	ServletInputStream
ServletContext	ServletOutputStream
ServletConfig	ServletException
ServletRequest	UnavailableException
ServletResponse	GenericServlet

Interface Summary	
Servlet	Defines methods that all servlets must implement.
ServletRequest	Defines an object to provide client request information to a servlet.
ServletResponse	Defines an object to assist a servlet in sending a response to the client.
ServletConfig	A servlet configuration object used by a servlet container to pass information to a servlet during initialization.
ServletContext	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.

Interface Servlet

Method	Description
void destroy( )	Called when the servlet is unloaded.
ServletConfig getServletConfig( )	Returns a **ServletConfig** object that contains any initialization parameters.
String getServletInfo( )	Returns a string describing the servlet.
void init(ServletConfig *sc*) throws ServletException	Called when the servlet is initialized. Initialization parameters for the servlet can be obtained from *sc*. An **UnavailableException** should be thrown if the servlet cannot be initialized.
void service(ServletRequest *req*, ServletResponse *res*) throws ServletException, IOException	Called to process a request from a client. The request from the client can be read from *req*. The response to the client can be written to *res*. An exception is generated if a servlet or IO problem occurs.

Interface ServletRequest

Method	Description
Object getAttribute(String *attr*)	Returns the value of the attribute named *attr*.
String getCharacterEncoding( )	Returns the character encoding of the request.
int getContentLength( )	Returns the size of the request. The value –1 is returned if the size is unavailable.
String getContentType( )	Returns the type of the request. A **null** value is returned if the type cannot be determined.
ServletInputStream getInputStream( ) throws IOException	Returns a **ServletInputStream** that can be used to read binary data from the request. An **IllegalStateException** is thrown if **getReader( )** has already been invoked for this request.
String getParameter(String *pname*)	Returns the value of the parameter named *pname*.
Enumeration getParameterNames( )	Returns an enumeration of the parameter names for this request.
String[ ] getParameterValues(String *name*)	Returns an array containing values associated with the parameter specified by *name*.
String getProtocol( )	Returns a description of the protocol.
BufferedReader getReader( ) throws IOException	Returns a buffered reader that can be used to read text from the request. An **IllegalStateException** is thrown if **getInputStream( )** has already been invoked for this request.
String getRemoteAddr( )	Returns the string equivalent of the client IP address.
String getRemoteHost( )	Returns the string equivalent of the client host name.
String getScheme( )	Returns the transmission scheme of the URL used for the request (for example, “http”, “ftp”).
String getServerName( )	Returns the name of the server.
int getServerPort( )	Returns the port number.

Interface ServletResponse

Method	Description
String getCharacterEncoding( )	Returns the character encoding for the response.
ServletOutputStream getOutputStream( ) throws IOException	Returns a **ServletOutputStream** that can be used to write binary data to the response. An **IllegalStateException** is thrown if **getWriter( )** has already been invoked for this request.
PrintWriter getWriter( ) throws IOException	Returns a **PrintWriter** that can be used to write character data to the response. An **IllegalStateException** is thrown if **getOutputStream( )** has already been invoked for this request.
void setContentLength(int *size*)	Sets the content length for the response to *size*.
void setContentType(String *type*)	Sets the content type for the response to *type*.

Interface **ServletConfig**

Method	Description
ServletContext getServletContext( )	Returns the context for this servlet.
String getInitParameter(String *param*)	Returns the value of the initialization parameter named *param*.
Enumeration getInitParameterNames( )	Returns an enumeration of all initialization parameter names.
String getServletName( )	Returns the name of the invoking servlet.

Interface **ServletContext**

Method	Description
Object getAttribute(String *attr*)	Returns the value of the server attribute named *attr*.
String getMimeType(String *file*)	Returns the MIME type of *file*.
String getRealPath(String *vpath*)	Returns the real path that corresponds to the virtual path *vpath*.
String getServerInfo( )	Returns information about the server.
void log(String *s*)	Writes *s* to the servlet log.
void log(String *s*, Throwable *e*)	Writes *s* and the stack trace for *e* to the servlet log.
void setAttribute(String *attr*, Object *val*)	Sets the attribute specified by *attr* to the value passed in *val*.

Class Summary	
GenericServlet	Defines a generic, protocol-independent servlet.
ServletInputStream	Provides an input stream for reading binary data from a client request, including an efficient readLine method for reading data one line at a time.
ServletOutputStream	Provides an output stream for sending binary data to the client.
ServletException	Defines a general exception a servlet can throw when it encounters difficulty.
UnavailableException	Defines an exception that a servlet or filter throws to indicate that it is permanently or temporarily unavailable.

Class GenericServlet

java.lang.Object
└ **javax.servlet.GenericServlet**

All Implemented Interfaces: Servlet, ServletConfig

Class ServletInputStream

java.lang.Object
└ java.io.InputStream
└ **javax.servlet.ServletInputStream**

Method Summary	
int	**readLine**(byte[] b, int off, int len) Reads the input stream, one line at a time.

Class ServletOutputStream

Method Summary	
void	**println**()
void	**print**(java.lang.String s) **Writes a String to the client, without a carriage return-line feed (CRLF) character at the end.**

void	**println()** **Writes a carriage return-line feed (CRLF) to the client.**
void	**println(java.lang.String s)** **Writes a String to the client, followed by a carriage return-line feed (CRLF).**

2. **javax.servlet.http.*;**

CLASSES	**INTERFACES**
Cookie	HttpServletRequest
HttpServlet	HttpServletResponse
HttpSessionBindingEvent	HttpSession

Interface Summary	
HttpServletRequest	Extends the ServletRequest interface to provide request information for HTTP servlets.
HttpServletResponse	Extends the ServletResponse interface to provide HTTP-specific functionality in sending a response.
HttpSession	Provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.

Interface HttpServletResponse

Method	Description
void addCookie(Cookie *cookie*)	Adds *cookie* to the HTTP response.
boolean containsHeader(String *field*)	Returns **true** if the HTTP response header contains a field named *field*.
String encodeURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs generated by a servlet should be processed by this method.
String encodeRedirectURL(String *url*)	Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs passed to **sendRedirect( )** should be processed by this method.
void sendError(int *c*) throws IOException	Sends the error code *c* to the client.
void sendError(int *c*, String *s*) throws IOException	Sends the error code *c* and message *s* to the client.
void sendRedirect(String *url*) throws IOException	Redirects the client to *url*.
void setDateHeader(String *field*, long *msec*)	Adds *field* to the header with date value equal to *msec* (milliseconds since midnight, January 1, 1970, GMT).
void setHeader(String *field*, String *value*)	Adds *field* to the header with value equal to *value*.
void setIntHeader(String *field*, int *value*)	Adds *field* to the header with value equal to *value*.
void setStatus(int *code*)	Sets the status code for this response to *code*.

Interface HttpServletRequest

Method	Description
String getAuthType( )	Returns authentication scheme.
Cookie[ ] getCookies( )	Returns an array of the cookies in this request.
long getDateHeader(String *field*)	Returns the value of the date header field named *field*.
String getHeader(String *field*)	Returns the value of the header field named *field*.
Enumeration getHeaderNames( )	Returns an enumeration of the header names.
int getIntHeader(String *field*)	Returns the **int** equivalent of the header field named *field*.
String getMethod( )	Returns the HTTP method for this request.
String getPathInfo( )	Returns any path information that is located after the servlet path and before a query string of the URL.
String getPathTranslated( )	Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path.
String getQueryString( )	Returns any query string in the URL.
String getRemoteUser( )	Returns the name of the user who issued this request.
String getRequestedSessionId( )	Returns the ID of the session.
String getRequestURI( )	Returns the URI.
StringBuffer getRequestURL( )	Returns the URL.
String getServletPath( )	Returns that part of the URL that identifies the servlet.
HttpSession getSession( )	Returns the session for this request. If a session does not exist, one is created and then returned.
HttpSession getSession(boolean *new*)	If *new* is **true** and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request.
boolean isRequestedSessionIdFromCookie( )	Returns **true** if a cookie contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdFromURL( )	Returns **true** if the URL contains the session ID. Otherwise, returns **false**.
boolean isRequestedSessionIdValid( )	Returns **true** if the requested session ID is valid in the current session context.

Interface HttpSession

Method	Description
Object getAttribute(String *attr*)	Returns the value associated with the name passed in *attr*. Returns **null** if *attr* is not found.
Enumeration getAttributeNames( )	Returns an enumeration of the attribute names associated with the session.
long getCreationTime( )	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when this session was created.
String getId( )	Returns the session ID.
long getLastAccessedTime( )	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request for this session.
void invalidate( )	Invalidates this session and removes it from the context.
boolean isNew( )	Returns **true** if the server created the session and it has not yet been accessed by the client.
void removeAttribute(String *attr*)	Removes the attribute specified by *attr* from the session.
void setAttribute(String *attr*, Object *val*)	Associates the value passed in *val* with the attribute name passed in *attr*.

Class Summary	
Cookie	Creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server.
HttpServlet	Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site.
HttpSessionBindingEvent	Events of this type are either sent to an object that implements HttpSessionBindingListener when it is bound or unbound from a session, or to a HttpSessionAttributeListener that has been configured in the deployment descriptor when any attribute is bound, unbound or replaced in a session.
HttpSessionEvent	This is the class representing event notifications for changes to sessions within a web application.

Class Cookie

Method	Description
Object clone( )	Returns a copy of this object.
String getComment( )	Returns the comment.
String getDomain( )	Returns the domain.
int getMaxAge( )	Returns the maximum age (in seconds).
String getName( )	Returns the name.
String getPath( )	Returns the path.
boolean getSecure( )	Returns **true** if the cookie is secure. Otherwise, returns **false**.
String getValue( )	Returns the value.
int getVersion( )	Returns the version.
void setComment(String *c*)	Sets the comment to *c*.
void setDomain(String *d*)	Sets the domain to *d*.
void setMaxAge(int *secs*)	Sets the maximum age of the cookie to *secs*. This is the number of seconds after which the cookie is deleted.
void setPath(String *p*)	Sets the path to *p*.
void setSecure(boolean *secure*)	Sets the security flag to *secure*.
void setValue(String *v*)	Sets the value to *v*.
void setVersion(int *v*)	Sets the version to *v*.

Class HttpServlet

Method	Description
void doDelete(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP DELETE request.
void doGet(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP GET request.
void doHead(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP HEAD request.
void doOptions(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP OPTIONS request.
void doPost(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP POST request.
void doPut(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP PUT request.
void doTrace(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Handles an HTTP TRACE request.
long getLastModified(HttpServletRequest *req*)	Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified.
void service(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException	Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively.

Class HttpSessionBindingEvent

Method Summary	
java.lang.String	**getName**() Returns the name with which the attribute is bound to or unbound from the session.
HttpSession	**getSession**() Return the session that changed.
java.lang.Object	**getValue**()

	Returns the value of the attribute that has been added, removed or replaced.

Class HttpSessionEvent

Method Summary	
HttpSession	**getSession**() Return the session that changed.

Reading Servlet parameters

In this example, we will show how a parameter is passed to a

index.html

```
<form method="post" action="check">
        Name <input type="text" name="user" >
        <input type="submit" value="submit">
</form>
```

MyServlet.java

```
public class MyServlet extends HttpServlet {

  protected void doPost(request, response){
     ... // set content type . . .
        String user=request.getParameter("user");

        out.println("<h2> Welcome "+user+"</h2>");
   }
}
```

NOTE: getParameter( ) returns string to get int value use

Integer.parseInt("sting");

Handling HTTP request and Response

HttpServlet class provides various methods that handle various types of HTTP request.

A servlet typically must override at least one method, usually one of these:

- doGet, if the servlet supports HTTP GET requests
- doPost, for HTTP POST requests
- doPut, for HTTP PUT requests
- doDelete, for HTTP DELETE requests

GET and POST methods are commonly used when handling form input.

NOTE: By default a request is Get request.

Difference between GET and POST requests

GET Request	POST Request
Data is sent in header to the server	Data is sent in the request body
Get request can send only limited amount of data	Large amount of data can be sent.
Get request is not secured because data is exposed in URL	Post request is secured because data is not exposed in URL.

Session

Session simply means a particular interval of time. **Session Tracking** is a way to maintain state (data) of an user. It is also known as **session management** in servlet.

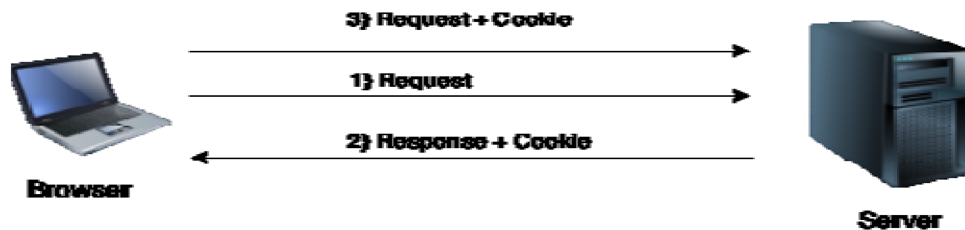
Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.

There are 2 techniques used in Session tracking:

1. **Cookies**
2. **HttpSession**

Cookies in Servlet

A **cookie** is a small piece of information that is persisted between the multiple client requests. By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.



Advantage of Cookies

1. Simplest technique of maintaining the state.
2. Cookies are maintained at client side.

Disadvantage of Cookies

1. It will not work if cookie is disabled from the browser.
2. Only textual information can be set in Cookie object.

Creating a new Cookie

```
Cookie ck = new Cookie("username",name);
```

creating a new cookie object

Setting up lifespan for a cookie

```
ck.setMaxAge(30*60);
```

setting maximum age of cookie

Sending the cookie to the client

```
response.addCookie(ck);
```

adding cookie to **response** object

Getting cookies from client request

```
Cookie[] cks = request.getCookies();
```

getting the cookie for **request** object

Example demonstrating usage of Cookies

index.html

```
<form method="post" action=" MyServlet ">
        Name:<input type="text" name="user" /><br/>
        Password:<input type="text" name="pass" ><br/>
            <input type="submit" value="submit">
</form>
```

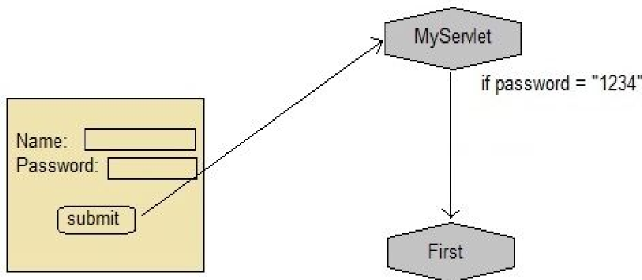
MyServlet.java

```
public class MyServlet extends HttpServlet {

  protected void doPost(HttpServletRequest request, HttpServletResponse response)
    {    // . . . . .
      String name = request.getParameter("user");
      String pass = request.getParameter("pass");

      if(pass.equals("1234"))
      {
        Cookie ck = new Cookie("username",name);
        response.addCookie(ck);

                //response.sendRedirect("First");//call ur servlet

                //creating submit button
                out.print("<form action= First >");
                out.print("<input type='submit' value='go'>");
                out.print("</form>");
      }
    }
}
```

First.java

```
public class First extends HttpServlet {
```

```
 protected void doGet(HttpServletRequest request, HttpServletResponse response)
   {     // . . .
       Cookie[] cks = request.getCookies();
       out.println("Welcome "+cks[0].getValue());
   }
}
```



Useful Methods of Cookie class

Method	Description
public void setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds.
public String getName()	Returns the name of the cookie. The name cannot be changed after creation.
public String getValue()	Returns the value of the cookie.
public void setName(String name)	changes the name of the cookie.
public void setValue(String value)	changes the value of the cookie.

Other methods required for using Cookies

For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces. They are:

1. **public void addCookie(Cookie ck):**method of HttpServletResponse interface is used to add cookie in response object.
2. **public Cookie[ ] getCookies():**method of HttpServletRequest interface is used to return all the cookies from the browser.

HttpSession

HttpSession object is used to store entire session with a specific client. We can store, retrieve and remove attribute from **HttpSession** object. Any servlet can have access to **HttpSession** object throughout the getSession() method.

Creating a new session

getSession() method returns a session. If the session already exist, it return the esisting session else create a new sesion

```
HttpSession session = request.getSession();
```

```
HttpSession session = request.getSession(true);
```

getSession(true) always return a new session

Getting a pre-existing session

```
HttpSession session = request.getSession(false);
```

return a pre-existing session

Destroying a session

```
session.invalidate();
```

destroy a session

Some Important Methods of HttpSession

Methods	Description
long getCreationTime()	returns the time when the session was created, measured in milliseconds since midnight January 1, 1970 GMT.

String getId()	returns a string containing the unique identifier assigned to the session.
int getMaxInactiveInterval()	returns the maximum time interval, in seconds.
void invalidate()	destroy the session
boolean isNew()	returns true if the session is new else false

Complete Example demonstrating usage of HttpSession

index.html

```
<form method="post" action="Validate">
  User: <input type="text" name="uname " /><br/>
    <input type="submit" value="submit">
</form>
```

Validate.java

```
public class Validate extends HttpServlet {

    protected void doPost(request, response)
{
       // . . . .
        String name = request.getParameter("user");
                    //creating a session
           HttpSession session = request.getSession();
           session.setAttribute("user", uname);
           response.sendRedirect("Welcome");
        }
}
```

Welcome.java

```
public class Welcome extends HttpServlet {

   protected void doGet(request, response){
        // . . . .
       HttpSession session = request.getSession();
       String user = (String)session.getAttribute("user");
       out.println("Hello "+user);
   }
}
```

JSP

- **Java Server Page** technology is used to create dynamic web applications.
- **JSP** pages are easier to maintain then a **Servlet**.
- JSP pages are opposite of Servlets as a servlet adds HTML code inside Java code, while JSP adds Java code inside HTML using JSP tags.
- Everything a Servlet can do, a JSP page can also do it.
- JSP enables us to write HTML pages containing tags, inside which we can include powerful Java programs

Why JSP is preffered over servlets?

- JSP provides an **easier** way to code **dynamic web pages**.
- JSP **does not require** additional files like, java class files, **web.xml** etc
- Any **change** in the JSP code is handled by **Web Container**(Application server like tomcat), and doesn't require re-compilation.
- JSP pages can be directly accessed, and web.xml mapping is not required like in servlets.

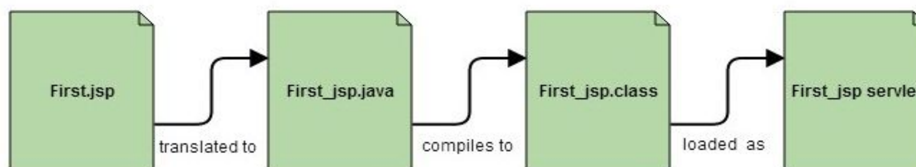
Advantage of JSP

- Easy to maintain and code.
- High Performance and Scalability.
- JSP is built on Java technology, so it is **platform independent**.

Life cycle of a JSP Page

The JSP pages follows these phases:

- Initialization (jspInit() method is invoked by the container).
- Request processing (_jspService() method is invoked by the container).
- Destroy (jspDestroy() method is invoked by the container).



In the end a JSP becomes a Servlet

- As depicted in the above diagram, JSP page is translated into servlet by the help of JSP translator. The JSP translator is a part of webserver that is responsible to translate the JSP page into servlet. After that Servlet page is compiled by the compiler and gets converted into the class file.
- **Web Container** translates JSP code into a **servlet class source(.java) file**, then compiles that into a java servlet class. In the third step, the servlet class bytecode is loaded using classloader. The Container then creates an instance of that servlet class.
- The initialized servlet can now service request. For each request the **Web Container** call the **_jspService()** method. When the Container removes the servlet instance from service, it calls the **jspDestroy()** method to perform any required clean up.

Do we need to follow directory structure to run a simple JSP ?

No, put jsp files in a folder directly and deploy that folder. It will be running fine. But if you are using bean class, Servlet or tld file then directory structure is required.

JSP Scripting Element

JSP Scripting element are written inside <% %> tags. These code inside <% %> tags are processed by the JSP engine during translation of the JSP page. Any other text in the JSP page is considered as HTML code or plain text.

There are five different types of scripting elements

1. **JSP scriptlet tag** A scriptlet tag is used to execute java source code in JSP.

<% java source code %>

In this example, we are displaying a welcome message.

```
<html>
<body>
<% out.print("welcome to jsp"); %>
</body>
</html>
```

2. **JSP Declaration Tag**

The **JSP declaration tag** is used *to declare variables, objects and methods*.
The code written inside the jsp declaration tag is placed outside the service() method of auto generated servlet.
So it doesn't get memory at each request.
<%! field or method declaration %>

Difference between JSP Scriptlet tag and Declaration tag

Jsp Scriptlet Tag	Jsp Declaration Tag
The jsp scriptlet tag can only declare variables not methods.	The jsp declaration tag can declare variables as well as methods.
The declaration of scriptlet tag is placed inside the _jspService() method.	The declaration of jsp declaration tag is placed outside the _jspService() method.

declaration tag with variable
In
index.jsp

```
<html>
<body>
<%! int data=50; %>
<%= "Value of the variable is:"+data %>
</body>
</html>
```

declaration tag that declares method
index.jsp

```
<html>
<body>
<%!
int cube(int n){
return n*n*n*;
}
%>
<%= "Cube of 3 is:"+cube(3) %>
```

3. **JSP Expression Tag**

Expression Tag is used to print out java language expression that is put between the tags. An expression tag can hold any java language expression that can be used as an argument to the **out.print()** method.
Syntax of Expression Tag
<%= *JavaExpression* %>

<%= (2*5) %>   //note no ; at end of statement.

4. **JSP directives**

The **jsp directives** are messages that tells the web container how to translate a JSP page into the corresponding servlet.
Syntax <%@ directive attribute="value" %>
There are three types of directives:

A. import directive
B. include directive
C. taglib directive

a. import

The import attribute is used to import class,interface or all the members of a package.It is similar to import keyword in java class or interface.
<%@ page import="java.util.Date" %>
Today is: <%= new Date() %>

b. Include Directive

The *include* directive tells the Web Container to copy everything in the included file and paste it into current JSP file. The include directive is used to include the contents of any resource it may be jsp file, html file or text file. Syntax of **include** directive is:
<%@ include file="filename.jsp" %>

c. JSP Taglib directive

The JavaServer Pages API allow you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior. The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides means for identifying the custom tags in your JSP page.
syntax <%@ taglib uri = "uri" prefix = "prefixOfTag" >
For example, suppose the **custlib** tag library contains a tag called **hello**. If you wanted to use the hello tag with a prefix of **mytag**, your tag would be **<mytag:hello>** and it will be used in your JSP file as follows

<%@ taglib uri = "http://www.example.com/custlib" prefix = "mytag" %>

<html>
  <mytag:hello/>
 </body>
</html>

5. **JSP Comments**

JSP comment marks text or statements that the JSP container should ignore.
syntax of the JSP comments **<%- - This is JSP comment - -%>**

Request String

- The query string contains the attribute & the value of the html form or JSP form, which sends with the POST /GET method to Servlet or JSP page for any request.

- A query string is the part of a URL which is attached to the end, after the file name. It begins with a question mark and usually includes information in pairs. The format is parameter=value, as in the following example:
  www.mediacollege.com/cgi-bin/myscript.cgi**?name="Umesh"**
- Query strings can contain multiple sets of parameters, separated by an ampersand (&) like so:

www.mediacollege.com/cgi-bin/myscript.cgi**?fname="Umesh"&lname="M"**

- to parse this info we use method of JSP **request** object
- we can easily get using **request.getParameter()** with name of parameter as argument, to get its value.

index.html

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

welcome.jsp

```
<%
String name=request.getParameter("uname");
out.print("welcome "+name);
%>
```

welcome.jsp

```
<html>
<body>
<%
        String
name=request.getParameter("uname");
        out.print("Welcome "+name);
        session.setAttribute("user",name);

<a href="second.jsp">second jsp page</a>

%>
</body>
</html>
```

Following are the JSP implicit object

Implicit Object	Description
request	The **HttpServletRequest** object associated with the request.
response	The **HttpServletRequest** object associated with the response that is sent back to the browser.
out	The **JspWriter** object associated with the output stream of the response.
session	The **HttpSession** object associated with the session for the given user of request.

Session implicit object (concept same as servlet)

In JSP, session is an implicit object of type HttpSession. The Java developer can use this object to set,get or remove attribute or to get session information.

Example of session implicit object

index.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go"><br/>
</form>
```

```
</body>
</html>
```

second.jsp

```
<html>
<body>
<%
      String name=(String)session.getAttribute("user");
         out.print("Hello "+name);
  %>
</body>
</html>
```

*<u>//other methods of **Session** what u used in servlets can be used here.</u>*

<u>Cookie (concept same as servlet)</u>

Step 1: Creating a Cookie object

Cookie cookie = new Cookie("key","value");

Step 2: Setting the maximum age

You use **setMaxAge** to specify how long (in seconds) the cookie should be valid. The following code will set up a cookie for 24 hours.

cookie.setMaxAge(60*60*24);

Step 3: Sending the Cookie into the HTTP response headers

You use **response.addCookie** to add cookies in the HTTP response header as follows

response.addCookie(cookie);

```
<%
  // Create cookies for first and last names.
  Cookie firstName = new Cookie("first_name", request.getParameter("first_name"));
  Cookie lastName = new Cookie("last_name", request.getParameter("last_name"));

  // Set expiry date after 24 Hrs for both the cookies.
  firstName.setMaxAge(60*60*24);
  lastName.setMaxAge(60*60*24);

  // Add both the cookies in the response header.
  response.addCookie( firstName );
  response.addCookie( lastName );
%>

<html>
  <body>
    p><b>First Name:</b>
        <%= request.getParameter("first_name")%>
      </p>
      <p><b>Last  Name:</b>
```

```
      <%= request.getParameter("last_name")%>
    </p>
  </body>
</html>
```

Reading Cookies with JSP

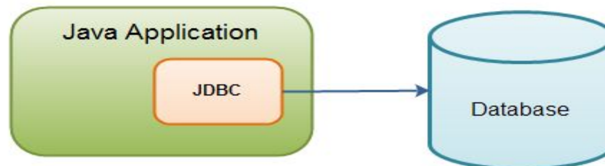
```
<html>
  <body>
    <%
      Cookie cookie = null;
      Cookie[] cookies = null;

      // Get an array of Cookies associated with the this domain
      cookies = request.getCookies();

      if( cookies != null ) {
        out.println("<h2> Found Cookies Name and Value</h2>");

        for (int i = 0; i < cookies.length; i++) {
          cookie = cookies[i];
          out.print("Name : " + cookie.getName( ) + ",  ");
          out.print("Value: " + cookie.getValue( )+" <br/>");
        }
      } else {
        out.println("<h2>No cookies founds</h2>");
      }
    %>
  </body>

</html>
```

*//other methods of **Cookie** what u used in servlets can be used here.*

MODULE -5

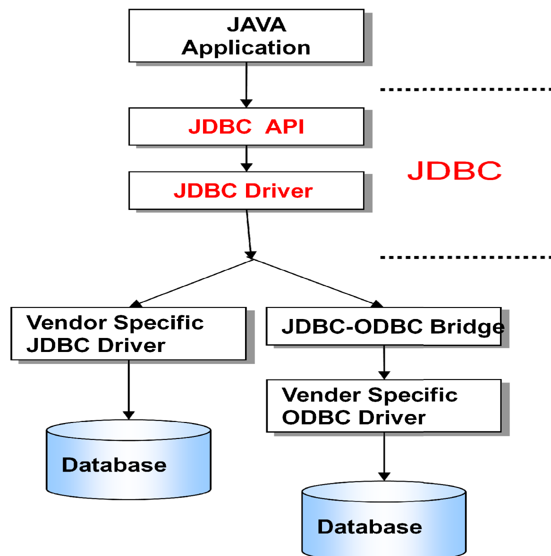
JDBC

- **Java Database Connectivity(JDBC)** is an **Application Programming Interface(API)** used to connect Java application with Database.



- JDBC is used to interact with various type of Database such as Oracle, MS Access, My SQL and SQL Server.
- JDBC can also be defined as the platform-independent interface between a relational database and Java programming.
- It allows java program to execute SQL statement and retrieve result from database.

JDBC Model



- JDBC consists of two parts:
  - JDBC API, a purely Java-based API
  - JDBC driver
    - Communicates with vendor-specific drivers
  - JDBC driver classified into 4 categories

JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

Advantages:

- easy to use.
- can be easily connected to any database.

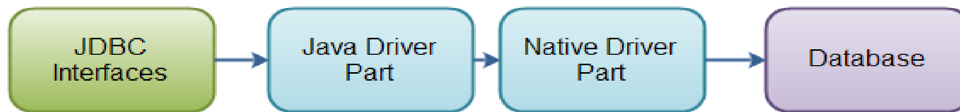
Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

---

2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.



Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

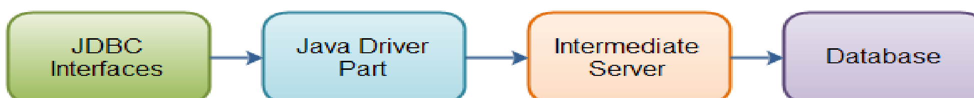
Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

---

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.



Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

---

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

Advantage:

- Better performance than all other drivers.

- No software is required at client side or server side.

Disadvantage:

- Drivers depends on the Database.

Various Database drivers

Database name	Driver name
MS Access	sun.jdbc.odbc.JdbcOdbcDriver
Oracle	oracle.jdbc.driver.OracleDriver
Microsoft SQL Server 2000	com.microsoft.sqlserver.jdbc.SQLServerDriver
MySQL	org.gjt.mm.mysql.Driver

JDBC Packages

JDBC API is contained in 2 packages.

- **import java.sql.*;**
  - contains core java data objects of JDBC API. It's a part of J2SE.
- **import javax.sql.* ;**
  - It extends java.sql and is in J2EE

5 Steps to connect to the database in java

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

- Register the driver class
- Creating connection
- Creating statement
- Executing queries
- Closing connection

---

1) Register the driver class

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Class.forName("driverClassName");

2) Create the connection object

The getConnection() method of DriverManager class is used to establish connection with the database.

Connection con = DriverManager.getConnection(url, user, password);

3) Create & Execute query

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Statement st = con.createStatement();

ResultSet rs = st.executeQuery(sql);

4) Process data returned form DBMS

```
while(rs.next()){
        System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

rs.close();

st.close();

con.close();

<u>complete example</u>

```
import java.sql.*;
import java.sql.*;
Class Dbconnection
```

```
{       public static void main(String args[])
  {     //Dynamically loads a driver class, for Oracle database
Class.forName("oracle.jdbc.driver.OracleDriver");

        //Establishes connection to database by obtaining a   Connection object
Connection con = DriverManager.getConnection( "jdbc:oracle:thin:@localhost:1521:XE",
"scott", "tiger");
        /*      jdbc- is API
                oracle- is DB name
                thin- is the driver
                localhost-is sever name on which oracle is running (can giv IP address)
                1521- is port number
                XE- is oracle service name      */
Statement statement = con.createStatement();
String sql = "select * from users";
ResultSet result = statement.executeQuery(sql);
while(result.next()) {
   String name = result.getString("name");
   long age = result.getInt("age");
   System.out.println(name);
   System.out.println(age);
}
result.close();
statement.close();
con.close();
}
```

Process results

When you execute an SQL query you get back a ResultSet. The ResultSet contains the result of your SQL query. The result is returned in rows with columns of data. You iterate the rows of the ResultSet like this:

```
while(result.next()) {
```

```
    String name = result.getString("name");
    long   age  = result.getLong  ("age");

}
```

The ResultSet.next() method moves to the next row in the ResultSet, if there are anymore rows. If there are anymore rows, it returns true. If there were no more rows, it will return false.

You can also pass an index of the column instead, like this:

```
 while(result.next()) {
   result.getString(1);
    result.getInt (2);
}
```

Statement Object

There are 3 types of statement objects to execute the sql query

Interfaces	Recommended Use
Statement	Use the for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

1. Statement object

- The **Statement interface** provides methods to execute queries with the database.
- The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

The important methods of Statement interface are as follows:

1) ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

Example show above

2) int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

Snippet

```
Statement stmt=con.createStatement();
// for insert
int result=stmt.executeUpdate("insert into emp values(33,'Irfan',50000)");
// for update
int result=stmt.executeUpdate("update empset name='Vimal',salary=10000 where id=33");
// for delete
int result=stmt.executeUpdate("delete from emp where id=33");
System.out.println(result+" records affected");
con.close();
```

3) boolean execute(String sql): is used to execute queries that may return multiple results.

```
boolean status = stmt.execute(anyquery);
    if(status){
      //query is a select query.
      ResultSet rs = stmt.getResultSet()
```

2. PreparedStatement object

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

Improves performance: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

```
// PreparedStatement to insert record
PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
        stmt.setInt(1,101);//1 specifies the first parameter in the query  stmt.setString(2,"Ratan");
        int i=stmt.executeUpdate();
```

System.out.println(i+" records inserted");

The **setXXX()** methods are used to supply values to the parameters

All of the **Statement object's** methods for interacting with the database execute(), executeQuery(), and executeUpdate() also work with the PreparedStatement object.

```
// PreparedStatement to update record
   PreparedStatement stmt=con.prepareStatement("update emp set name=? where id=?");
   stmt.setString(1," Ratan ");//1 specifies the first parameter in the query i.e. name
   stmt.setInt(2,101);
   int i=stmt.executeUpdate();
   System.out.println(i+" records updated");
```

```
// PreparedStatement to delete record
   PreparedStatement stmt=con.prepareStatement("delete from emp where id=?");
   stmt.setInt(1,101);
    int i=stmt.executeUpdate();
   System.out.println(i+" records deleted");
```

3. CallableStatement Objects

CallableStatement interface is used to call the **stored procedures and functions**.

Suppose you need the get the age of the employee based on the date of birth, you may create a function that receives date as the input and returns age of the employee as the output.

Snippet

```
String SQL = "{call getEmpName (?, ?)}"; // stored procedure called
  cs = conn.prepareCall (SQL);
cs.setInt(100);
// resisterOutParameter() used to register OUT type used by stored procedure
cs.resisterOutParameter(2, VARCHAR);
cs.execute();
String Name = cs.getString(1);
Cs.close();
```

Oracle stored procedure −

```
CREATE OR REPLACE PROCEDURE getEmpName
  (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
  SELECT first INTO EMP_FIRST
  FROM Employees
  WHERE ID = EMP_ID;
END;
```

Three types of parameters exist: IN, OUT, and INOUT.

Parameter	Description
IN	Data that needs to be passed to stored procedure. values to IN parameters with the setXXX() methods.
OUT	contains value supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values.

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

Method Name	Description
first()	Moves the cursor to the first row
last()	Moves the cursor to the last row.
previous()	Moves the cursor to the previous row. This method returns false if the previous row is off the result set
next()	Moves the cursor to the next row. This method returns false if there are no more rows in the result set

absolute(int row)	Moves the cursor to the specified row
relative(int row)	Moves the cursor the given number of rows from where it currently is pointing.
getRow()	Returns the row number that the cursor is pointing to.
beforeFirst()	Moves the cursor to just before the first row
afterLast()	Moves the cursor to just after the last row

When you create a ResultSet there are three attributes you can set. These are:

1. **Type**
    1. **ResultSet.TYPE_FORWARD_ONLY** (default type)- TYPE_FORWARD_ONLY means that the ResultSet can only be navigated forward
    2. **ResultSet.TYPE_SCROLL_INSENSITIVE**- TYPE_SCROLL_INSENSITIVE means that the ResultSet can be navigated (scrolled) both forward and backwards. The ResultSet is insensitive to changes while the ResultSet is open. That is, if a record in the ResultSet is changed in the database by another thread or process, it will not be reflected in already opened ResulsSet's of this type.
    3. **ResultSet.TYPE_SCROLL_SENSITIVE**- means that the ResultSet can be navigated (scrolled) both forward and backwards. The ResultSet is sensitive to changes in the underlying data source while the ResultSet is open.
2. **Concurrency** determines whether the ResultSet can be updated, or only read.
    1. **ResultSet.CONCUR_READ_ONLY**- means that the ResultSet can only be read.

    2. **ResultSet.CONCUR_UPDATABLE**- means that the ResultSet can be both read and updated.

//Inserting Rows into a ResultSet u have updateXXX( )
rs.updateString (1, "raj");
rs.updateInt (2, 55);
rs.**insertRow(); // call this method**

//updating Rows into a ResultSet u have updateXXX( )

Updates the current row by updating the corresponding row in the database.

rs.updateString ("name" , "ram");

rs.updateInt ("age" , 55);

rs.**updateRow();// call this method**

//Delete Rows from a ResultSet

Deletes the current row from the database

rs.**deleteRow(3);** //delete current row

3. **Holdability-** determines if a ResultSet is closed when the commit() method of the underlying connection is called.
    1. **ResultSet.CLOSE_CURSORS_OVER_COMMIT-** means that all ResultSet instances are closed when connection.commit() method is called on the connection that created the ResultSet.
    2. **ResultSet.HOLD_CURSORS_OVER_COMMIT-** means that the ResultSet is kept open when the connection.commit() method is called on the connection that created the ResultSet.

The HOLD_CURSORS_OVER_COMMIT holdability might be useful if you use the ResultSet to update values in the database. Thus, you can open a ResultSet, update rows in it, call connection.commit() and still keep the same ResultSet open for future transactions on the same rows.

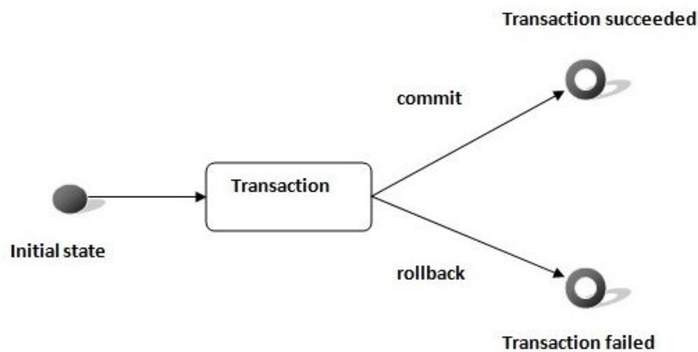
- U can set resultset attribute for Statement or PreparedStatement, like this:

```
Statement statement = connection.createStatement(
   ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY,
   ResultSet.CLOSE_CURSORS_OVER_COMMIT
  );
```

```
PreparedStatement statement = connection.prepareStatement(sql,
   ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY,
   ResultSet.CLOSE_CURSORS_OVER_COMMIT
  );
```

Transaction

- A transaction is a set of actions to be carried out as a single, atomic action. Either all of the actions are carried out, or none of them are.
- *Advantage **is** fast performance It makes the performance fast because database is hit at the time of commit.*



In JDBC, **Connection interface** provides methods to manage transaction.

Method	Description
void setAutoCommit(boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

```
try{
   //Assume a valid connection object conn
   conn.setAutoCommit(false);
   Statement stmt = conn.createStatement();

   String SQL = "INSERT INTO Employees  " +
               "VALUES (106, 20, 'Rita', 'Tez')";
   stmt.executeUpdate(SQL);
   //Submit a malformed SQL statement that breaks
   String SQL = "INSERTED IN Employees  " +
               "VALUES (107, 22, 'Sita', 'Singh')";
```

```
  stmt.executeUpdate(SQL);
  // If there is no error.
  conn.commit();
}catch(SQLException se){
  // If there is any error.
  conn.rollback();
}
```

Savepoints

- When you set a savepoint you define a logical rollback point within a transaction. If an error occurs past a savepoint, you can use the rollback method to undo either all the changes or only the changes made after the savepoint**.**

The Connection object has two new methods that help you manage savepoints −

- **setSavepoint(String savepointName):** Defines a new savepoint. It also returns a Savepoint object.
- **releaseSavepoint(Savepoint savepointName):** Deletes a savepoint. Notice that it requires a Savepoint object as a parameter. This object is usually a savepoint generated by the setSavepoint() method.

```
try{
  //Assume a valid connection object conn
  conn.setAutoCommit(false);
  Statement stmt = conn.createStatement();

  //set a Savepoint
  Savepoint savepoint1 = conn.setSavepoint("Savepoint1");
  String SQL = "INSERT INTO Employees " +  "VALUES (106, 20, 'Rita', 'Tez')";
  stmt.executeUpdate(SQL);
  //Submit a malformed SQL statement that breaks
  String SQL = "INSERTED IN Employees " +  "VALUES (107, 22, 'Sita', 'Tez')";
  stmt.executeUpdate(SQL);
```

```
   // If there is no error, commit the changes.
   conn.commit();

}catch(SQLException se){
   // If there is any error.
   conn.rollback(savepoint1);
}
```

Batch Processing in JDBC

Instead of executing a single query, we can execute a batch (group) of queries. It makes the performance fast.

Methods of Statement interface

The required methods for batch processing are given below:

Method	Description
void addBatch(String query)	It adds query into batch.
int[] executeBatch()	It executes the batch of queries.
clearBatch()	Clears the batch

```
Statement stmt=con.createStatement();
stmt.addBatch("insert into user values(190,'abhi',40000)");
stmt.addBatch("insert into user values(191,'umesh',50000)");
int[] count = stmt.executeBatch();
// u can get number of sql stmt that was executed by count[] array.
```

MetaData

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

methods of DatabaseMetaData interface

- **String getDriverName():** it returns the name of the JDBC driver.
- **String getDriverVersion():** it returns the version number of the JDBC driver.
- **String getUserName():** it returns the username of the database.
- **String getDatabaseProductName():** it returns the product name of the database.
- **String getDatabaseProductVersion():** it returns the product version of the database.
- **String getSchemas()**
- String getPrimaryKeys()
- String getProcedures()
- String getTables()

```
DatabaseMetaData dbmd=con.getMetaData();
 System.out.println("Driver Name: "+dbmd.getDriverName());
System.out.println("Driver Version: "+dbmd.getDriverVersion());
System.out.println("UserName: "+dbmd.getUserName());
System.out.println("Database Product Name: "+dbmd.getDatabaseProductName());
System.out.println("Database Product
```

ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

methods of ResultSetMetaData interface

Method	Description
public int getColumnCount()	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)	it returns the column name of the specified column index.
public String getColumnTypeName(int	it returns the column type name for the specified index.

index)	
public String getTableName(int index)	it returns the table name for the specified column index.

Exceptions

JDBC methods throws 3 types of exceptions

1. SQLException
2. SQLWarnings
3. DataTruncation.

1. **SQLException Methods** - An SQLException can occur both in the driver and the database.

Method	Description
getErrorCode( )	Gets the error number associated with the exception.
getMessage( )	Gets the JDBC driver's error message for an error, handled by the driver or gets the Oracle error number and message for a database error.
getNextException( )	Gets the next Exception object in the exception chain.

2. **SQLWarnings** - An exception that provides information on database access warnings.

getWarnings()-Retrieves the first warning reported by calls on this Connection object.

getNextWarning()-Retrieves subsequent warnings.

3. **DataTruncation** – this exception is thrown when a data values is unexpectedly truncated