The Need for Structures :-

There are cases where the value of one variable depends upon that of another variable. Take the example of date. A date can be programmatically represented in C by three different integer variables taken together. Say,
int d,m,y; //three integers for representing dates

Here ‘d’, ‘m’, and ‘y’ represent the day of the month, the month, and the year, respectively.
Observe carefully. Although these three variables are not grouped together in the code, they actually belong to the same group. The value of one variable may influence the value of the other two. In order to understand this clearly, consider a function next_day () that accepts the addresses of the three integers that represent a date and changes their values to represent
the next day.

The prototype of this function will be
void next_day (**int *, int *, int ***); //function to calculate the next day
Suppose,
d=1;
m=1;
y=2002; //1st January, 2002
Now, if we write
next_day(&d,&m, &y);
‘d’ will become 2, ‘m’ will remain 1, and ‘y’ will remain 2002.
But if
d=28;
m=2;
y=1999; //28th February, 1999
As you can see, ‘d’, ‘m’, and ‘y’ actually belong to the same group. A change in the value of one may change the value of the other two. But there is no language construct that actually places them in the same group. Thus, members of the wrong group may be accidentally sent to the function.

Let us try arrays to solve the problem. Suppose the next_day() function accepts an array as a parameter. Its prototype will be
void next_day(**int ***);

notes4free.in
Source diginotes.in

Let us declare date as an array of three integers.

```
int  date[3];
date[0]=28;
date[1]=2;
date[2]=1999; //28th February, 1999
```

Now, let us call the function as follows:

```
next_day(date);
```

The values of 'date[0]', 'date[1]', and 'date[2]' will be correctly set to 1, 3, and 1999, respectively. Although this method seems to work, it certainly appears unconvincing. After all any integer array can be passed to the function, even if it does not necessarily represent a date. There is no data type of date itself. Moreover, this solution of arrays will not work if the variables are not of the same type. The solution to this problem is to create a data type called date itself using structures

```
struct date //a structure to represent dates
{
int d, m, y;
};
```

Now, the next_day() function will accept the address of a variable of the structure date
as a parameter. Accordingly, its prototype will be as follows:

```
void next_day(struct date *);

struct date d1;
d1.d=28;
d1.m=2;
d1.y=1999;
next_day(&d1);
```

'd1.d', 'd1.m', and 'd1.y' will be correctly set to 1, 3, and 1999, respectively. Since the function takes the address of an entire structure variable as a parameter at a time, there is no chance of variables of the different groups being sent to the function.

Structure is a programming construct in C that allows us to put together variables that should be together.

Library programmers use structures to create new data types. Application programs and other library programs use these new data types by declaring variables of this data type.

Finally, they use the resultant value of the passed variable further as per requirements

notes4free.in
Source diginotes.in

printf("The next day is: %d/%d/%d\n", d1.d, d1.m, d1.y);

Output

The next day is: 1/3/1999

Creating a New Data Type Using Structures

Creation of a new data type using structures is loosely a three-step process that is executed
by the library programmer.

Step 1: Put the structure definition and the prototypes of the associated functions in a header
file,

Header file containing definition of a structure variable and prototypes of its associated functions.

```
/*Beginning of date.h*/
/*This file contains the structure definition and
prototypes of its associated functions*/

struct date
{
int d,m,y;
};
void next_day(struct date *); //get the next date
void get_sys_date(struct date *); //get the current
//system date
//Prototypes of other useful and relevant functions to work upon variables of the
date structure
/*End of date.h*/ code and create a library.
```


notes4free.in
Source diginotes.in

Defining the associated functions of a structure

```
//Beginning of date.c
//This file contains the definitions of the associated functions
#include "date.h"
void next_day(struct date * p)
{
//calculate the date that immediately follows the one
//represented by *p and set it to *p.
}
void get_sys_date(struct date * p)


{
//determine the current system date and set it to *p
}
//Definitions of other useful and relevant functions to work upon variables
of the date structure

/*End of date.c*/
```

Step 3: Provide the header file and the library, in whatever media, to other programmers who
want to use this new data type. Creation of a structure and creation of its associated functions are two separate steps that together constitute one complete process.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

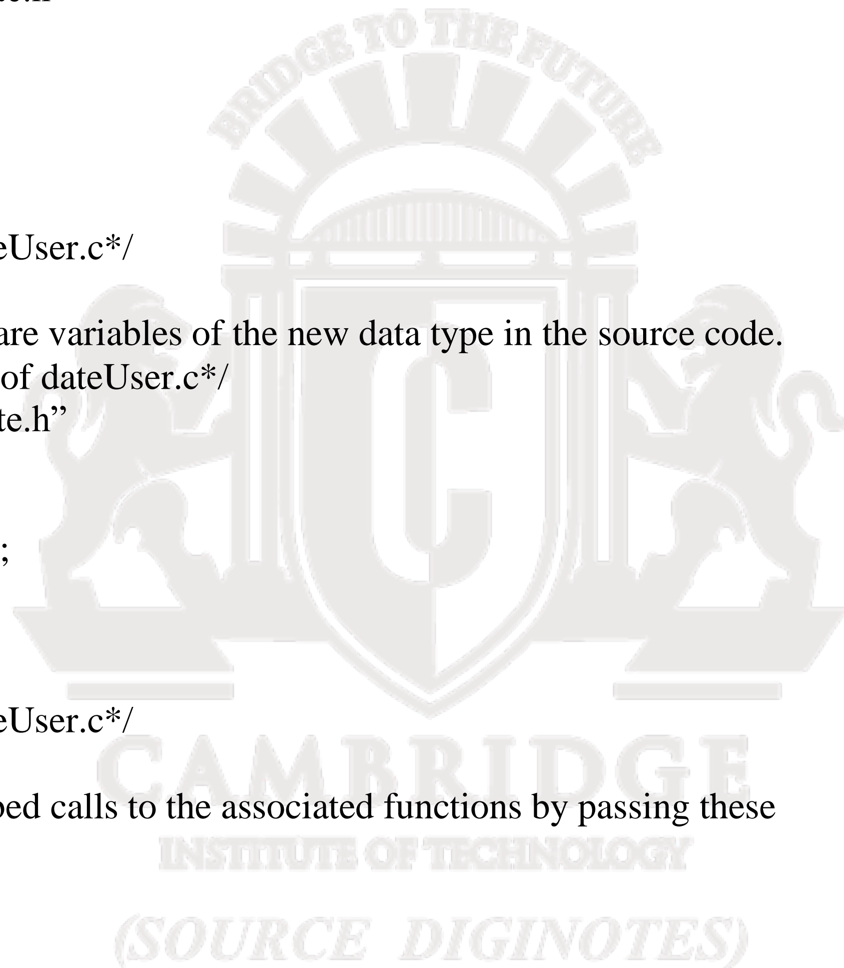
Using Structures in Application Programs

The steps to use this new data type are as follows:
Step 1: Include the header file provided by the library programmer in the source code.

```
/*Beginning of dateUser.c*/

#include"date.h"
void main( )
{
. . . .
. . . .
}
/*End of dateUser.c*/
```

Step 2: Declare variables of the new data type in the source code.

```
/*Beginning of dateUser.c*/
#include"date.h"
void main( )
{
struct date d;
. . . .
. . . .
}
/*End of dateUser.c*/
```



Step 3: , embed calls to the associated functions by passing these

variables in the source code.

Using a structure in an application program

```
/*Beginning of dateUser.c*/
#include"date.h"
void main()
{
struct date d;
d.d=28;
d.m =2;
d.y=1999;
next_day(&d);
. . . .
. . . .
}
/*End of dateUser.c*/
```

Step 4: Compile the source code to get the object file.

Step 5: Link the object file with the library provided by the library programmer to get the
executable or another library.

Program

```
//date.h
struct date
{
	int d,m,y;
};
void nextdate(struct date *);
void getdate(struct date *);
// end of date.h

//program.cpp

#include<stdio.h>
#include "date.h"

void nextdate(struct date *p)
{
	p->d++;
}
void getdate(struct date *p)
```


notes4free.in
Source diginotes.in

```
{
        printf("date is %d/%d/%d\n",p->d,p->m,p->y);
}
void main()
{
        struct date dat;
        dat.d=2;
        dat.m=3;
        dat.y=2001;
        nextdate(&dat);
        getdate(&dat);


}
```

Procedural oriented programming (pop):-

A program in a procedural language is a list of instruction where each statement tells the computer to do some task. It focuses on procedure (function) & algorithm is needed to perform the derived computation.

When program become larger, it is divided into function & each function has clearly defined purpose. Dividing the program into functions & module is one of the cornerstones of structured programming.

Characteristics of Procedural oriented programming:-

- It focuses on function rather than data.
- It takes a problem as a sequence of things to be done such as reading, calculating and printing. Hence, a number of functions are written to solve a problem.
- A program is divided into a number of functions and each function has clearly defined purpose.
- Most of the functions share global data.
- Data moves openly around the system from function to function.

Drawback of Procedural oriented programming (structured programming):-

- It emphasis on doing things(functionality). Data is not given important status even through data is the reason for the existence of the program.

notes4free.in

Source diginotes.in

- Since every function has complete access to the global variables, the new programmer can corrupt the data accidentally by creating function. Similarly, if new data is to be added, all the function needed to be modified to access the data.

Object-Oriented Programming Systems

In OOPS, we try to model real-world objects.
But, what are real-world objects?
Most real world objects have internal parts and interfaces that enable us to operate them. These interfaces perfectly manipulate the internal parts of the objects. They also have the exclusive rights to do so.

In object-oriented programming languages like C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an object.

A class is an extended concept similar to that of structure in C programming language; this class describes the data properties alone.

In C++ programming language, a class describes both the properties (data) and behaviors (functions) of objects.

Classes are not objects, but they are used to instantiate objects.

Encapsulation:

Encapsulation is an object-oriented programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse

Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. Typically, only the object's own methods can directly inspect or manipulate its fields.

notes4free.in
Source diginotes.in

Data abstraction :

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this −

```
#include <iostream>

using namespace std;


int main() {

   cout << "Hello C++" <<endl;

   return 0;

}
```

Here, you don't need to understand how **cout** displays the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change.

notes4free.in
Source diginotes.in

Inheritance

Inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support.

In classical inheritance where objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called base classes, superclasses, parent classes or ancestor classes.

The resulting classes are known as derived classes, subclasses or child classes

Polymorphism:

Polymorphism means one name, many forms. Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality.

There are 2 basic types of polymorphism.

Overridding, also called run-time polymorphism. For method overloading, the compiler determines which method will be executed, and this decision is made when the code gets compiled.

Overloading, which is referred to as compile-time polymorphism. Method will be used for method overriding is determined at runtime based on the dynamic type of an object.

Comparison of C and C++

C++ is the extension of C language, ie it is superset of C language this means that C++ compiler can compile programs written in C language(vice versa is not possible).

The syntax of decision making looping constructs and structure remains same as that of C language.

The main differences between C++ over C language

- The keyword "class" has been used instead of "struct".

notes4free.in

Source diginotes.in

- The C++ uses access specifiers (public, private, protected) for providing security, but this option is not there in C.
- Apart from the data members it also has one special function called "constructor".it has same name as that of class but no return type and access specifier.

Console Input /Output in C++

Console Output

The output functions in C language, such as printf(), can be included in C++ programs
because they are anyway defined in the standard library. However, there are some more ways of outputting to the console in C++. Let us consider an example

```
/*Beginning of cout.cpp*/
#include<iostream.h>
void main()
{
int x;
x=10;
cout<<x; //outputting to the console
}
/*End of cout.cpp*/
```

Output
10
Syntax
cout<<variable

cout (pronounce see-out) is actually an object of the class ostream_withassign .
It stands as an alias for the **c**onsole **out**put device, that is, the monitor.

The << symbol, originally the left shift operator, has had its definition extended in C++.In the given context, it operates as the insertion operator. It is a binary operator. It takes two operands.

notes4free.in
Source diginotes.in

The operand on its left must be some object of the ostream class. The operand on its right must be a value of some fundamental data type.

The value on the right side of the **insertion operator** is ‘inserted’ (hence the name) into the stream headed towards the device associated with the object on the left. Consequently, the value of ‘x’ is displayed on the monitor.

The file iostream.h needs to be included in the source code to ensure successful compilation because the object cout and the insertion operator have been declared in that file.

Cascading the insertion operator

```
#include<iostream.h>
 int main()
{
int x;
float y;
x=10;
y=12.5;
cout<<”the value of x=”<<x<<endl<<”the value of y=”<<y<<endl;
}
```

Output

The value of x=10

The value of y=12.5

Console input

The input function in C language such as scanf() can be included in C++ program because it is defined in standard library. However,we have some more ways to input in C++ that is “cin”.

Cin (see-in) is actually an object of the class ‘istream_withassign’.it stands as an alias for standard input ie keyboard.

Syntax

cin >>variable.

“>>” originally the right shift operator, has had its definition extended in C++ as the **‘extraction’ operator**.

It is a binary operator takes two operands.

the operand on its left must be some object of ‘istream_withassign’ class.

notes4free.in
Source diginotes.in

The operand on right side is variable, this operator will extracted from the stream originating from the keyboard device and stores in the variable on right hand side.

```
/*Beginning of cin.cpp*/
#include<iostream.h>
void main()
{
int iVar;
char cVar;
float fVar;
cout<<"Enter a whole number: ";
cin>>iVar;
cout<<"Enter a character: ";
cin>>cVar;
cout<<"Enter a real number: ";
cin>>fVar;
cout<<"You entered: "<<iVar<<" "<<cVar<<" "<<fVar;
}
/*End of cin.cpp*/
```

Output

Enter a whole number: **10**<*enter*>
Enter a character: **x**<*enter*>
Enter a real number: **2.3**<*enter*>
You entered: 10 x 2.3

Cascading the extraction operator

```
#include<iostream.h>
void main()
{
int x,y;

cout<<"enter the two values";

cin>>x>>y;

cout<<"the two values are"<<x<,"and<<y;

}
```


notes4free.in
Source diginotes.in

Variables in C++

We can declare the variable anywhere inside the function and not necessarily at very beginning.

Example

```
#include<iostram.h>
Void main()
{
int x=10;
cout<<"x="<<x;
int y=x;
cout<<"y="<<y;
int sum=x+y;
cout<<"sum="<<sum;
}
```

Reference Variables in C++

First, let us understand the basics. How does the operating system (OS) display the value of variables? How are assignment operations such as 'x=y' executed during run time?
The OS maintains the addresses of each variable as it allocates memory for them during run time. In order to access the value of a variable, the OS first finds the address of the variable and then transfers control to the byte whose address matches that of the variable.
Suppose the following statement is executed ('x' and 'y' are integer type variables).
x=y;
The steps followed are:
1. The OS first finds the address of 'y'.
2. The OS transfers control to the byte whose address matches this address.
3. The OS reads the value from the block of four bytes that starts with this byte (most
C++

(SOURCE DIGINOTES)

compilers cause integer-type variables to occupy four bytes during run time and we will
accept this value for our purpose).
4. The OS pushes the read value into a temporary stack.
5. The OS finds the address of ‘x’.
6. The OS transfers control to the byte whose address matches this address.
7. The OS copies the value from the stack, where it had put it earlier, into the block of four
 bytes that starts with the byte whose address it has found above (address of ‘x’).

A reference variable is nothing but a reference for an existing variable. It shares the memory location with an existing variable.

The syntax for declaring a reference variable is as follows:
<data-type> & <ref-var-name>=<existing-var-name>;

For example, if ‘x’ is an existing integer-type variable and we want to declare iRef as a reference to it the statement is as follows:
int & iRef=x;
iRef is a reference to ‘x’. This means that although iRef and ‘x’ have separate entries in the OS, their addresses are actually the same!
Thus, a change in the value of ‘x’ will naturally reflect in iRef and vice versa

Example :

```
#include<iostream.h>
void main()
{
int x;
x=10;
cout<<x<<endl;
int & iRef=x; //iRef is a reference to x
iRef=20; //same as x=10;
cout<<x<<endl;
```


notes4free.in
Source diginotes.in

```
x++; //same as iRef++;
cout<<iRef<<endl;
}
```

Output

10
20
21

Reading the value of a reference variable

```
#include<iostream.h>
void main()
{
int x,y;
x=10;
int & iRef=x;
y=iRef; //same as y=x;
cout<<y<<endl;
y++; //x and iRef unchanged
cout<<x<<endl<<iRef<<endl<<y<<endl;
}
```

Output

10
10
10
11

Passing by reference

```
#include<iostream.h>
void increment(int &); //formal argument is a reference
//to the passed parameter
void main()
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in
Source diginotes.in

```
{
int x;
x=10;
increment(x);
cout<<x<<endl;
}
void increment(int & r)
{
r++; //same as x++;
}
```

Output

11

Function Prototyping

A prototype describes the function's interface the compiler.it tells to the compiler the return type of the function and number and type of the formal parameters of the function.

Syntax

return_type function_name(argument_list);

example

int add(int,int);

this tells to compiler that the return type of add function is int and it takes two parameters of type int.

providing names to the formal parameter is optional.

Example

```
#include<iostream.h>
int add(int,int);
void main()
{
int x,y,z;
cout<<"enter the value two numbers";
cin>>x>>y;
z=add(x,y);
cout<<"sum="<<z;
```

notes4free.in

Source diginotes.in

```
}
int add(int a,int b)
{
return(a+b);
}
```

Output

Enter the two numbers 10 20

Sum=30

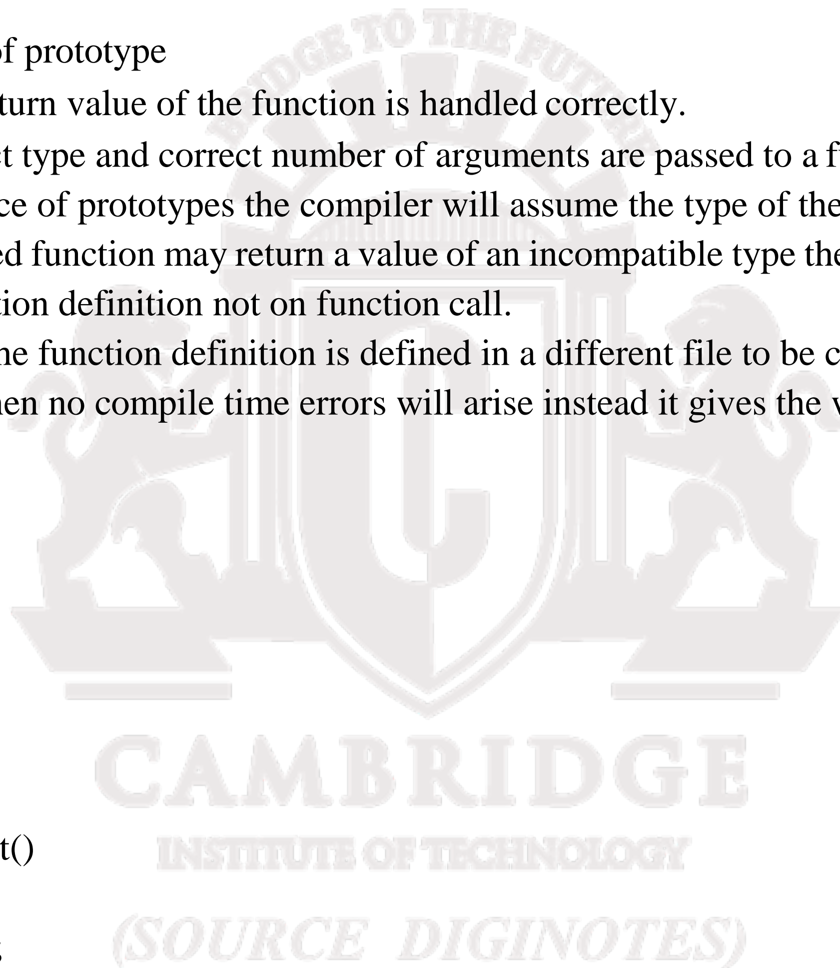
Importance of prototype

- The return value of the function is handled correctly.
- Correct type and correct number of arguments are passed to a function

In the absence of prototypes the compiler will assume the type of the returned type value, if called function may return a value of an incompatible type then it shows the error in function definition not on function call.

However if the function definition is defined in a different file to be compiled separately, then no compile time errors will arise instead it gives the wrong result.

Example



CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)


```
//def.c
struct abc
{
char a;
int b;
}
struct abc test()
{
struct abc a1;
a1.a='x';
a1.b=15;
return a1;
}
//end od def.c
//driver.c
void main()
{
```


notes4free.in
Source diginotes.in

```
int x;
x=test();
printf("%d",x);
}
```
Output

1688

Since the C++ compiler necessitates function prototyping ,it will report an error against the “function call “ not on function definition thus providing guarantees protection from errors arising out of incorrect function calls.

Function prototyping produces automatic type conversion wherever appropriate

Suppose if the compiler do not enforces prototyping and a function except the integer value but we passed a double value, the the first 4 bytes of data is extracted from 8 bytes of data which is undesirable.

However, C++ compiler will convert the double value to integer if we give the function prototype (because it already knows that the function parameter is integer)

But the C++ compiler cannot convert from a structure type to integer type.

in absence of function prototype is it possible for the compiler to simply scan the rest of the source code and find out how function has been defined.

Answer is “No”

Why because

- **It is inefficient:** the compiler will have to suspend the compilation of the line containing the function call and search the rest of the file.
- Most of the times the function definition is not contained in the file where it is called.it is usually contained in a library.

Such type of checking is known as static type checking

Function overloading

C++ allows two or more functions to have the same name.

It is possible only when the two or more functions have different signature

Signature means here they should have different type or different number of parameters.

Depending upon the type and number of parameters that are passed to the function call the compiler will decide which of the function definition to be executed.

notes4free.in
Source diginotes.in

Example

```
#include<iostream>
using namespace std;
int add(int a,int b);
int add(int a,int b,int c);


int main()
{
    int res=add(5,6,7);
    cout<<res<<endl;
    res=add(4,6);
    cout<<res<<endl;
}
int add(int a,int b)
{
    return (a+b);
}
int add(int a,int b,int c)
{
    return(a+b+c);
}
```

Function prototyping is important for function overloading because the compiler is able to not only restrict the number of ways in which the function can be called but also support more than one way in which the function can be called.

Function overloading is also known as function polymorphism because just like in the real world where an entity exists in more than one form with different meanings

Since which function definition should execute is decided by the compiler during the function call. So the function overloading is called static polymorphism.

notes4free.in
Source diginotes.in

Default values for formal argument of functions

It is possible to specify default values for some or all the formal arguments of a function.

If no value is passed during the function call the default value specified is passed.

If all parameter values are passed in normal fashion the default value is ignored.

Example

```
#include<iostream>
using namespace std;
int add(int a,int b,int c=0);


int main()
{
    int res=add(5,6);
    cout<<res<<endl;
    res=add(4,6,5);
    cout<<res<<endl;
}
int add(int a,int b,int c)
{
    return(a+b+c);
}
```

Output

11

15

- Default values can be assigned to more than one argument starting from the rightmost argument

Example : int add(int a,int b=0,int c=0);

Program

```
#include<iostream>
using namespace std;
int add(int a,int b=0,int c=0);
```

notes4free.in

Source diginotes.in

```
int main()
{
    int res=add(4,6,5);
    cout<<res<<endl;
    res=add(4,6);
    cout<<res<<endl;
    res=add(4);
    cout<<res<<endl;


}
int add(int a,int b,int c)
{
    return(a+b+c);
}
```

Output

15

10

4

- **`int add(int a,int b=0,int c);**

this is not possible compiler will throw an error because the third value is missing

- **Default values must be specified in function prototype alone.**

  If the function definition is given after the function call, the compiler will not know the default value if it is given in function definition, so it will throw an error.

  Sometimes the function definition will be different file, if we try to give default value in function prototype as well as function definition, the compiler will think that we are passing two different values for the same argument, so it will throw an error.

  For these two reasons we must specify the default values in function prototype alone.

notes4free.in

Source diginotes.in

- If two or more functions are overloaded with default value of same type of parameters it will lead to ambiguity error.
  Ex: int add(int ,int ,int =0);
  int add(int,int);
- We can assign any data type values as default value
  double add(double,double=3.2);
  void print(char='a');

Inline function

When a program started to execute the operating system loads each instructions in to the memory.
If there is any looping or branch out,the control skips over instruction or jumps backward or forward as needed.
When a program reaches the function call,it stores the memory address of the instruction immediately following the function call and jumps to the line where the function is defined.
After completion of function statements it jumps back to the instruction whose address is it has saved earlier.
There are overhead involved in

- Making the control jump back and forth and
- Storing the address of the instruction to which the control should jump after function terminates

To overcome this overhead C++ provides the solution "inline".

An inline function is function whose compiled code 'in line' with the rest of the program
ie the compiler replaces the function call with the corresponding function code
for specifying an inline function, we must

- Prefix the definition of the function with the inline keyword
- Define the function before all the function calls it.

```
#include<iostream>
using namespace std;
```


notes4free.in
Source diginotes.in

```
inline double cube(double x);


int main()
{
        double res=cube(5);//compiler replaces the function definition of cube
        cout<<res<<endl;
        res=cube(1.1 );//compiler replaces the function definition of cube
        cout<<res<<endl;
}

double cube(double x)
{
        return(x*x*x);
}
```

Output
125
1.331

However under some circumstances the compiler despite our indications may not expand the function inline instead it will run as ordinary function call for:

- If the function is recursive
- There are looping constructs in the function.
- There are static variables in the function

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in
Source diginotes.in

<u>**Class and structure**</u>

Introduction to Classes and Objects

Classes are to C++ what structures are to C. Both provide the library programmer a means to create new data types.

First, we must notice that functions have also been defined within the scope of the structure definition. This means that not only the member data of the structure can be accessed through the variables of the structures but also the member functions can be invoked. The struct keyword has actually been redefined in C++.

Member functions are invoked in much the same way as member data are accessed, that is, by using the variable-to-member access operator. In a member function, one can refer directly to members of the object for which the member function is invoked.

However, in this example, note that the member data of structure variables can still be accessed directly. The following line of code illustrates this, d1.ifeet=2; //legal!!

```
#include<iostream>
using namespace std;
struct dist
{
        int ifeet;
        float finch;

                void setfeet(int x)
                  {
                        ifeet=x;
                        }
                int getfeet()
                {
                        return ifeet;
                }
                void setinch(float y)
                {
                        finch=y;
                }
                float getinch()
                {
                        return finch;
                }

};
int main()
{
        dist d1;
```


notes4free.in
Source diginotes.in

```
	d1.setfeet(29);
	d1.setinch(3.8);
	cout<<d1.getfeet()<<endl<<d1.getinch();

}
```

2.1.1 Private and Public Members

What is the advantage of having member functions also in structures? We have put together the data and functions that work upon the data but we have not been able to give exclusive rights to these functions to work upon the data. Problems in code debugging can still arise as before. Specifying member functions as public but member data as private obtains the advantage.

```
struct distance
{
private:
	int iFeet;
	float fInches;
public:
	void setFeet(int x)
	{
		iFeet=x; //LEGAL: private member accessed by
		  //member function
	}
	int getFeet()
	{
		return iFeet;
	}
	void setInches(float y)
	{
		fInches=y;
	}
	float getInches()
	{
		return fInches;
	}
};
void main()
{
	distance d1,d2;
	d1.setFeet(2);
	d1.setInches(2.2);
	d1.iFeet++; //ERROR!!: private member accessed by
			//non-member function
	cout<<d1.getFeet()<<" "<<d1.getInches()<<endl;
```

}

new keywords, private and public have been introduced in the definition of the structure. Their presence in the foregoing example tells the compiler that iFeet and fInches are private data members of variables of the structure Distance and the member functions are public. Thus, values of iFeet and fInches of each variable of the structure Distance can be accessed/ modified only through member functions of the structure and not by any non-member function .
As we can observe , the compiler refuses to compile the line in which a private member of a structure variable is accessed from a non-member function .
The keywords private and public are also known as access modifiers or access specifiers because they control the access to the members of structures.

C++ introduces a new keyword class as a substitute for the keyword struct. *In a structure, members are public by default.*

```
struct Distance
{
private:
        int iFeet;
        float fInches;
public:
        void setFeet(int x)
        {
                iFeet=x;
        }
        int getFeet()
        {
                return iFeet;
        }
        void setInches(float y)
        {
                fInches=y;
        }
        float getInches()
        {
                return fInches;
        }
};
```

can also be written as
struct Distance

notes4free.in

Source diginotes.in

```
{
        void setFeet(int x) //public by default
        {
                iFeet=x;
        }
        int getFeet() //public by default
        {
                return iFeet;
        }
        void setInches(float y) //public by default
        {
                fInches=y;
        }
        float getInches() //public by default
        {
                return fInches;
        }
        private:
                int iFeet;
                float fInches;
};
```

Class members are private by default

```
class Distance
{
        int iFeet; //private by default
        float fInches; //private by default
public:
        void setFeet(int x)
        {
                iFeet=x;
        }
        int getFeet()
        {
                return iFeet;
        }
        void setInches(float y)
        {
                fInches=y;
        }
        float getInches()
        {
                return fInches;
        }
};
```

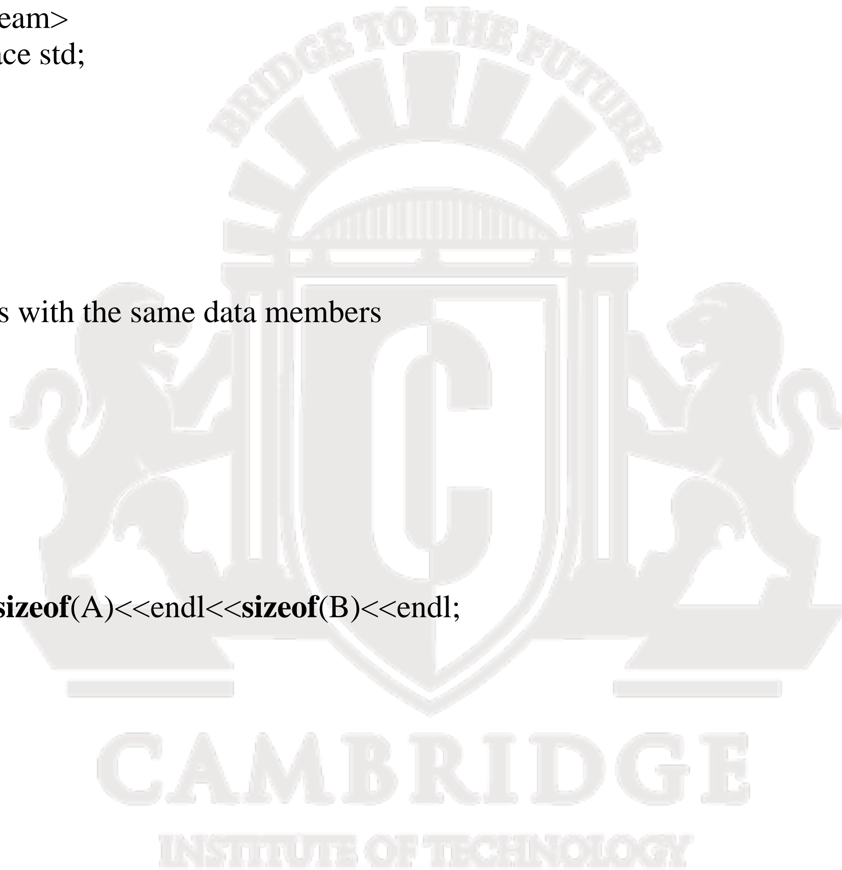
---

notes4free.in
Source diginotes.in

2.1.2 Objects

Variables of classes are known as objects.

An object of a class occupies the same amount of memory as a variable of a structure that has the same data members. This is illustrated by Listing 2.6. Size of a class object is equal to that of a structure variable with identical data members



```
/*Beginning of objectSize.cpp*/
#include<iostream>
Using namespace std;
struct A
{
        char a;
        int b;
        float c;
};
class B //a class with the same data members
{
        char a;
        int b;
        float c;
};
void main()
{
        cout<<sizeof(A)<<endl<<sizeof(B)<<endl;
}
```

Output

9

9

Scope resolution operator

It is possible and usually necessary for the library programmer to define the member functions outside their respective classes.

The scope resolution operator makes this possible.

The use of the scope resolution operator (::).

notes4free.in
Source diginotes.in

```
/*Beginning of scopeResolution.cpp*/
class Distance
{
        int iFeet;
        float fInches;
public: void setFeet(int); //prototype only
        int getFeet(); //prototype only
        void setInches(float); //prototype only
        float getInches(); //prototype only
};
void Distance::setFeet(int x) //definition
{
        iFeet=x;
}
int Distance::getFeet() //definition
{
        return iFeet;
}
void Distance::setInches(float y) //definition
{
        fInches=y;
}
float Distance::getInches() //definition
{
        return fInches;
} /*End of scopeResolution.cpp*/
```

We can observe that the member functions have been only prototyped within the class; they have been defined outside. The scope resolution operator signifies the class to which they belong.

The class name is specified on the left-hand side of the scope resolution operator. The name of the function being defined is on the right-hand side.

Creating Libraries Using the Scope Resolution Operator As in C language, creating a new data type in C++ using classes is also a three-step process that is executed by the library programmer.

Step 1: Place the class definition in a header file.

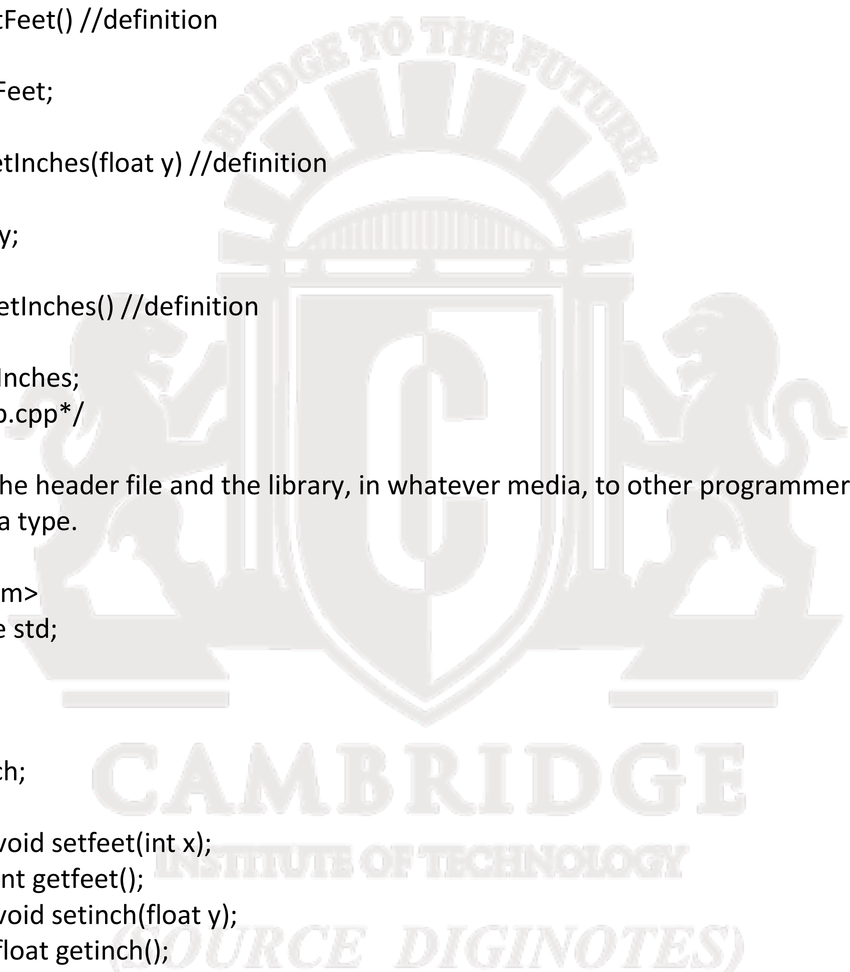
```
/*Beginning of Distance.h*/
/*Header file containing the definition of the Distance class*/
class Distance
{
        int iFeet; float fInches;
public: void setFeet(int); //prototype only
        int getFeet(); //prototype only
        void setInches(float); //prototype only
        float getInches(); //prototype only
}; /*End of Distance.h*/
```


notes4free.in
Source diginotes.in

Step 2: Place the definitions of the member functions in a C++ source file (the library source code). A file that contains definitions of the member functions of a class is known as the implementation file of that class. Compile this implementation file and put in a library.

```
 /*Beginning of Distlib.cpp*/
 /*Implementation file for the class Distance*/
#include"Distance.h"
void Distance::setFeet(int x) //definition
 {
        iFeet=x;
 }
 int Distance::getFeet() //definition
{
        return iFeet;
 }
void Distance::setInches(float y) //definition
 {
       fInches=y;
}
float Distance::getInches() //definition
 {
        return fInches;
 } /*End of Distlib.cpp*/
```

Step 3: Provide the header file and the library, in whatever media, to other programmers who want to use this new data type.




```
#include<iostream>
using namespace std;
class dist
{
       int ifeet;
       float finch;
       public:
              void setfeet(int x);
              int getfeet();
              void setinch(float y);
              float getinch();
};
   void dist::setfeet(int x)
               {
                     ifeet=x;
                     }
              int dist:: getfeet()
              {
                     return ifeet;
              }
              void dist:: setinch(float y)
              {
```


notes4free.in
Source diginotes.in

```
                    finch=y;
            }
            float dist::getinch()
            {
                    return finch;
            }

int main()
{
        dist d1;
        d1.setfeet(29);
        d1.setinch(3.8);
        cout<<d1.getfeet()<<endl<<d1.getinch();

}
```

This pointer

Every object in c++ has access to its own address through an important pointer called this pointer.

The this pointer is an implicit parameter to all member function. Therefore , inside a member function , this may be used to refer to the invoking object .

The this pointer is always a constant pointer. The this pointer always points at the object with respect to which the function was called.

```
#include<iostream>

using namespace std;

class Test
{
        int x;
        public:
        void setx(int x)
        {
                this->x=x;
        }
        void print()
        {
                cout<<"x="<<x<<endl<<"address of obj"<<this;
        }
};
int main()
{
        Test b1,b2;
        b1.setx(5);
        b1.print();
        b2.setx(8);
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
        b2.print();
}
```

An explanation that follows shortly explains why and how it functions.

After the compiler has ascertained that no attempt has been made to access the private members of an object by non-member functions, it converts the C++ code into an ordinary C language code as follows:

1. It converts the class into a structure with only data members as follows.

Before

```
class Distance
{
        int iFeet;
        float fInches;
public: void setFeet(int);//prototype      only
        int getFeet();      //prototype only
        void setInches(float);     //prototype only
        float getInches();         //prototype only
};
```

After

```
struct Distance
{
int iFeet;
float fInches;
};
```

2. It puts a declaration of the this pointer as a leading formal argument in the prototypes of all member functions as follows. (Distance * const)

Before

```
void setFeet(int);
```

After

```
void setFeet(Distance * const, int);
```

Before

```
int getFeet();
```

After

```
int getFeet(Distance * const);
```

Before

```
void setInches(float);
```

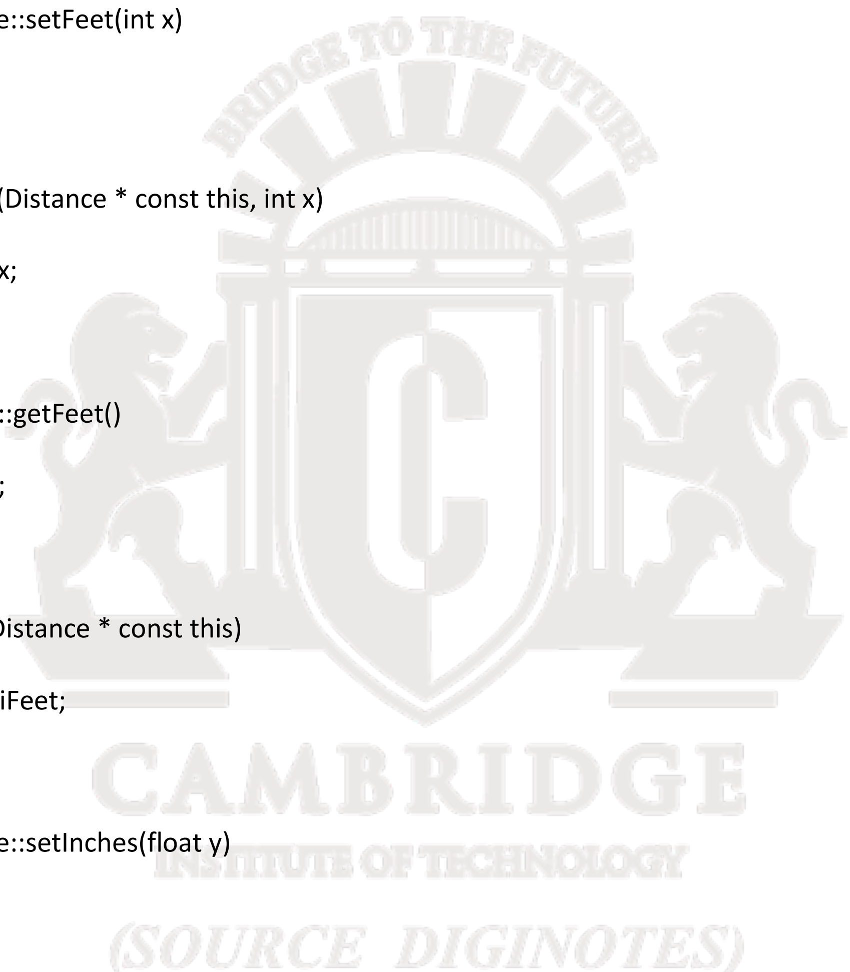
After

```
void setInches(Distance * const, float);
```


notes4free.in
Source diginotes.in

```
Before
 float getInches();
After
 float getInches(Distance * const);
```

3. It puts the definition of the this pointer as a leading formal argument in the definitions of all member functions as follows. It also modifies all the statements to access object members by accessing them through the this pointer using the pointer-to-member access operator (->).




```
 Before
void Distance::setFeet(int x)
 {
iFeet=x;
}
 After
void setFeet(Distance * const this, int x)
 {
 this->iFeet=x;
}

 Before
 int Distance::getFeet()
 {
 return iFeet;
 }

After
int getFeet(Distance * const this)
 {
return this->iFeet;
}

 Before
void Distance::setInches(float y)
 {
 fInches=y;
 }
 After
void setInches(Distance * const this, float y)
 {
 this->fInches=y;
 }
 Before
float Distance::getInches()
 {
return fInches;
 }
 After float getInches(Distance * const this)
```


notes4free.in
Source diginotes.in

```
{
 return this->fInches;
 }
```

We must understand how the scope resolution operator works. The scope resolution operator is also an operator. Just like any other operator, it operates upon its operands. The scope resolution operator is a binary operator, that is, it takes two operands. The operand on its left is the name of a pre-defined class. On its right is a member function of that class.

Based upon this information, the scope resolution operator inserts a constant operator of the correct type as a leading formal argument to the function on its right.

For example, if the class name is Distance, as in the above case, the compiler inserts a pointer of type Distance * const as a leading formal argument to the function on its right.

4. It passes the address of invoking object as a leading parameter to each call to the member functions as follows.

```
Before
    d1.setFeet(1);
After
    setFeet(&d1,1);

Before
    d1.setInches(1.1);
After
    setInches(&d1,1.1);
Before
        cout<<d1.getFeet()<<endl;
After
        cout<<getFeet(&d1)<<endl;
```

Example :

```
Class dist
{
        int ifeet;
        float finch;
        public:
         void setfeet(int ifeet)
           {
                this->ifeet=ifeet;
                }
         int getfeet()
         {
                return ifeet;
         }
         void setinch(float finch)
```

```
		{
			this->finch=finch;
		}
		float getinch()
		{
			return finch;
		}

};
int main()
{
		dist d1,d2;
		d1.setfeet(29);
		d1.setinch(3.8);
		d2.setfeet(30);
		d2.setinch(31.8);
		cout<<d1.getfeet()<<endl<<d1.getinch()<<endl;
		cout<<d2.getfeet()<<endl<<d2.getinch();

}
```

Accessing data members of local objects inside member functions and of objects that are passed as parameters

```
/*Beginning of Distance.h*/
 class Distance
 {
		/* rest of the class Distance */
		Distance add(Distance);
}; /*End of Distance.h*/

/*Beginning of Distlib.cpp*/
#include"Distance.h"
Distance Distance::add(Distance dd)
 {
		Distance temp;
		temp.iFeet=iFeet+dd.iFeet; //legal to access both temp.iFeet and dd.iFeet
		temp.fInches=fInches+dd.fInches;
		return temp;
 }

 /* definitions of the rest of the functions of class	Distance
/*End of Distlib.cpp*/

/*Beginning of Distmain.cpp*/
 #include<iostream>
 #include"Distance.h"
```


notes4free.in
Source diginotes.in

```
using namespace std;

int main()
{
		Distance d1,d2,d3;
		d1.setFeet(1);
		d1.setInches(1.1);
		d2.setFeet(2);
		d2.setInches(2.2);
		d3=d1.add(d2);
		cout<<d3.getFeet()<<"'-"<<d3.getInches()<<"'\n";
}
/*End of Distmain.cpp*/ Output 3'-3.3'
```

Explicit address manipulation

```
#include<iostream>
using namespace std;
class dist
{
		int ifeet;
		float finch;
		public:
		  void setfeet(int x)
		    {
		    ifeet=x;
		        }
		  int getfeet()
		  {
		        return ifeet;
		  }
		  void setinch(float y)
		  {
		        finch=y;
		  }
		  float getinch()
		  {
		        return finch;
		  }

};
int main()
{
```


notes4free.in
Source diginotes.in

```
        dist d1;
        d1.setfeet(256);
        d1.setinch(3.8);
        char *p=(char *)&d1;
        *p=1;
        cout<<d1.getfeet()<<endl<<d1.getinch()<<endl;
}
```

Arrow operator

```
#include<iostream>
using namespace std;
class dist
{
        int ifeet;
        float finch;
        public:
          void setfeet(int x)
            {
            ifeet=x;
                    }
          int getfeet()
          {
                return ifeet;
          }
          void setinch(float y)
          {
                finch=y;
          }
          float getinch()
          {
                return finch;
          }

};
int main()
{
        dist d1,*d2;
        d1.setfeet(256);
        d1.setinch(3.8);
        d2=&d1;
        cout<<d2->getfeet()<<endl<<d2->getinch()<<endl;
}
```

Calling one member function from another

```
#include<iostream>
using namespace std;
class dist
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in
Source diginotes.in

```
{
        int ifeet;
        float finch;
        public:
          void setfeet(int x)
            {
            ifeet=x;
                    }
          int getfeet()
          {
                return ifeet;
          }
          void setinch(float y)
          {
                finch=y;
          }
          float getinch()
          {
                return finch;
          }
          void setfeetfeet(int r)
          {
                setfeet(r);
          }
};
int main()
{
        dist d1;
        d1.setfeetfeet(256);
        d1.setinch(3.8);
        cout<<d1.getfeet()<<endl<<d1.getinch()<<endl;
}
```

//Overloaded member function

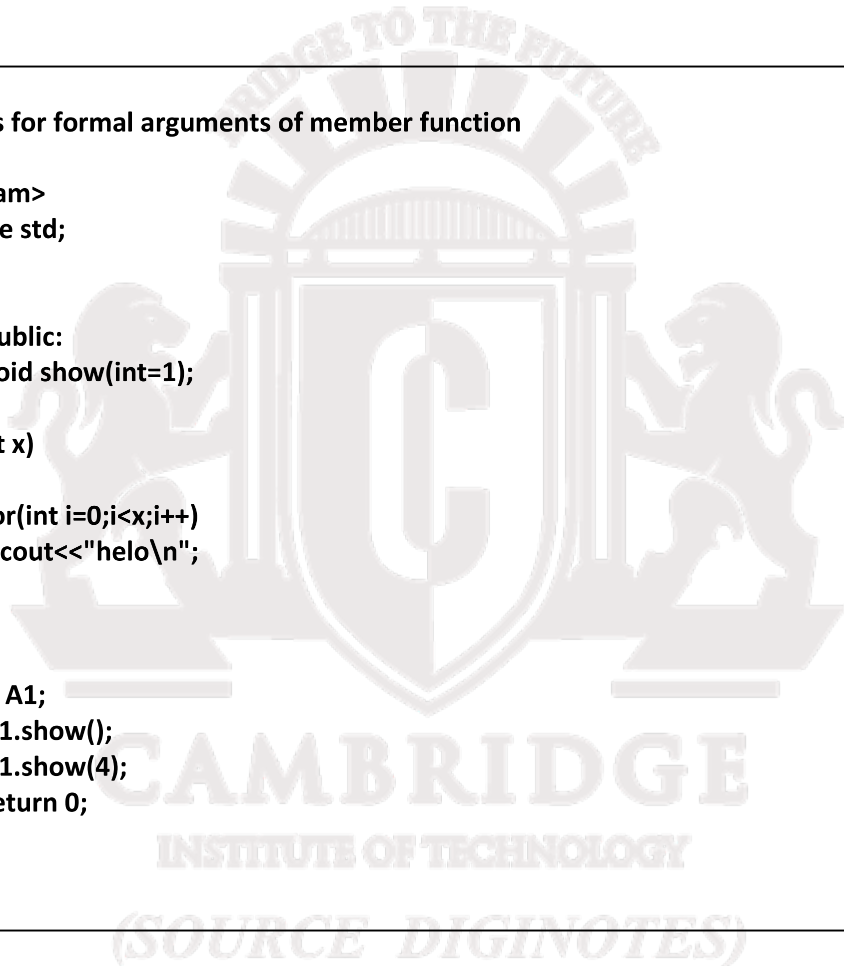
CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
#include<iostream>
using namespace std;
class A
{
        public:void show();
        void show(int);
};
void A::show()
{
        cout<<"HI\n";
}
void A::show(int x)
{
```


notes4free.in
Source diginotes.in

```
        for(int i=0;i<x;i++)
          cout<<"helo\n";
}
int main()
{
        A A1;
        A1.show();
        A1.show(3);
        return 0;
}
```

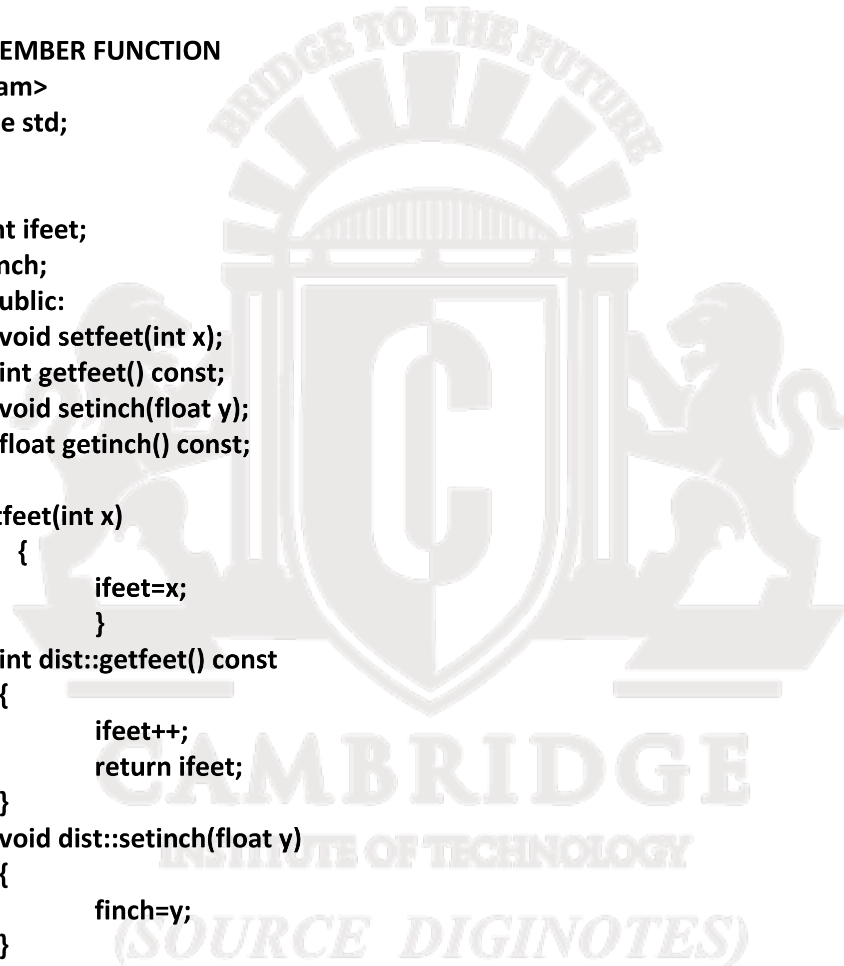
```
//Default values for formal arguments of member function

#include<iostream>
using namespace std;
class A
{
        public:
        void show(int=1);
};
void A::show(int x)
{
        for(int i=0;i<x;i++)
          cout<<"helo\n";
}
int main()
{
        A A1;
        A1.show();
        A1.show(4);
        return 0;
}
```


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

```
//INLINE MEMBER FUNCTION
#include<iostream>
using namespace std;
 #define square(v) v*v
 inline int square1(int x)
 {
        int r=0;
        r=x*x;
        return r;
 }
```


notes4free.in
Source diginotes.in

```
int main()
{
        int r1=0,r2=0,r3=0,r4=0;
        r1=square(5);
        r2=square(2+3);
        r3=square1(5);
        r4=square1(2+3);
        cout<<"r1="<<r1<<endl<<"r2="<<r2<<endl<<"r3="<<r3<<endl<<"r4="<<r4<<endl;
}
```



```
//CONSTANT MEMBER FUNCTION
#include<iostream>
using namespace std;
class dist
{
        int ifeet;
mutable float finch;
        public:
         void setfeet(int x);
         int getfeet() const;
         void setinch(float y);
         float getinch() const;
};
  void dist::setfeet(int x)
          {
               ifeet=x;
               }
         int dist::getfeet() const
         {
//             ifeet++;
               return ifeet;
         }
         void dist::setinch(float y)
         {
               finch=y;
         }
         float dist::getinch() const
         {
        finch=0.0;
               return finch;
         }

int main()
{
        dist d1;
        d1.setfeet(29);
        d1.setinch(3.8);
```


notes4free.in
Source diginotes.in

```
                cout<<d1.getfeet()<<endl<<d1.getinch();

}
```

```
//MUTABLE DATA MEMBER
#include<iostream>
using namespace std;
class dist
{
                 mutable int ifeet;
                float finch;
                public:
                  void setfeet(int x);
                  int getfeet()const ;
                  void setinch(float y);
                  float getinch();
};
    void dist::setfeet(int x)
                   {
                        ifeet=x;
                        }
                  int dist::getfeet() const
                  {
                        ifeet++;
                        return ifeet;
                  }
                  void dist::setinch(float y)
                  {
                        finch=y;
                  }
                  float dist::getinch()
                              {
                        finch=0.0;
                        return finch;
                  }

int main()
{
                dist d1;
                d1.setfeet(29);
                d1.setinch(3.8);
                cout<<d1.getfeet()<<endl<<d1.getinch();

}
```

---

notes4free.in
Source diginotes.in

Friend as non member function

A friend function is a non-member function that has special rights to access private data members of any object of the class of whom it is a friend.

A friend function is prototyped within the definition of the class of which it is intended to be a friend.

The prototype is prefixed with the keyword friend.

Since it is a non-member , it is defined without using the scope resolution operator. Moreover, it is not called with respect to an object.

```
#include<iostream>
using namespace std;
class A
{
	int x;
	public:
		void setx(int);
		int getx();
		friend void Display(A a);
};
void Display(A a)
{
	a.x=10;
	cout<<"now the x value after changing"<<a.x<<endl;
}

void A::setx(int a)
{
	x=a;
}
int A::getx()
{
	return x;
}

int main()
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in
Source diginotes.in

```
{
        A a1;
        a1.setx(3);
        cout<<"the value of x is"<<a1.getx()<<endl;
        Display(a1);
}
```

A few points about the friend functions that we must keep in mind are as follows:
*) friend keyword should appear in the prototype only and not in the definition.

*) Since it is a non-member function of the class of which it is a friend, it can be prototyped in either the private or the public section of the class.

*) A friend function takes one extra parameter as compared to a member function that performs the same task. This is because it cannot be called with respect to any object. Instead, the object itself appears as an explicit parameter in the function call.

*)We need not and should not use the scope resolution operator while defining a friend function.

Friend as a class

A class can be a friend of another class.

Member functions of a friend class can access private data members of objects of the class of which it is a friend.

If class B is to be made a friend of class A, then the statement

friend class B; should be written within the definition of class A.

It does not matter whether the statement declaring class B as a friend is mentioned within the private or the public section of class A. Now, member functions of class B can access the private data members of objects of class A

```
#include<iostream>
using namespace std;
class A
{
```


notes4free.in
Source diginotes.in

```
        int x;
        public:
         void setx(int);
        int getx();
        friend class B;
};
class B
{
        public:
                void read(A a);
};
void A::setx(int a)
{
        x=a;
}
int A::getx()
{
        return x;
}
void B::read(A a)
{
        a.x=10;
        cout<<"now the x value after accessing in B is"<<a.x<<endl;
}
int main()
{
        A a1;
        B b;
        a1.setx(3);
        cout<<"the value of x is"<<a1.getx()<<endl;
        b.read(a1);
}
```

Friend as a member function of another class

Friend member functions How can we make some specific member functions of one class friendly to another class?

For making only B::test_friend() function a friend of class A, replace the line

friend class B; in the declaration of the class A with the line

notes4free.in

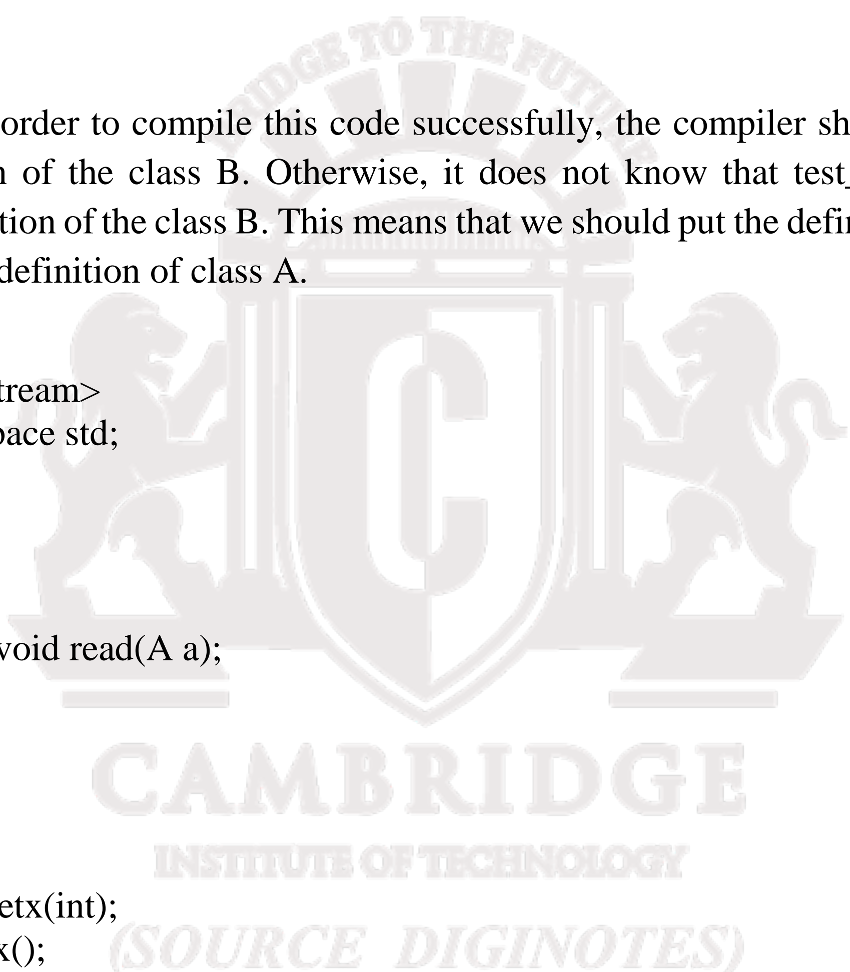
Source diginotes.in

friend void B::test_friend();

The modified definition of the class A is

```
class A {
/* rest of the class A */
 friend void B::test_friend();
 };
```

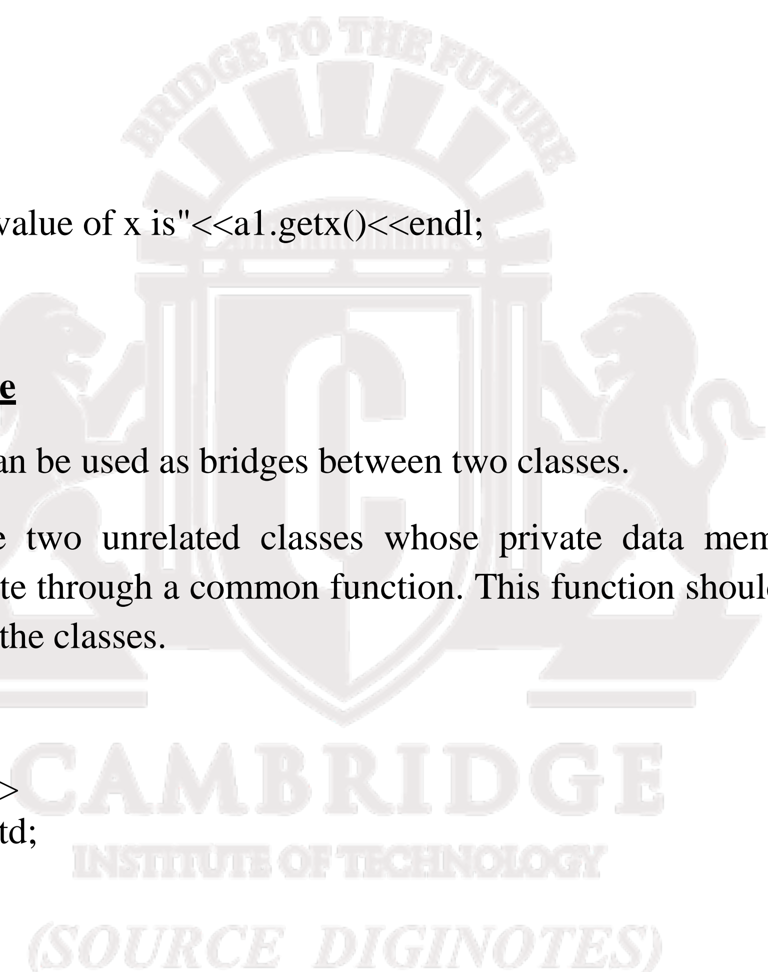
However, in order to compile this code successfully, the compiler should first see the definition of the class B. Otherwise, it does not know that test_friend() is a member function of the class B. This means that we should put the definition of class B before the definition of class A.



```
#include<iostream>
using namespace std;
class A;
class B
{
	public:
		void read(A a);
};
class A
{
	int x;
	public:
	 void setx(int);
	int getx();
	friend void B::read( A a);
};

void A::setx(int a)
{
	x=a;
}
int A::getx()
{
```


notes4free.in
Source diginotes.in

```
        return x;
}
void B::read(A a)
{

        a.x=10;
        cout<<"now the x value after accessing in B is"<<a.x<<endl;
}
int main()
{
        A a1;
        B b;
        a1.setx(5);
        cout<<"the value of x is"<<a1.getx()<<endl;
        b.read(a1);
}
```



Friend as a Bridge

Friend functions can be used as bridges between two classes.

Suppose there are two unrelated classes whose private data members need a simultaneous update through a common function. This function should be declared as a friend to both the classes.

```
#include<iostream>
using namespace std;
class A;
class B
{
        int y;
        public:void sety(int v)
        {
                y=v;
        }
                int gety( )
                  {
                        return y;
                   }
```


notes4free.in
Source diginotes.in

```
                friend void Display(A a,B b);
};
class A
{
        int x;
        public:void setx(int v)
        {
                x=v;
        }
                int getx( )
                  {
                        return x;
                   }

        friend void Display( A a,B b);
};
void Display(A a, B b)
{
cout<<"after changing";
a.x=20;
b.y=25;
cout<<"x="<<a.x<<"y="<<b.y<<endl;
}
int main()
{
        A a;
        B b;
        a.setx(100);
        b.sety(200);
        cout<<"x="<<a.getx()<<"y="<<b.gety()<<endl;
        Display(a,b);
}
```

Static Data Member

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class.

notes4free.in
Source diginotes.in

All static data is initialized to zero when the first object is created, if no other initialization is present.

We can't put it in the class definition but it can be initialized outside the class as done in the following example by declaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

Introducing static data members does not increase the size of objects of the class. Static data members are not contained within objects. There is only one copy of the static data member in the memory.
Static data members are not a part of objects

```
/*Beginning of staticSize.cpp*/
#include<iostream>
using namespace std;
class Account
{
	static int x;
	float y;
};
int main()
{
	Account a;
	cout<<"size of account is"<<sizeof(a)<<endl;

}
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

```
#include<iostream>
using namespace std;
class sample
{
public:static int a,b;
};
int sample::a;
int sample::b=10;
int main()
{
	sample s;
```

notes4free.in

Source diginotes.in

```
        cout<<"a= "<<s.a<<endl;
        cout<<"b="<<sample::b<<endl;

}
```

```
#include<iostream>
using namespace std;
class sample
{
public:static int a,b;
void sum()
{
        int s=a+b;
        cout<<"sum  "<<s<<endl;
}
};
int sample::a;
int sample::b=10;
int main()
{
        sample s;
        cout<<"a= "<<s.a<<endl;
        cout<<"b="<<sample::b<<endl;
        s.sum();
}
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

static Function Members

By declaring a function member as static, you make it independent of any particular object of the class.

A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

notes4free.in
Source diginotes.in

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

```
#include<iostream>
using namespace std;
class sample
{
public:static int a,b;
static int x;
static void sum()
{
        int s=a+b;
        int avg=s/x;
        cout<<"sum "<<s<<endl;
        cout<<"avg  "<<avg<<endl;
}
};
int sample::a;
int sample::b=10;
int main()
{
        sample s;
        cout<<"a= "<<s.a<<endl;
        cout<<"b="<<sample::b<<endl;
        sample::sum();
        s.sum();
}
```

STATIC VARIABLE CAN BE USED AS DEFAULT VALUE

```
#include<iostream>
using namespace std;
class Account
{
        static int x;
        public :
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in
Source diginotes.in

```
        void display(int=x);

};
int Account::x=5;
void Account::display(int m)
        {
                        cout<<"m value="<<m<<endl;
        }
int main()
{
        Account a;
        a.display();
        a.display(100);
}
```

Example :

```
#include<iostream>
using namespace std;
class Account
{
        int z;
        static float rate;
        static char name[30];
public:         void interest(float p,int t);
};
float Account::rate=5;
char Account::name[30]="state bank of india";

void Account::interest(float p,int t)
{
        cout<<"the name of the bank is    "<<name<<endl;
        float i=p*t*rate/100;
        cout<<"interest="<<i<<endl;
}
int main()
{
        Account a;
        a.interest(1000,2);
```


notes4free.in
Source diginotes.in

```
        // cout<<"size of Account"<<sizeof(Account)<<endl;
}
```

Namespace

```
#include<iostream>
using namespace std;
namespace a
{
 void add()
        {
                int a=5,b=9.4;
                int sum=a+b;
                cout<<"sum="<<sum;
        }
}

 namespace b
 {
        void add()
        {
                float a=5,b=9.4;
                float sum=a+b;
                cout<<"sum="<<sum;
         }
 }
```

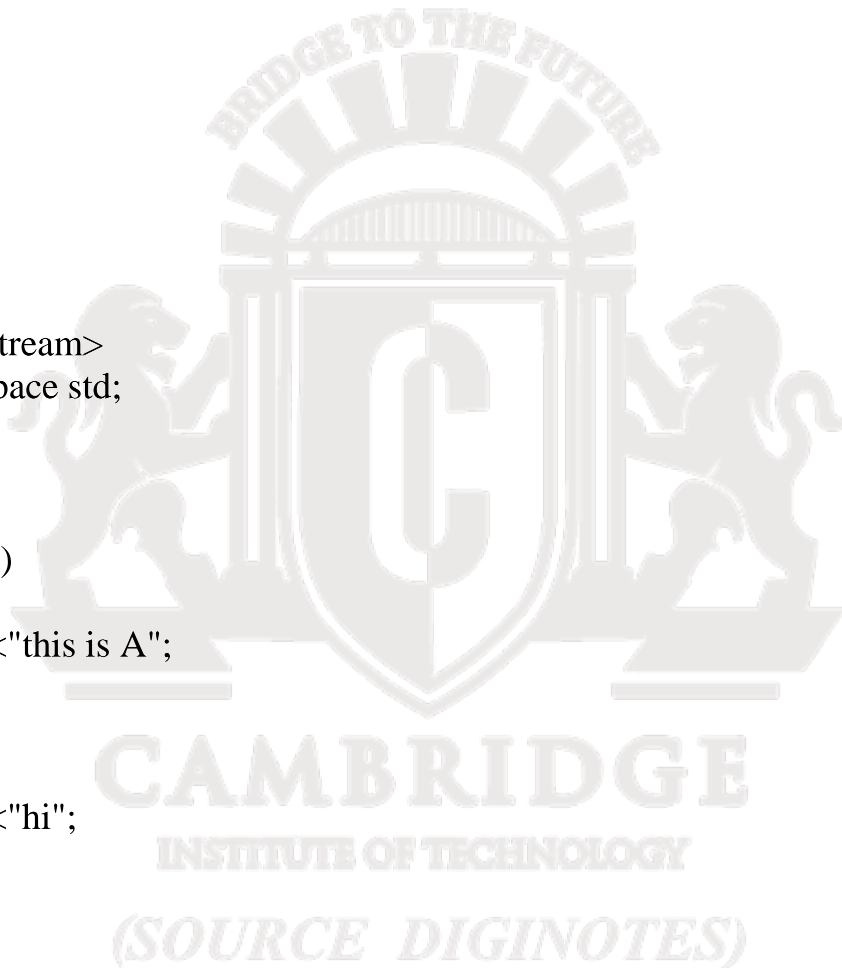
```
//namespace.cpp

#include<iostream>
#include "a.cpp"
#include "b.cpp"
using namespace a;
using namespace b;

using namespace std;
int main()
```


notes4free.in
Source diginotes.in
CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
{
b::display();
        read();

}



int main()
{
a::add();
b::add();
}
```

a.cpp

```
#include<iostream>
using namespace std;
namespace a
{

void display()
{
        cout<<"this is A";
}
void read()
{
        cout<<"hi";
}
}
```

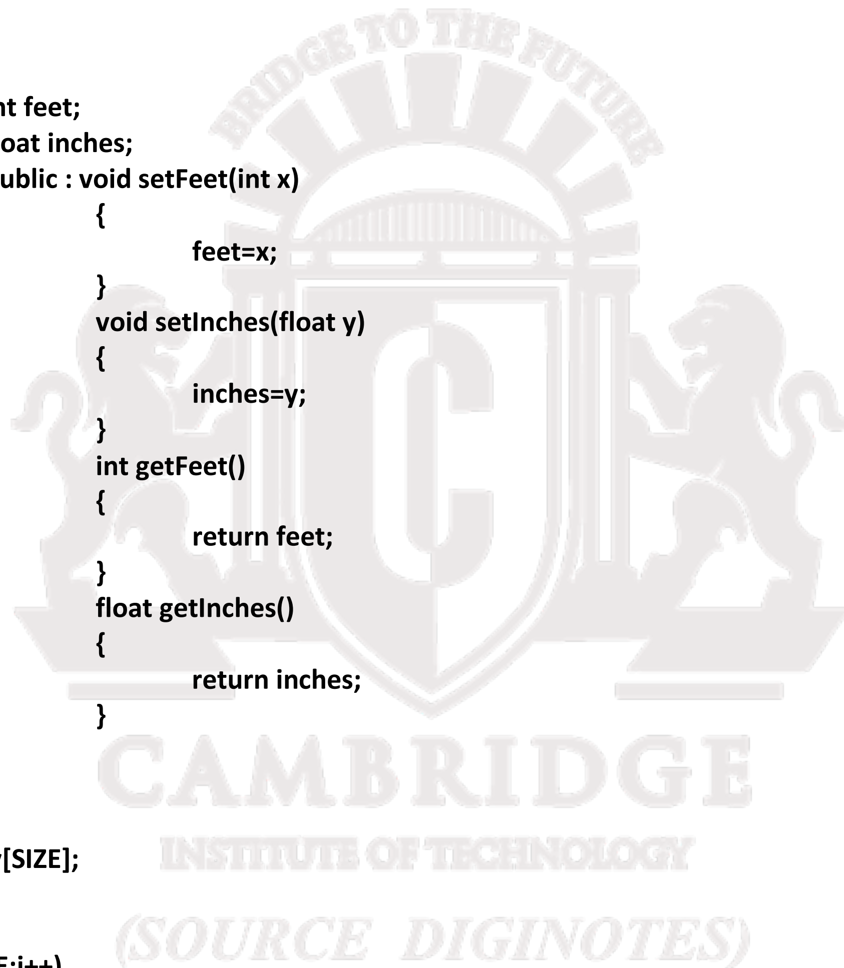


b.cpp

```
#include<iostream>
using namespace std;
namespace b
{

void display()
{
```

```
cout<<"this is B";
}
}
```

```
//ARRAY OF OBJECT
#include<iostream>
using namespace std;
#define SIZE 3
class Distance
{
        int feet;
        float inches;
        public : void setFeet(int x)
                {
                        feet=x;
                }
                void setInches(float y)
                {
                        inches=y;
                }
                int getFeet()
                {
                        return feet;
                }
                float getInches()
                {
                        return inches;
                }
};
int main()
{
Distance dArray[SIZE];
int a;
float b;
for(int i=0;i<SIZE;i++)
{
cout<<"Enter the feet : ";
cin>>a;
dArray[i].setFeet(a);
cout<<"Enter the inches : ";
cin>>b;
dArray[i].setInches(b);
}
for(int i=0;i<SIZE;i++)
{
```



notes4free.in
Source diginotes.in

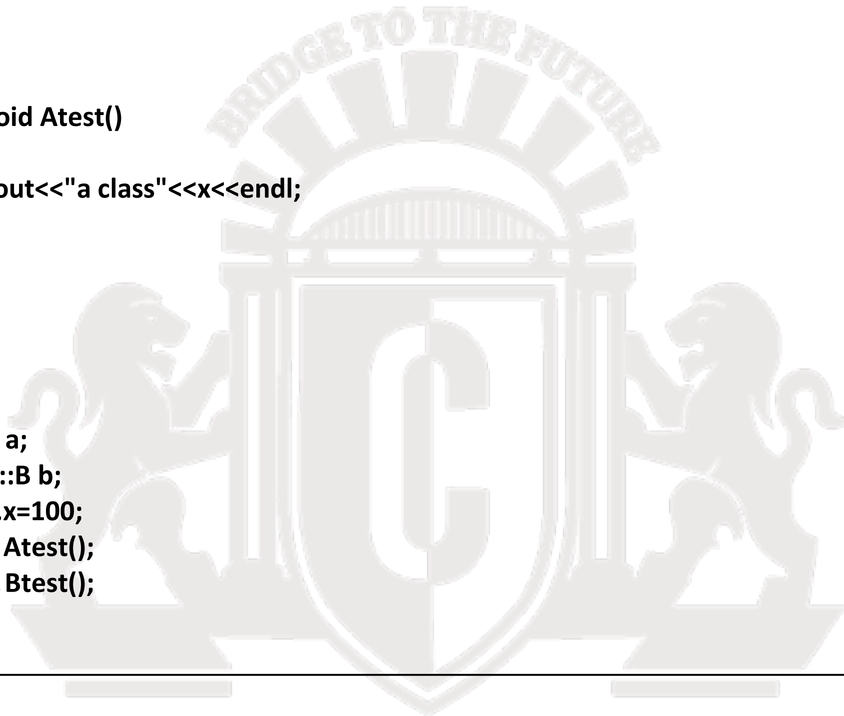
```
cout <<dArray[i].getFeet()<<" "<<dArray[i].getInches()<<endl;
}
}
```

```
//ARRAY INSIDE OBJECT
#include<iostream>
using namespace std;
#define size 3

class student
{
            int roll_no;
            int marks[size];
            public:
             void getdata ()
             {
                   cout<<"\nEnter roll no: ";
                   cin>>roll_no;
                   for(int i=0; i<size; i++)
                   {
                         cout<<"Enter marks in subject"<<(i+1)<<": ";
                         cin>>marks[i] ;
                   }
             }
             void tot_marks()
             {
                   int total=0;
                   for(int i=0; i<size; i++)
                   total=total+ marks[i];
                   cout<<"\n\nTotal marks "<<total;
             }
};


int main()
{
student s;
s.getdata() ;
s.tot_marks() ;

}
```

NESTED CLASS

```
#include<iostream>
using namespace std;
```

```
class A
{
                public: int x;
                public :
                class B
                {
                  public:void Btest()
                  {
                           cout<<"b class"<<endl;

     }

                };
                void Atest()
                {
                cout<<"a class"<<x<<endl;
                }


};
int main()
{
                A a;
                A::B b;
                a.x=100;
                a.Atest();
                b.Btest();
}
```



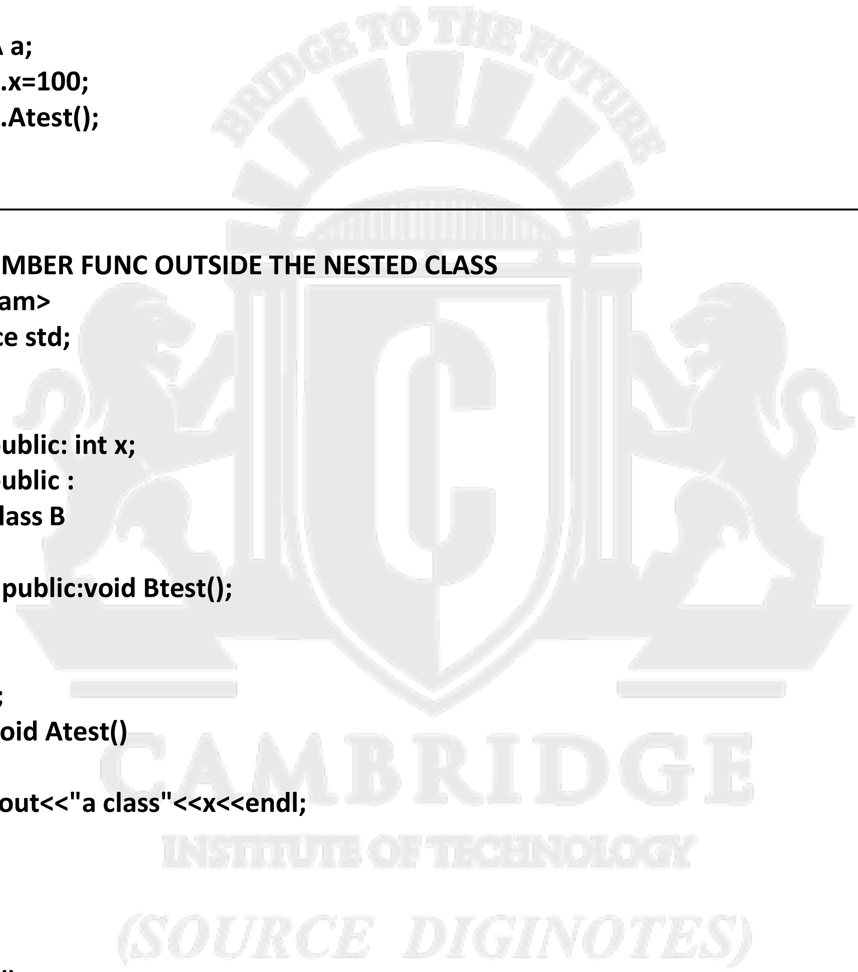
```
//CREATING OBJECT INSIDE THE NESTED CLASS
#include<iostream>
using namespace std;
class A
{
                public: int x;
                public :
                class B
                {
                  public:void Btest();


                };
                B b1;
                void Atest()
                {
                  b1.Btest();
                cout<<"a class"<<x<<endl;
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

Source diginotes.in

```
                }


};
void A::B::Btest()
{
                        cout<<"b class"<<endl;

}
int main()
{
                A a;
                a.x=100;
                a.Atest();
}
```

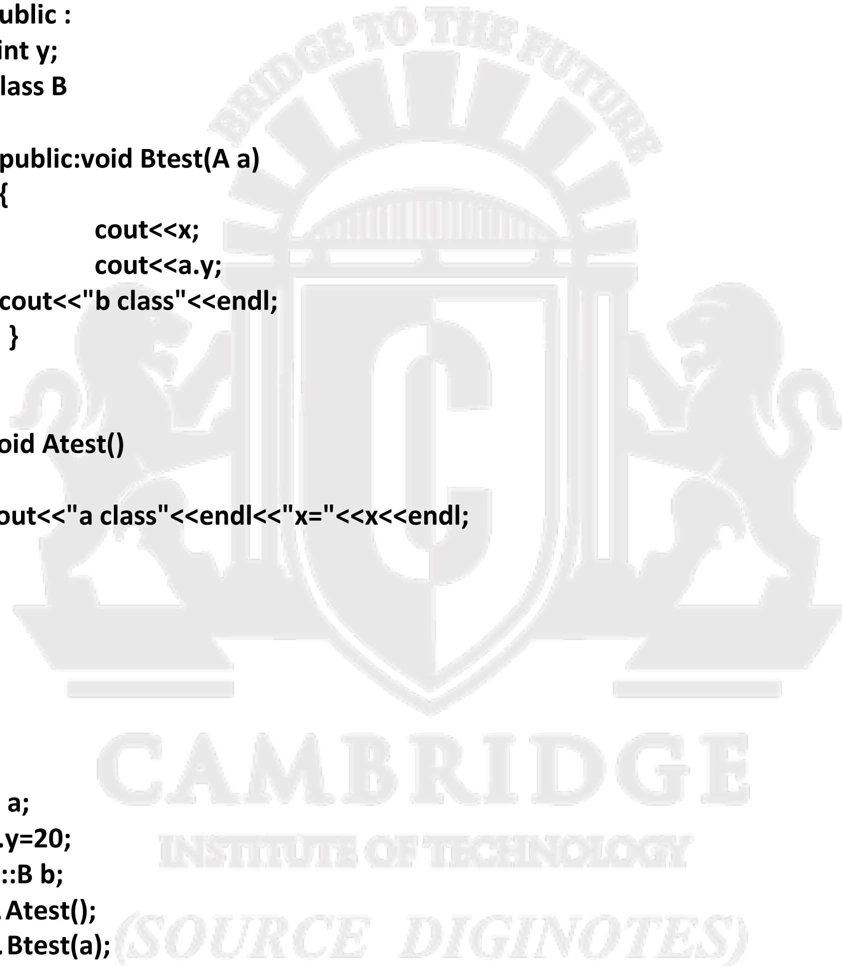
```
//DEFINING MEMBER FUNC OUTSIDE THE NESTED CLASS
#include<iostream>
using namespace std;
class A
{
                public: int x;
                public :
                class B
                {
                  public:void Btest();


                };
                void Atest()
                {
                cout<<"a class"<<x<<endl;
                }


};
void A::B::Btest()
{
                        cout<<"b class"<<endl;

}
int main()
{
                A a;
                A::B b;
                a.x=100;
                a.Atest();
```



notes4free.in
Source diginotes.in

```
		b.Btest();
}
```

```
//ACCESSING NESTED CLASS VARIABLE
#include<iostream>
using namespace std;
class A
{
		static int x;

		public :
		 int y;
		class B
		{
		  public:void Btest(A a)
		  {
				cout<<x;
				cout<<a.y;
		  cout<<"b class"<<endl;
		   }

		};
		void Atest()
		{
		cout<<"a class"<<endl<<"x="<<x<<endl;
		}

};
int A::x=100;

int main()
{
		A a;
		a.y=20;
		A::B b;
		a.Atest();
		b.Btest(a);
}
```

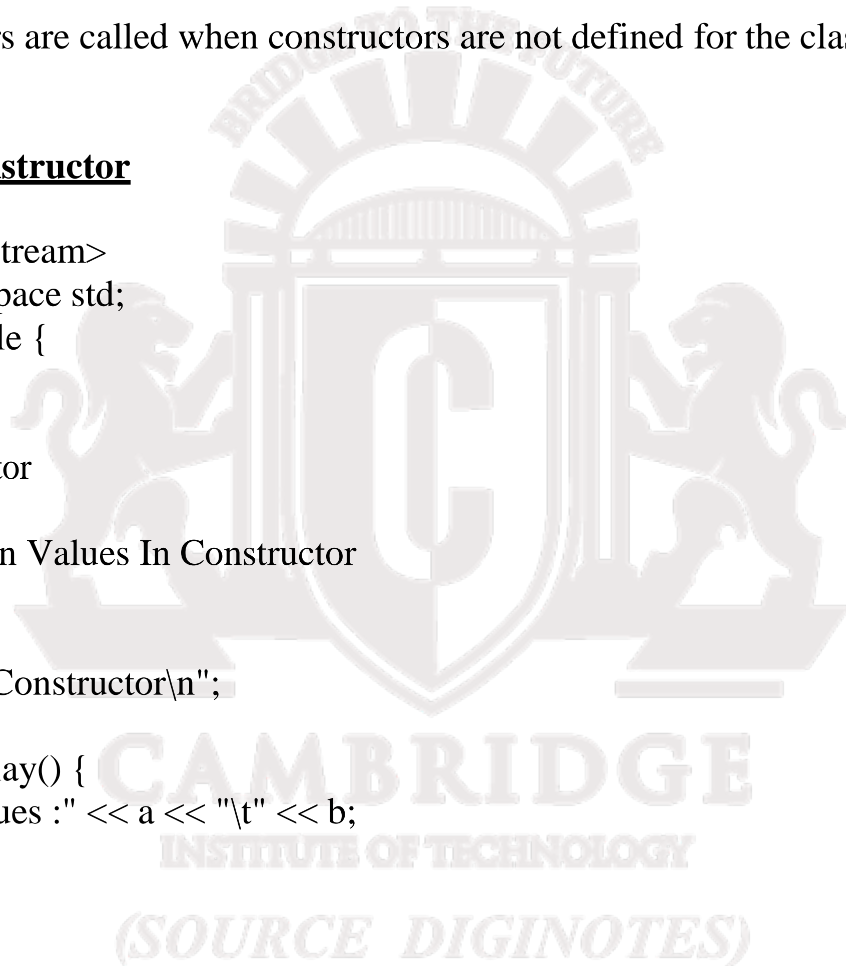


notes4free.in
Source diginotes.in

Rules of Constructor

In C++, Constructor is automatically called when an object (a instance of the class) create. It is a special member function of the class.

- It has the same name of the class.
- It must be a public member.
- No Return Values.
- Default constructors are called when constructors are not defined for the classes.



Default Constructor

```
#include<iostream>
using namespace std;
class Example {
int a, b;
public:
   //Constructor
Example() {
      // Assign Values In Constructor
      a = 10;
      b = 20;
cout<< "Im Constructor\n";
   }
   void Display() {
cout<< "Values :" << a << "\t" << b;
   }

};

int main() {
Example Object; // Constructor invoked.
Object.Display();
}
```


notes4free.in
Source diginotes.in

Parameterized Constructor

```
#include<iostream>
using namespace std;
class Example {
int a, b;
public:
   //parameterized Constructor
Example(intx,int y) {
        // Assign Values In Constructor
        a = x;
        b = y;
cout<< "Im parameterized Constructor\n";
   }
   void Display() {
cout<< "Values :" << a << "\t" << b;
   }
};

Int main() {
   Example Object(10,100); // Constructor invoked.
Object.Display();
}
```

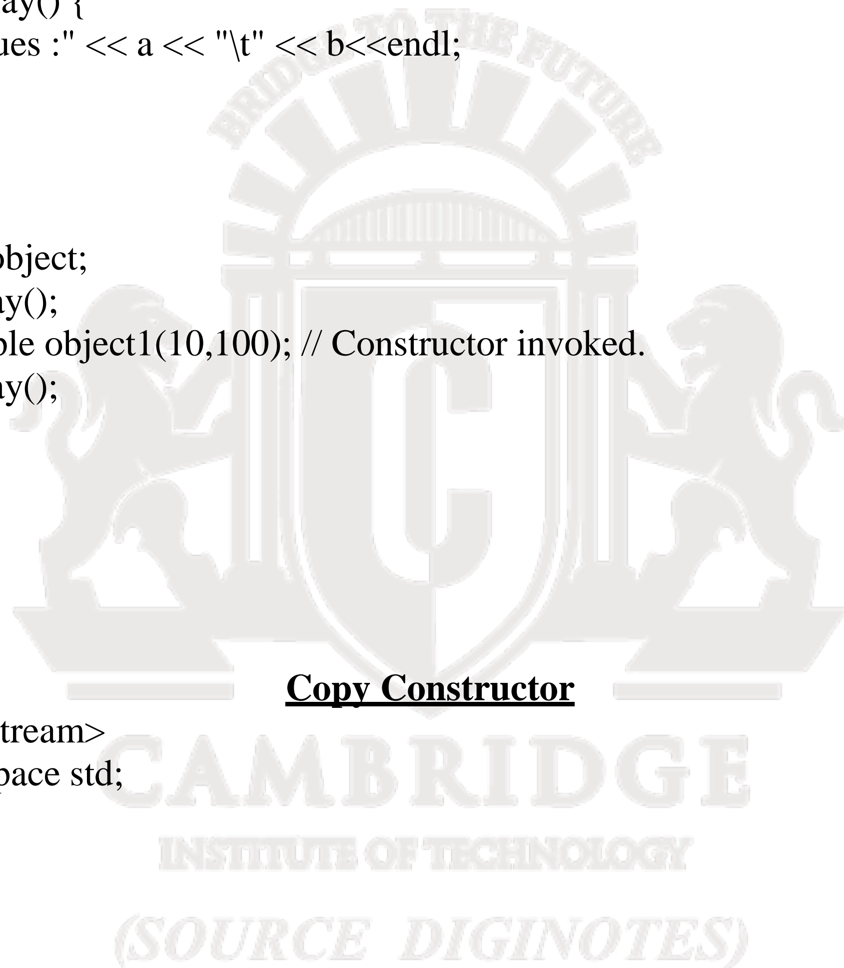
Overloading Constructor

```
#include<iostream>
using namespace std;
class Example {
int a, b;
public:
          //default constructor
          Example()
          {
                    a=3;
                    b=6;
                    cout<<"Im default Constructor"<<endl;
          }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in
Source diginotes.in

```
   //parameterized Constructor
Example(int x,int y) {
       // Assign Values In Constructor
       a = x;
       b = y;
cout<< "Im parameterized Constructor\n";
   }
   void Display() {
cout<< "Values :" << a << "\t" << b<<endl;
   }
};

Int main() {
   Example object;
object.Display();
        Example object1(10,100); // Constructor invoked.
object.Display();

}
```



Copy Constructor

```
#include<iostream>
using namespace std;

class Point
{

int x, y;
public:
Point(int x1, int y1)
         {
          x = x1;
          y = y1;
   }

   // Copy constructor
```


notes4free.in
Source diginotes.in

```
Point( Point&p)
        {
        x = p.x;
        y = p.y;
        }

intgetX()
        {
          return x;
    }
Int getY()
        {
         return y;
        }
};

intmain()
{
    Point p1(10, 15); // Normal constructor is called here
   Point p2 = p1;
   // or Point p2(p1); Copy constructor is called here
    // Let us access values assigned by constructors

cout<< "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
cout<< "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
    return 0;
}
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

Rules of Destructor.

- Should start with a tilde(~) and same name of the class.
- Destructors do not have parameters and return type.
- Destructors are called automatically and cannot be called from a program manually.

Destructor usage

- Releasing memory of the objects.
- Releasing memory of the pointer variables.
- Closing files and resources.

```
Class class_name {
public:
 ~class_name() //Destructor
   {
   }
};
```

notes4free.in
Source diginotes.in

In C++, Constructor is automatically called when an object (a instance of the class) create. It is a special member function of the class.

- It has the same name of the class.
- It must be a public member.
- No Return Values.
- Default constructors are called when constructors are not defined for the classes.

Default Constructor

```
#include<iostream>
using namespace std;
class Example
{
        int a, b;
        public:
                //Constructor
        Example( )
         {
                // Assign Values In Constructor
                a = 10;
                b = 20;
                cout<< "Im Constructor\n";
           }
        void Display()
        {
                cout<< "Values :" << a << "\t" << b;
        }

};

int main( )
 {
Example Object; // Constructor invoked.
Object.Display();
}
```

Source diginotes.in
notes4free.in

Parameterized Constructor

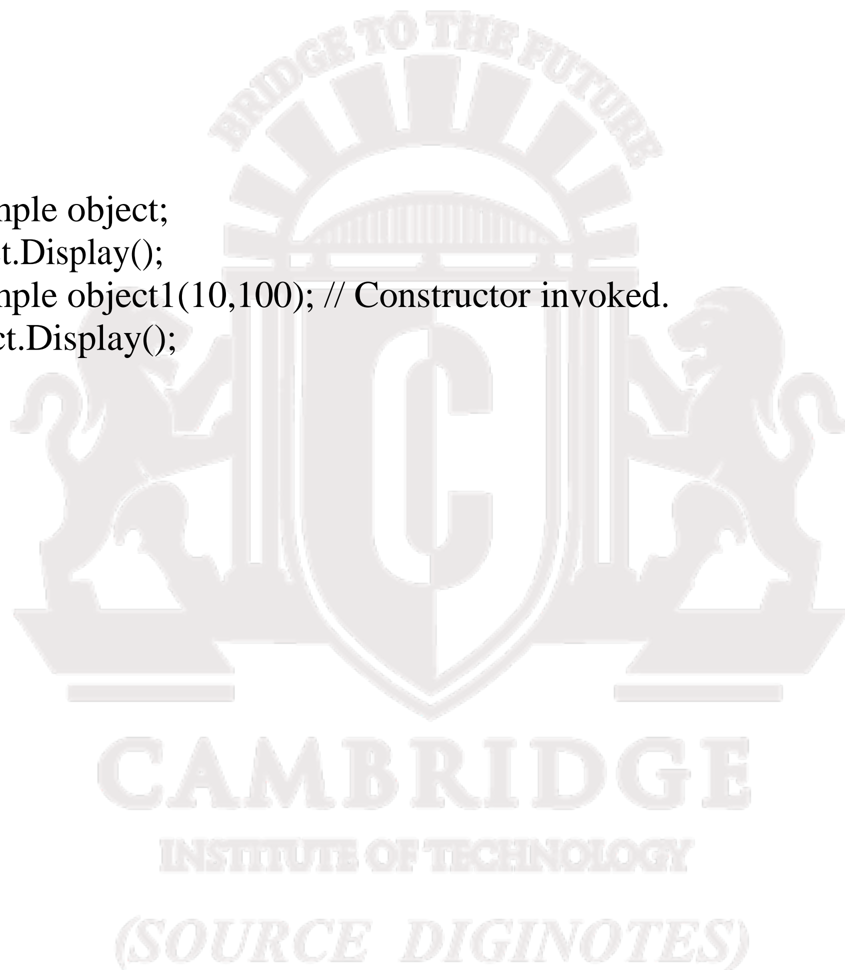
```
#include<iostream>
using namespace std;
class Example
{
        int a, b;
        public:
                //parameterized Constructor
                Example(int x,int y)
                 {
                // Assign Values In Constructor
                        a = x;
                        b = y;
                        cout<< "Im parameterized Constructor\n";
                }
   void Display( )
 {
        cout<< "Values :" << a << "\t" << b;
  }
};

int main( )
 {
         Example Object(10,100); // Constructor invoked.
        Object.Display();
}
```

Overloading Constructor

```
#include<iostream>
using namespace std;
class Example {
int a, b;
public:
        //default constructor
        Example( )
        {
                a=3;
                b=6;
                cout<<"Im default Constructor"<<endl;
        }
```

Source diginotes.in
notes4free.in

```
    //parameterized Constructor
Example(int x,int y)
 {
        // Assign Values In Constructor
        a = x;
        b = y;
          cout<< "Im parameterized Constructor\n";

}
          void Display( )
           {
                    cout<< "Values :" << a << "\t" << b<<endl;
          }
};

int main( )
 {
          Example object;
          object.Display();
          Example object1(10,100); // Constructor invoked.
          object.Display();

}
```



Source diginotes.in
notes4free.in

Copy Constructor

```
#include<iostream>
using namespace std;

class Point
{

int x, y;
public:
Point(int x1, int y1)
          {
           x = x1;
           y = y1;
    }

    // Copy constructor
Point( Point&p)
          {
          x = p.x;
          y = p.y;
          }

int getX()
          {
            return x;
    }
int getY()
          {
           return y;
          }
};

int main()
{
    Point p1(10, 15); // Normal constructor is called here
   Point p2 = p1;
   // or Point p2(p1); Copy constructor is called here
    // Let us access values assigned by constructors

cout<< "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
cout<< "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
    return 0;
}
```

Source diginotes.in
notes4free.in

Rules of Destructor.

- Should start with a tilde(~) and same name of the class.
- Destructors do not have parameters and return type.
- Destructors are called automatically and cannot be called from a program manually.

Destructor usage

- Releasing memory of the objects.
- Releasing memory of the pointer variables.
- Closing files and resources.

```
Class class_name
{
public:
 ~ class_name() //Destructor
   {
   }
};
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

Source diginotes.in

notes4free.in

Module 2:
1.History of Java
2. Evolution of java,
3.DataTypes,Variables and Arrays.
4.Operators.
5.Control statements.

Java is an object-oriented programming language developed by Sun Microsystems, a company best known for its high-end Unix workstations.
• Java is modeled after C++
• Java language was designed to be small, simple, and portable across platforms and operating systems, both at the source and at the binary level
• Java also provides for portable programming with applets. Applets appear in a Web page much in the same way as images do, but unlike images, applets are dynamic and interactive.

The C# Connection

- Java's innovative features, constructs, and concepts have become baseline for any new language.
- C# is closely related to Java.Created by Microsoft to support the .NET Framework.
- Both languages share the same general syntax, support distributed programming, and utilize the same object model.
- There are differences between Java and C#, but the overall "look and feel" of these languages is very similar.

How Java Changed the Internet

- Applet changed the way the content can be rendered online.
- Java also addressed issues associated with the Internet: portability and security

Java Applets

- An *applet* is a special kind of Java program that is designed to be transmitted over Internet and automatically executed by a Java-compatible web browser.
- If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser.
- Applets are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that can execute locally, rather than on the server.
- The applet allows some functionality to be moved from the server to the client.
- In a web page majorly two types of content is rendered.
  - 1st passive information (reading e-mail,is viewing passive data)
  - dynamic, active program(the program's code execution)
- Applet is a dynamic, self-executing program on the client computer, yet it is initiated by the server.

notes4free.in

- Dynamic, networked programs are serious problems in the areas of security and portability. As program that downloads and executes automatically on the client computer must be prevented from doing harm.
- It must also be able to run in a variety of different environments and under different operating systems.
- Java solved these problems in an effective and elegant way.

Security

- the code we download might contain virus, Trojan horse, or other harmful code that can gain unauthorized access to system resources.
- For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of computer.
- Java achieved protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.

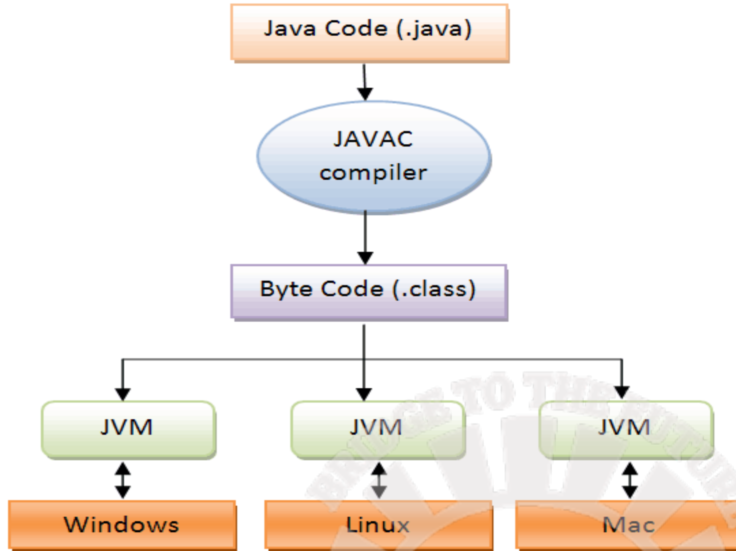
Portability

- different types of computers and operating systems connected to internet
- Java program must be able to run on any computer connected to the Internet,
- The same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet.
- It is not practical to have different versions of the applet for different computers. The *same* code must work on *all* computers.
- Therefore, some means of generating portable executable code was needed. The same mechanism which ensure security also helps in portability.

Java's Magic: The Bytecode

- The key that allows Java to solve both the security and the portability problems is that the output of a Java compiler is not executable code. Rather, it is bytecode.
- *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM).*
- modern programming languages are designed to be compiled into executable code because of performance concerns
- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.
- Once the run-time package exists for a given system, any Java program can run on it.
- the JVM will differ from platform to platform, all understand the same Java bytecode.
- If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution.
- Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

notes4free.in



- The fact that a Java program is executed by the JVM also helps to make it secure.
- Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.
- bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster

Servlets: Java on the Server Side

- A servlet is a small program that executes on the server.
- Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server.
- Servlets are used to create dynamically generated content that is then served to the client.
- For example, an online store might use a servlet to look up the price for an item in a database. The price information is then used to dynamically generate a web page that is sent to the browser.
- Servlets increases performance.
- Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments.

The Java Buzzwords

Simple

- Java was designed to be easy for the professional programmer to learn and use effectively.
- As Java inherits the C/C++ syntax and many of the object-oriented features of C++, its easy to learn.

Object-Oriented

- The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

 notes4free.in

Robust

- the program must execute reliably in a variety of systems. To gain reliability, Java restricts us to find mistakes early in program development.
- As Java is a strictly typed language, it checks code at compile time. also checks code at run time.
- Java programmes behave in a predictable way under diverse conditions is a key feature of Java.
- Programs fail in 2 conditions, memory management mistakes and mishandled exceptional conditio.
- Java virtually eliminates memory management problems by managing memory allocation and deallocation automatically.
- Exceptional conditions(run time errors) in Java is handled well by providing object-oriented exception handling

Multithreaded

- Java programs can do many things simultaneously.
- The Java run-time system supports multiprocess synchronization that enables to construct smoothly running interactive systems.
- Java's easy-to-use approach to multithreading allows to work on specific behavior of the program, not the multitasking subsystem.

Architecture-Neutral

- A central issue for the Java design was that of code longevity and portability.
- As Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. (same program will not execute in different platforms)
- Java Virtual Machine in an attempt to alter this situation. The goal is "write once; run anywhere, any time, forever."

Interpreted and High Performance

- Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine.
- the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

Distributed

- Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.
- Java supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

notes4free.in

Dynamic

- Java programs carry run-time type information that is used to verify and resolve accesses to objects at run time.
- This makes it possible to dynamically link code in a safe manner.
- This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

.

A First Simple Program

```
/* "Example.java". */
class Example
{
        public static void main(String args[])
        {
        System.out.println("This is a simple Java program.");
        }
}
```

.

Compiling the Program in jdk

- the name of the source file should be **Example.java**.
- To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:
  C:\>javac Example.java
- The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program.
- the Java bytecode is the intermediate representation of program that contains instructions the Java Virtual Machine will execute.
- Thus, the output of **javac** is not code that can be directly executed.
- To run the program, you must use the Java application launcher, called **java**.
  C:\>java Example
- When the program is run, the following output is displayed:

This is a simple Java program.

Explaination

- class Example {

This line uses the keyword **class** to declare that a new class is being defined.

- **Example** is an *identifier* that is the name of the class.
- The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}).
- public static void main(String args[]) {
- This line begins the **main( )** method. This is the line at which the program will begin executing. All Java applications begin execution by calling **main( )**.
- The **public** keyword is an *access specifier,* which allows the programmer to control the visibility of class members.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

- When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared.
- **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started.
- The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java Virtual Machine before any objects are made.
- The keyword **void** tells the compiler that **main( )** does not return a value.
- **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**.
- **args** receives any command-line arguments present when the program is executed.
- System.out.println("This is a simple Java program.");
- Output is actually accomplished by the built-in **println( )** method, **println( )** displays the string which is passed to it.
- **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

2. Variables and Data Types

- Variables are locations in memory in which values can be stored. They have a name, a type, and a value.
- Java has three kinds of variables: instance variables, class variables, and local variables.
- Instance variables, are used to define attributes or the state for a particular object.
- Class variables are similar to instance variables, except their values apply to all that class's instances (and to the class itself) rather than having different values for each object.
- Local variables are declared and used inside method(function) definitions,
- Variable declarations consist of a type and a variable name:
  Examples :int myAge; String myName; boolean value;

The Primitive Types

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**.

2.1 Integer types.

Type	*Size*	*Range*
byte	8 bits	—128 to 127
short	16 bits	—32,768 to 32,767
int	32 bits	—2,147,483,648 to 2,147,483,647
long	64bits	—9223372036854775808 to 9223372036854775807

// Compute distance light travels using long variables.

```
class Light
{
        public static void main(String args[])
        {
                int lightspeed;
                long days;
                long seconds;
                long distance;
                lightspeed = 186000;

                days = 1000; // specify number of days here
                seconds = days * 24 * 60 * 60; // convert to seconds
                distance = lightspeed * seconds; // compute distance
                System.out.print("In " + days);
                System.out.print(" days light will travel about ");
                System.out.println(distance + " miles.");
        }
}
```

output:
In 1000 days light will travel about 16070400000000 miles.
Clearly, the result could not have been held in an **int** variable.

2.2 Floating-point

- This is used for numbers with a decimal part.
- There are two floating-point types:
  float (32 bits, single-precision) and double (64bits, double-precision).

```
class Area
{
        public static void main(String args[])
         {
                double pi, r, a;
                r = 10.8; // radius of circle
                pi = 3.1416; // pi, approximately
                a = pi * r * r; // compute area
                System.out.println("Area of circle is " + a);
        }
}
```

Output:
Area of circle is 366.24

2.3 Char

- The char type is used for individual characters. Because Java uses the Unicode character set, the char type has 16 bits of precision, unsigned.

```
class CharDemo
```


notes4free.in

```
{
    public static void main(String args[])
    {
        char ch1, ch2;
        ch1 = 88; // code for X
        ch2 = 'Y';
        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

output:
ch1 and ch2: X Y

```
// char variables behave like integers.
class CharDemo2
{
    public static void main(String args[])
    {
        char ch1;
        ch1 = 'X';
        System.out.println("ch1 contains " + ch1);
        ch1++; // increment ch1
        System.out.println("ch1 is now " + ch1);
    }
}
```

output:
ch1 contains X
ch1 is now Y

2.4 Boolean

- The boolean type can have one of two values, true or false.

```
class BoolTest
{
    public static void main(String args[])
    {
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);
        if(b)
        System.out.println("This is executed.");
        b = false;
        if(b)
```


notes4free.in

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
                System.out.println("This is not executed.");
                System.out.println("10 > 9 is " + (10 > 9));
        }
}
```

output:
b is false
b is true
This is executed.
10 > 9 is true

2.5 Literals

- Literals are used to indicate simple values in Java programs.
- Number Literals
- There are several integer literals.Ex: 4, is a decimal integer literal of type int
- Floating-point literals usually have two parts: the integer part and the decimal part— Ex: 5.677777.
- Boolean Literals:Boolean literals consist of the keywords true and false.
- These keywords can be used anywhere needed a test or as the only possible values for Boolean variables.

2.6 Character Literals

- Character literals are expressed by a single character surrounded by single quotes: 'a', '#', '3', and so on. Characters are stored as 16-bit Unicode characters.

The Java Class Libraries

- **println( )** and **print( )**. these methods are members of the **System** class, which is a class predefined by Java that is automatically included in your programs.
- the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics.

.

Dynamic Initialization of variables.

- Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.
- For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
class DynInit
{
        public static void main(String args[])
        {
                double a = 3.0, b = 4.0;
                // c is dynamically initialized
                double c = Math.sqrt(a * a + b * b);
```

 notes4free.in

```
                System.out.println("Hypotenuse is " + c);
        }
}
```

- **sqrt( )**, is a built in method of the **Math** class.

The Scope and Lifetime of Variables

- Java allows variables to be declared within any block.
- a block is begun with an opening curly brace and ended by a closing curly brace.
- A block defines a *scope*. Thus, each time we start a new block, we are creating a new scope.
- A scope determines what objects are visible to other parts of program.
- It also determines the lifetime of those objects.
- In Java, the two major scopes are those defined by a class and those defined by a method.
- In nested scopes objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.
- To understand the effect of nested scopes, consider the following program:

```
// Demonstrate block scope.
class Scope
{
public static void main(String args[])
{
int x; // known to all code within main
x = 10;
        if(x == 10)
        {
        int y = 20; // known only to this block
                                // x and y both known here.
        System.out.println("x and y: " + x + " " + y);
        x = y * 2;
        }
// y = 100; // Error! y not known here
// x is still known here.
System.out.println("x is " + x);
}
}
```

- a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.
- If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered.
- For example, consider the next program.

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

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

notes4free.in

```
                int x;
                for(x = 0; x < 3; x++)
                {
                        int y = -1; // y is initialized each time block is entered
                        System.out.println("y is: " + y); // this always prints -1
                        y = 100;
                        System.out.println("y is now: " + y);
                }
        }
}
```

output:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

Type Conversion and Casting

- To assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically.
- For example, it is always possible to assign an **int** value to a **long** variable.
- However, not all types are compatible, and thus, not all type conversions are implicitly allowed.
- there is no automatic conversion defined from **double** to **byte**.
- It is still possible to obtain a conversion between incompatible types. We must use a *cast,* which performs an explicit conversion between incompatible types.

Java's Automatic Conversions

- When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

  • The two types are compatible.
  • The destination type is larger than the source type.
- When these two conditions are met, a *widening conversion* takes place.
- Ex, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.
- the numeric types, including integer and floating-point types, are compatible with each other.
- there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other.

notes4free.in

Casting Incompatible Types

- if we want to assign an **int** value to a **byte** variable, This conversion will not be performed automatically, because a **byte** is smaller than an **int**(narrowing conversion).
- To create a conversion between two incompatible types, we must use a cast.
- A *cast* is simply an explicit type conversion.
- It has this general form:

  (*target-type*) *value*
- int a;
  byte b;
  // ...
  b = (byte) a;
- A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*.
- Integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.
- Ex:if the value 1.23 is assigned to an integer, the resulting value will be 1.

```
// Demonstrate casts.
class Conversion
 {
	public static void main(String args[])
	 {
		byte b;
		int i = 257;
		double d = 323.142;
		System.out.println("\nConversion of int to byte.");
		b = (byte) i;
		System.out.println("i and b " + i + " " + b);
		System.out.println("\nConversion of double to int.");
		i = (int) d;
		System.out.println("d and i " + d + " " + i);
		System.out.println("\nConversion of double to byte.");
		b = (byte) d;
		System.out.println("d and b " + d + " " + b);
	}
}
```

Output:
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.

notes4free.in

d and b 323.142 67

Automatic Type Promotion in Expressions

- In the following expression:
```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```
- The result of the intermediate term **a * b** easily exceeds the range of either of its **byte** operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**, or **char** operand to **int** when evaluating an expression. This means that the subexpression **a * b** is performed using integers—not bytes.
- For example, this seemingly correct code causes a problem:
```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```
- In such cases we should use an explicit cast, such as
```
byte b = 50;
b = (byte)(b * 2);
```
which yields the correct value of 100.

The Type Promotion Rules

- First,all **byte**, **short**, and **char** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float,** the entire expression is promoted to **float**. If any of the operands is **double**, the result is **double**.
- The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote
{
	public static void main(String args[])
	 {
		byte b = 42;
		char c = 'a';
		short s = 1024;
		int i = 50000;
		float f = 5.67f;
		double d = .1234;
		double result = (f * b) + (i / c) - (d * s);
		System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
		System.out.println("result = " + result);
	}
}
```


notes4free.in

- Here,double result = (f * b) + (i / c) - (d * s);
- In the first subexpression, **f * b**, **b** is promoted to a **float** and the result of the subexpression is **float**. Next, in the subexpression **i / c**, **c** is promoted to **int**, and the result is of type **int**.
- Then, in **d * s**, the value of **s** is promoted to **double**, and the type of the subexpression is **double**.
- three intermediate values, **float**, **int**, and **double**, are considered. The outcome of **float** plus an **int** is a **float**. Then the resultant **float** minus the last **double** is promoted to **double**, which is the type for the final result of the expression.

Arrays

- An *array* is a group of like-typed variables that are referred to by a common name.
- Aspecific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

- A *one-dimensional array* is, essentially, a list of like-typed variables.
- The general form of a one-dimensional array declaration is
  type array-var = new *type*[*size*];

- Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array,
- *array-var* is the array variable that is linked to the array.
- The elements in the array allocated by **new** will automatically be initialized to zero.
- using **new** ,allocate the memory that will hold the array
- This example allocates a 12-element array of integers and links them to **month_days**.
  int month_days = new int[12];

```
// Demonstrate a one-dimensional array.
class Array
{
        public static void main(String args[])
        {
        int month_days[];
        month_days = new int[12];
        month_days[0] = 31;
        month_days[1] = 28;
        month_days[2] = 31;
        month_days[3] = 30;
        month_days[4] = 31;
        month_days[5] = 30;
        month_days[6] = 31;
        month_days[7] = 31;
        month_days[8] = 30;
        month_days[9] = 31;
        month_days[10] = 30;
```


notes4free.in

```
                    month_days[11] = 31;
                    System.out.println("April has " + month_days[3] + " days.");
                    }
}
```

```
// An improved version of the previous program.
class AutoArray
{
          public static void main(String args[])
          {
                    int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,30, 31 };
                    System.out.println("April has " + month_days[3] + " days.");
          }
}
```

Output: April has 30 days.

Example prog that uses a one-dimensional array to find the average of a set of numbers.

```
class Average
{
          public static void main(String args[]) {
                    double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
                    double result = 0;
                    int i;
                    Chapter 3: Data Types, Variables, and Arrays 51
                    for(i=0; i<5; i++)
                    result = result + nums[i];
                    System.out.println("Average is " + result / 5);
          }
}
```

Multidimensional Arrays

- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- For example, the following declares a twodimensional array variable called **twoD**.

```
int twoD[][] = new int[4][5];
```

```
// Demonstrate a two-dimensional array.

class TwoDArray
{
          public static void main(String args[])
          {
                    int twoD[][]= new int[4][5];
                    int i, j, k = 0;
                    for(i=0; i<4; i++)
                    for(j=0; j<5; j++) {
```


notes4free.in

```
                    twoD[i][j] = k;
                    k++;
          }
          for(i=0; i<4; i++)
           {
                    for(j=0; j<5; j++)
                    System.out.print(twoD[i][j] + " ");
                    System.out.println();
          }
          }
}
```

This program generates the following output:

```
0    1    2    3   4
5    6    7    8   9
10   11   12   13  14
15   16   17   18  19
```

- thefollowing code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

```
// Manually allocate differing size second dimensions.

          class TwoDAgain
          {
                    public static void main(String args[])
                    {
                              int twoD[][] = new int[4][];
                              twoD[0] = new int[1];
                              twoD[1] = new int[2];
                              twoD[2] = new int[3];
                              twoD[3] = new int[4];
                              int i, j, k = 0;

                              for(i=0; i<4; i++)
                                        for(j=0; j<i+1; j++)
                                         {
                                                  twoD[i][j] = k;
                                                  k++;
                                        }

                              for(i=0; i<4; i++)
                                        for(j=0; j<i+1; j++)
                                        {
                                        System.out.print(twoD[i][j] + " ");
                                        System.out.println();
                                        }
                    }
```


notes4free.in

```
        }
This program generates the following output:
0
1 2
3 4 5
6 7 8 9

// Demonstrate a three-dimensional array.

class ThreeDMatrix
{
        public static void main(String args[])
        {
                int threeD[][][] = new int[3][4][5];
                int i, j, k;

                for(i=0; i<3; i++)
                        for(j=0; j<4; j++)
                                for(k=0; k<5; k++)
                                        threeD[i][j][k] = i * j * k;

                for(i=0; i<3; i++)
                {
                        for(j=0; j<4; j++)
                         {
                                for(k=0; k<5; k++){
                                        System.out.print(threeD[i][j][k] + " ");
                                        System.out.println();
                                        }
                        }
                System.out.println();
                }
        }
}
```

This program generates the following output:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

0 2 4 6 8
0 4 8 12 16
0 6 12 18 24

3.Operators

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

The operands of the arithmetic operators must be of a numeric type. we cannot use them on **boolean** types, but we can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

```
// Demonstrate the basic arithmetic operators.

class BasicMath
 {
        public static void main(String args[])
         {
        // arithmetic using integers
        System.out.println("Integer Arithmetic");
        int a = 1 + 1;
        int b = a * 3;
        int c = b / 4;
        int d = c - a;
        int e = -d;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("e = " + e);
        }
```


notes4free.in

```
}
```

When you run this program, you will see the following output:

```
Integer Arithmetic
a = 2
b = 6
c = 1
d = -1
e = 1
```

The Modulus Operator

The modulus operator(**%**), returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following example program demonstrates the **%**:

```
// Demonstrate the % operator.

class Modulus
{
        public static void main(String args[])
         {
        int x = 42;
        double y = 42.25;
        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
        }
}
```

output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators

Operation	Equivalent Operation
a = a + 4;	a += 4;
a = a % 2;	a %= 2;

- the **%=** obtains the remainder of **a**/2 and puts that result back into **a**.

```
class OpEquals
{
        public static void main(String args[])
         {
                int a = 1;
                int b = 2;
                int c = 3;
```


notes4free.in

```
        a += 5;
        b *= 4;
        c += a * b;
        c %= 6;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

The output of this program is shown here:

```
a = 6
b = 8
c = 3
```

Increment and Decrement

- The ++ and the – – are Java's increment and decrement operators.
- The statement: x = x + 1; can be written as x++;
- The statement x = x - 1; is equivalent to x--;
- These operators are unique where they can appear both in *postfix* form and *prefix* form.
- In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.
- In postfix form, the previous value is obtained for use in the expression, and then the operand is modified.
- For example:

```
x = 42;
y = ++x;
```

  In this case, **y** is set to 43 because the increment occurs *before* **x** is assigned to **y**.

- Thus, the line **y = ++x;** is the equivalent of these two statements:

```
x = x + 1;
y = x;
```

- Here,

```
x = 42;
y = x++;
```

- the value of **x** is obtained before the increment operator is executed, so the value of **y** is 42.
- Here, the line **y = x++;** is the equivalent of these two statements:

```
y = x;
x = x + 1;
```

```
class IncDec
{
    public static void main(String args[])
    {
        int a = 1;
        int b = 2;
```


notes4free.in

```
        int c;
        int d;
        c = ++b;
        d = a++;
        c++;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
    }
}
```

The output of this program follows:

```
a = 2
b = 3
c = 4
d = 1
```

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**.

Operator	**Result**
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

- Since the bitwise operators manipulate the bits within an integer,
- Ex:, the **byte** value for 42 in binary is 00101010,
- All of the integer types (except **char**) are signed integers. This means that they can represent negative values as well as positive ones.
- Java uses an encoding known as *two's complement,* which means that negative numbers are represented by inverting (changing 1's to 0's and vice versa) all of the bits in a value, then adding 1 to the result.

notes4free.in

- For example, –42 is represented as
  00101010
  11010101, then adding 1, which results in
  11010110, or –42.

- a **byte** value, zero is represented by 00000000.
  Inverting, its 11111111 adding 1 results in 100000000.
  where –0 is the same as 0,
- 11111111 is the encoding for –1.

The Bitwise Logical Operators

- The bitwise logical operators are **&**, |, ^, and ~.
- the bitwise operators are applied to each individual bit within each operand.
-

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

- Also called the *bitwise complement,* the unary NOT operator, ~, inverts all of the bits of its operand.
- For example:
  00101010 (42)
  11010101 after the NOT operator is applied.

The Bitwise AND

- The AND operator, **&**, produces a 1 bit if both operands are also 1. A zero is produced in all other cases.
- Ex:
  00101010 42
  & 00001111 15
  00001010 10

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

00101010 42
| 00001111 15
00101111 47

notes4free.in

The Bitwise XOR

The XOR operator, **^**, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.

```
  00101010   42
^ 00001111   15
  00100101   37
```

Using the Bitwise Logical Operators

The following program demonstrates the bitwise logical operators:

```
class BitLogic
{
        public static void main(String args[])
         {
        String binary[] = {"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
        "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"};

        int a = 3;               // 0011 in binary
        int b = 6;                // 0110 in binary
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        int f = (~a & b) | (a & ~b);
        int g = ~a & 0x0f;

System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);

System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
System.out.println(" a^b = " + binary[e]);
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);
}
}
```

Here is the output from this program:

```
a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
```

 notes4free.in

~a = 1100

The Left Shift

- The left shift operator, **<<**, shifts all of the bits in a value to the left a specified number of times.
- It has this general form: *value << num*
- Here, *num* specifies the number of positions to left-shift the value in *value.*
- That is, the **<<**moves all of the bits in the specified value to the left by the number of bit positions specified by *num.*
- For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.
- This means that when a left shift is applied to an **int** operand, bits are lost once they are shifted past bit position 31.
- If the operand is a **long**, then bits are lost after bit position 63.
- Java's automatic type promotions produce unexpected results when you are shifting **byte** and **short** values.
- **byte** and **short** values are promoted to **int** when an expression is evaluated. The result of such an expression is also an **int**. This means that the outcome of a left shift on a **byte** or **short** value will be an **int**,

```
class ByteShift
 {
	public static void main(String args[])
	{
		byte a = 64, b;
		int i;
		i = a << 2;
		b = (byte) (a << 2);
		System.out.println("Original value of a: " + a);
		System.out.println("i and b: " + i + " " + b);
	}
}
```

The output generated by this program is shown here:
Original value of a: 64
i and b: 256 0

- Since **a** is promoted to **int** for the purposes of evaluation, left-shifting the value 64 (0100 0000) twice results in **i** containing the value 256 (1 0000 0000). However, the value in **b** contains 0 because after the shift, the low-order byte is now zero. Its only 1 bit has been shifted out.

```
// Left shifting as a quick way to multiply by 2.
class MultByTwo
{
	public static void main(String args[])
	 {
```

```
                int i;
                int num = 0xFFFFFFE;
                for(i=0; i<4; i++)
                 {
                num = num << 1;
                System.out.println(num);
                }
        }
}
```

The program generates the following output:
536870908
1073741816
2147483632
-32
The starting value was carefully chosen so that after being shifted left 4 bit positions, it would produce –32. As you can see, when a 1 bit is shifted into bit 31, the number is interpreted as negative.

The Right Shift

- The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times.
- Its general form is shown here: *value >> num*
- Here, *num* specifies the number of positions to right-shift the value in *value.* That is, the **>>** moves all of the bits in the specified value to the right the number of bit positions specified by *num.*
- int a = 32;
  a = a >> 2; // a now contains 8
- When a value has bits that are "shifted off," those bits are lost.
- For example, the value 35 is shifted to the right two positions, which causes the two low-order bits to be lost, resulting again in **a** being set to 8.

  ```
  int a = 35;
  a = a >> 2; // a still contains 8
  ```
- Looking at the same operation in binary shows more clearly how this happens:

  ```
  00100011        35
  >> 2
  00001000        8
  ```
- When we are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called *sign extension* and serves to preserve the sign of negative numbers when we shift them right.
- For example,

  ```
  11111000 –8
  >>1
  11111100 –4
  ```

class HexByte

notes4free.in

```
{
        Public static void main(String args[])
        {
        byte a=-8;
        byte b = (byte) (a>>1);
        System.out.println("Right shift value is" +b);
        }
}
```

Here is the output of this program:
b = -4

The Unsigned Right Shift

- the >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value.
- Java's unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit.
- The following code fragment demonstrates the >>>.
- Here, **a** is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets **a** to 255.

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form :

```
11111111 11111111 11111111 11111111        –1
>>>24
00000000 00000000 00000000 11111111        255
```

Bitwise Operator Compound Assignments

- All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combines the assignment with the bitwise operation.

```
a = a >> 4;
a >>= 4;
```

- Likewise, the following two statements are equivalent:

```
a = a | b;
a |= b;
```

```
class OpBitEquals
 {
        public static void main(String args[])
         {
                int a = 1;
                int b = 2;
                int c = 3;
                a |= 4;
                b >>= 1;
```


notes4free.in

```
        c <<= 1;
        a ^= c;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        }
}
```

The output of this program is shown here:

a = 3
b = 1
c = 6

Relational Operators

- The *relational operators* determine the relationship that one operand has to the other.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

- The outcome of these operations is a **boolean** value.
- only integer, floating-point, and character operands may be compared to see which is greater or less than the other.
- int a = 4;
  int b = 1;
  boolean c = a < b;
  In this case, the result of **a<b** (which is **false**) is stored in **c**.

Boolean Logical Operators

- The Boolean logical operators operate only on **boolean** operands. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment

 notes4free.in

==	Equal to
!=	Not equal to
?:	Ternary if-then-else

A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

```
class BoolLogic
{
	public static void main(String args[])
	{
		boolean a = true;
		boolean b = false;
		boolean c = a | b;
		boolean d = a & b;
		boolean e = a ^ b;
		boolean f = (!a & b) | (a & !b);
		boolean g = !a;
		System.out.println(" a = " + a);
		System.out.println(" b = " + b);
		System.out.println(" a|b = " + c);
		System.out.println(" a&b = " + d);
		System.out.println(" a^b = " + e);
		System.out.println("!a&b|a&!b = " + f);
		System.out.println(" !a = " + g);
	}
}
```

```
a = true
b = false
a|b = true
a&b = false
a^b = true
a&b|a&!b = true
!a = false
```

Short-Circuit Logical Operators

- There are secondary versions of the Boolean AND and OR operators, and are known as *short-circuit* logical operators.
- the OR operator results in **true** when **A** is **true**, no matter what **B** is.
- Similarly, the AND operator results in **false** when **A** is **false**, no matter what **B** is.(therefore there is no need to evaluate the second operand.)
- Short circuit logical operators are the || and **&&** f

The Assignment Operator

- The *assignment operator* is the single equal sign, =.
- It has this general form:

  var = expression;
- Here, the type of *var* must be compatible with the type of *expression.*
- int x, y, z;
  x = y = z = 100; // set x, y, and z to 100
- This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement.

.

The ? Operator

- Java provides *ternary* (three-way) *operator* that can replace certain types of if-then-else statements.
- The **?** has this general form:

  expression1 **?** *expression2* **:** *expression3*
- Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated.

.

```
class Ternary
{
        public static void main(String args[])
        {
                int a=5,b=10;
                int c= a>b? a: b;
                System.out.println("bigger number is " +c);
        }
}
```

output :

bigger number is 10.

Operator Precedence

Highest			
( )	[ ] .		
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			

?:
= op=
Lowest

Control Statements

- A programming language uses *control* statements to cause the flow of execution to advance and branch into different part of a program.
- Java's program control statements can be put into the following categories:
  - selection,
  - iteration, and
  - jump.
- *Selection* statements allow program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- *Iteration* statements enable program execution to repeat one or more statements.
- *Jump* statements allow program to execute in a nonlinear fashion.

Java's Selection Statements

- Java supports two selection statements: **if** and **switch**.
- These statements allow us to control the flow of program's execution based upon conditions known only during run time.

if

- **if** statement is Java's conditional branch statement.
- General form of **if** statement:
  if (*condition*) *statement1*;
  else *statement2*;
- Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*).
- The *condition* is any expression that returns a **boolean** value.
- The **else** clause is optional.
- If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed.
- For example, :

```
int a, b;
// ...
if(a < b)
 a = 0;
else
b = 0;
```

Nested ifs

- A *nested* **if** is an **if** statement that is the target of another **if** or **else.**
- an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**.

 notes4free.in

- Here is an example:

```
if(i == 10)
{
    if(j < 20) a = b;
        if(k > 100)      // this if is
         c = d;
        else
        a = c;           // associated with this else
}
else
 a = d;                  // this else refers to if(i == 10)
```

- As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**)
- . The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

The if-else-if Ladder

- A common programming construct that is based upon a sequence of nested **if**s is the *if-else-if ladder*.
- General form:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
...
else
statement;
```

- The **if** statements are executed from the top down.
- As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed.
- If none of the conditions is true, then the final **else** statement will be executed.
- The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed.
- If there is no final **else** and all other conditions are **false**, then no action will take place.

```
class IfElse
 {
     public static void main(String args[])
      {
          int month = 4; // April
          String season;

          if(month == 12 || month == 1 || month == 2)
```

```
                season = "Winter";
                else if(month == 3 || month == 4 || month == 5)
                season = "Spring";
                else if(month == 6 || month == 7 || month == 8)
                season = "Summer";
                else if(month == 9 || month == 10 || month == 11)
                season = "Autumn";
                else
                season = "Bogus Month";
                System.out.println("April is in the " + season + ".");
        }
}
```

output:
April is in the Spring.

switch

- The **switch** statement is Java's multiway branch statement.
- It provides an easy way to dispatch execution to different parts of code based on the value of an expression.
- It provides a better alternative than a large series of **if-else-if** statements.
- Here is the general form of a **switch** statement:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
...
case valueN:
// statement sequence
break;
default:
// default statement sequence
}
```

- The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression.
- Each **case** value must be a unique literal (that is, it must be a constant, not a variable).
- Duplicate **case** values are not allowed.
- The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed.
- If none of the constants matches the value of the expression, then the **default** statement is executed.
- However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

- The **break** statement is used inside the **switch** to terminate a statement sequence.

```
class SampleSwitch
{
        public static void main(String args[])
         {
                for(int i=0; i<6; i++)
                switch(i)
                 {
                case 0:
                System.out.println("i is zero.");
                break;
                case 1:
                System.out.println("i is one.");
                break;
                case 2:
                System.out.println("i is two.");
                break;
                case 3:
                System.out.println("i is three.");
                break;
                default:
                System.out.println("i is greater than 3.");
                }
        }
}
 output:
                        i is zero.
                        i is one.
                        i is two.
                        i is three.
                        i is greater than 3.
                        i is greater than 3.
```

- The **break** statement is optional. If we omit the **break**, execution will continue on into the next **case**. It is sometimes desirable to have multiple **case**s without **break** statements between them.

```
class MissingBreak
{
        public static void main(String args[])
         {
                for(int i=0; i<12; i++)
                switch(i)
                 {
                case 0:
```


notes4free.in

```
                case 1:
                case 2:
                case 3:
                case 4:
                System.out.println("i is less than 5");
                break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                System.out.println("i is less than 10");
                break;
                default:
                System.out.println("i is 10 or more");
                }
        }
}
```

output:
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more

```
class Switch
{
        public static void main(String args[])
        {
                int month = 4;
                String season;
                switch (month)
                 {
                case 12:
                case 1:
                case 2:
                season = "Winter";
                break;
                case 3:
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

 notes4free.in

```
                case 4:
                case 5:
                season = "Spring";
                break;
                case 6:
                case 7:
                case 8:
                season = "Summer";
                break;
                case 9:
                case 10:
                case 11:
                season = "Autumn";
                break;
                default:
                season = "Bogus Month";
                }
                System.out.println("April is in the " + season + ".");
        }
}
```

Nested switch Statements

- **switch can be used** as part of an outer **switch**. This is called a *nested* **switch**.
- Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**.

```
switch(count)
{
case 1:
                switch(target)
                 {                              // nested switch
                case 0:
                System.out.println("target is zero");
                break;
                case 1: // no conflicts with outer switch
                System.out.println("target is one");
                break;
                }
                break;
case 2: // ...
```

- Here, the **case 1:** statement in the inner switch does not conflict with the **case 1:** statement in the outer switch.
- The **count** variable is only compared with the list of cases at the outer level.
- If **count** is 1, then **target** is compared with the inner list cases.

notes4free.in

In summary, there are three important features of the **switch** statement to note:

• The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression.
• No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.
• A **switch** statement is usually more efficient than a set of nested **if**s.

Iteration Statements

- Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops.*
- a loop repeatedly executes the same set of instructions until a termination condition is met.

while

- The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true.
- Here is its general form:

```
while(condition) {
// body of loop
}
```

- The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true.
- When *condition* becomes false, control passes to the next line of code immediately following the loop.

```
class While
{
        public static void main(String args[])
         {
                int n = 5;
                while(n > 0)
                 {
                System.out.println("tick " + n);
                n--;
                }
        }
}
```

When you run this program, it will "tick" five times:

```
tick 5
tick 4
tick 3
tick 2
tick 1
```

 notes4free.in

- Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with.
- For example, in the following fragment, the call to **println( )** is never executed:

```
int a = 10, b = 20;
while(a > b)
System.out.println("This will not be displayed");
```

- The body of the **while** (or any other of Java's loops) can be empty. This is because a *null Statement* is syntactically valid in Java.
- For example,

```
class NoBody
{
    public static void main(String args[])
    {
        int i, j;
        i = 100;
        j = 200;
        // find midpoint between i and j
        while(++i < --j) ; // no body in this loop
        System.out.println("Midpoint is " + i);
    }
}
```

This program finds the midpoint between **i** and **j**.

output: Midpoint is 150

do-while

- if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all.
- sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with.
- In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning.
- Java supplies a loop that does just that: the **do-while**.
- The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do {
// body of loop
} while (condition);
```

- Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.

```
class DoWhile
{
    public static void main(String args[])
    {
        int n = 5;
```

```
                do {
                System.out.println("tick " + n);
                n--;
                } while(n > 0);
        }
}

class Menu
{
        public static void main(String args[]) throws java.io.IOException
        {
                char choice;
                        do
                         {
                        System.out.println("Help on:");
                        System.out.println(" 1. good");
                        System.out.println(" 2. better");
                        System.out.println(" 3. best");
                        System.out.println("4. Excellent");
                        System.out.println("Choose one:");
                        choice = (char) System.in.read();
                        } while( choice < '1' || choice > '4');
                System.out.println("\n");
                switch(choice)
                 {
                        case '1':       System.out.println("Good");
                                        break;
                        case '2':       System.out.println("better");
                                        break;
                        case '3':       System.out.println("best");
                                        break;
                        case '4':       System.out.println("Excellent");
                                        break;
                }
        }
}
```

Here is a sample run produced by this program:

```
Help on:
1. good
2. better
3. best
4. Excellent

Choose one:
4
Excellent.
```


notes4free.in
BRIDGE TO THE FUTURE CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

for

- there are two forms of the **for** loop.
- The first is the traditional form that has been in use since the original version of Java.
- The second is the new "for-each" form.
- general form of the traditional **for** statement:

```
for(initialization; condition; iteration) {
// body
}
```

```
class ForTick
{
	public static void main(String args[])
	 {
		int n;
		for(n=10; n>0; n--)
		System.out.println("tick " + n);
	}
}
```

Declaring Loop Control Variables Inside the for Loop

- Often the variable that controls a **for** loop is only needed for the purposes of the loop and is not used elsewhere.
- When this is the case, it is possible to declare the variable inside the initialization portion of the **for**.
- For example,the loop control variable **n** is declared as an **int** inside the **for**:

```
class ForTick
 {
	public static void main(String args[])
	 {
		for(int n=10; n>0; n--)
		System.out.println("tick " + n);
	}
}
```

```
class FindPrime
{
	public static void main(String args[])
	 {
		int num;
		boolean isPrime = true;
		num = 14;
		for(int i=2; i <= num/i; i++)
		 {
			if((num % i) == 0)
			{
			isPrime = false;
```


notes4free.in

```
                    break;
                    }
            }
            if(isPrime)
             System.out.println("Prime");
            else
            System.out.println("Not Prime");
        }
}
```

Using the Comma

- There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop.
- For example, consider the loop in the following program:

```
class Comma
{
        public static void main(String args[])
         {
                int a, b;
                for(a=1, b=4; a<b; a++, b--)
                 {
                        System.out.println("a = " + a);
                        System.out.println("b = " + b);
                }
        }
}
```

- the initialization portion sets the values of both **a** and **b**. The two comma separated statements in the iteration portion are executed each time the loop repeats.
- output:

```
a = 1
b = 4
a = 2
b = 3
```

Some for Loop Variations

- The **for** loop supports a number of variations that increase its power and applicability.

```
class ForVar
 {
        public static void main(String args[])
         {
                int i;
                boolean done = false;
                i = 0;
                for( ; !done; )
                 {
```


notes4free.in

```
                    System.out.println("i is " + i);
                    if(i == 10) done = true;
                    i++;
                }
            }
        }
```

- Here, the initialization and iteration expressions have been moved out of the **for**. Thus, parts of the **for** are empty.

The For-Each Version of the for Loop

- The advantage of this approach is that no new keyword is required, and no preexisting code is broken.
- The for-each style of **for** is also referred to as the *enhanced* **for** loop.
- The general form for-each version of the **for** is shown here:
  for(*type itr-var : collection*) *statement-block*
- Here, *type* specifies the type
- *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by *collection*.
- There are various types of collections that can be used with the **for**, but the only type used here is the array..

```
class ForEach
{
    public static void main(String args[])
     {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;

        for(int x : nums)
        {
            System.out.println("Value is: " + x);
            sum += x;
        }
    System.out.println("Summation: " + sum);
    }
}
```

The output from the program is shown here.
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6

notes4free.in

```
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

- the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement.

```
class ForEach2
 {
        public static void main(String args[])
        {
                int sum = 0;
                int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

                for(int x : nums)
                 {
                        System.out.println("Value is: " + x);
                        sum += x;
                        if(x == 5) break; // stop the loop when 5 is obtained
                }
                System.out.println("Summation of first 5 elements: " + sum);
                }
}
```

output :

```
                Value is: 1
                Value is: 2
                Value is: 3
                Value is: 4
                Value is: 5
                Summation of first 5 elements: 15
```

Nested Loops

- Java allows loops to be nested. That is, one loop may be inside another.

```
class Nested
{
        public static void main(String args[])
         {
                int i, j;
                for(i=0; i<10; i++)
                 {
                        for(j=i; j<10; j++)
                        {
```

 notes4free.in

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

The output produced by this program is shown here:

```
..........
.........
........
.......
......
.....
....
...
..
.
```

Jump Statements

- Java supports three jump statements: **break**, **continue**, and **return**.
- These statements transfer control to another part of your program..

Using break

- In Java, the **break** statement has three uses.
- First, as you have seen, it terminates a statement sequence in a **switch** statement.
- Second, it can be used to exit a loop.
- Third, it can be used as a "civilized" form of goto.

Using break to Exit a Loop

- By using **break**, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.
- When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

```
class BreakLoop
{
    public static void main(String args[])
    {
        for(int i=0; i<100; i++)
        {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
```

}

This program generates the following output:
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.

- We can Use break to exit a while loop.
- When used inside a set of nested loops, the **break** statement will only break out of the innermost loop.

```
class BreakLoop3
{
        public static void main(String args[])
        {
                for(int i=0; i<3; i++)
                {
                                System.out.print("Pass " + i + ": ");
                                for(int j=0; j<100; j++)
                                {
                                        if(j == 10) break; // terminate loop if j is 10
                                        System.out.print(j + " ");
                                }
                System.out.println();
                }
        System.out.println("Loops complete.");
        }
}
```

This program generates the following output:
Pass 0: 0 1 2 3 4 5 6 7 8 9
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.

Using break as a Form of Goto

- the **break** statement can also be employed by itself to provide a “civilized” form of the goto statement.

notes4free.in

- Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner.
- The general form of the labeled **break** statement is shown here:
  break *label*;
- Most often, *label* is the name of a label that identifies a block of code. When this form of **break** executes, control is transferred out of the named block.

```
class Break
 {
	public static void main(String args[])
	{
		boolean t = true;
		first: {
			second: {
				third: {
					System.out.println("Before the break.");
					if(t) break second; // break out of second block
					System.out.println("This won't execute");
				}
			System.out.println("This won't execute");
			}
		System.out.println("This is after second block.");
		}
	}
}
output:
Before the break.
This is after second block.
```

- One of the most common uses for a labeled **break** statement is to exit from nested loops.

```
class BreakLoop4
 {
	public static void main(String args[])
	 {
		outer: for(int i=0; i<3; i++)
		{
			System.out.print("Pass " + i + ": ");
			for(int j=0; j<100; j++)
			 {
				if(j == 10) break outer; // exit both loops
				System.out.print(j + " ");
			}
		System.out.println("This will not print");
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
            }
        System.out.println("Loops complete.");
        }
}
```

This program generates the following output:
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.

Using continue

- Here the loop will skip the execution of a particular iteration upen certain condition and continue to execute further iteration.

```
class Continue
{
        public static void main(String args[])
         {
                for(int i=0; i<10; i++)
                 {
                        if (i == 2) continue;
                        System.out.print(i + " ");
                }
        }
}
 output :
                0 1 3 4 5 6 7 8 9
```

```
class ContinueLabel
{
        public static void main(String args[])
         {
                outer: for (int i=0; i<10; i++)
                 {
                        for(int j=0; j<10; j++)
                        {
                                if(j > i)
                                 {
                                System.out.println();
                                continue outer;
                                }
                        System.out.print(" " + (i * j));
                        }
                }
        System.out.println();
        }
}
```

The **continue** statement in this example terminates the loop counting **j** and continues with

notes4free.in

the next iteration of the loop counting **i**. Here is the output of this program:

0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81

return

- The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main( )**.

```
class Return
{
        public static void main(String args[])
        {
                boolean t = true;
                System.out.println("Before the return.");
                if(t) return; // return to caller
                System.out.println("This won't execute.");
        }
}
```

output:

Before the return.

- As you can see, the final **println( )** statement is not executed. As soon as **return** is executed, control passes back to the caller.

notes4free.in

Introducing Classes

- Class defines the shape and nature of an object.
- class forms the basis for object-oriented programming in Java.
- Any concept can be implemented in a Java program must be encapsulated within a class.

Class Fundamentals

- a class defines a new data type. Once defined, this new type can be used to create objects of that type.
- Thus, a class is a *template* for an object, and an object is an *instance* of a class.

The General Form of a Class

- class specifies the data that it contains and the code that operates on that data.
- While very simple classes may contain only code or only data, most real-world classes contain both.
- A class is declared by use of the **class** keyword.
- A simplified general form of a **class** definition is shown here:

```
class classname
 {
        type instance-variable1;
        type instance-variable2;
        // ...
        type instance-variableN;

        type methodname1(parameter-list) {
        // body of method
        }

        type methodname2(parameter-list) {
        // body of method
        }
        // ...

        type methodnameN(parameter-list) {
        // body of method
        }
}
```

- The data, or variables, defined within a **class** are called *instance variables.*
- The code is contained within *methods.*
- Collectively, the methods and variables defined within a class are called *members* of the class.
- Thus the methods that determine how a class' data can be used.
- each object of the class contains its own copy of these variables.
- Thus, the data for one object is separate and unique from the data for another.

notes4free.in

A Simple Class

- Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**.

```
class Box
 {
        double width;
        double height;
        double depth;
}

class BoxDemo2
{
        public static void main(String args[])
         {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;

                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;

                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;

                // compute volume of first box
                vol = mybox1.width * mybox1.height * mybox1.depth;
                System.out.println("Volume is " + vol);

                // compute volume of second box
                vol = mybox2.width * mybox2.height * mybox2.depth;
                System.out.println("Volume is " + vol);
                }
}
```

output:

```
                Volume is 3000.0
                Volume is 162.0
```

- **mybox1**'s data is completely separate from the data contained in **mybox2**.

Declaring Objects

- when a class is created , we are creating a new data type.
- We can use this type to declare objects of that type.
- However, obtaining objects of a class is a two-step process.

notes4free.in

- First, we must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
- Second, we must acquire an actual, physical copy of the object and assign it to that variable by using the **new** operator.
- The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it
- Box mybox = new Box();
  This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

Assigning Object Reference Variables

```
Box b1 = new Box();
Box b2 = b1;
```

- **b1** and **b2** will both refer to the *same* object.
- The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**.
- Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.
- Although **b1** and **b2** both refer to the same object, they are not linked in any other way.
- For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**.
- For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

  Here, **b1** has been set to **null**, but **b2** still points to the original object.

Introducing Methods

- classes usually consist of two things: instance variables and methods.
- This is the general form of a method:

```
type name(parameter-list) {
// body of method
}
```

- Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that we create.
- If the method does not return a value, its return type must be **void**.
- The name of the method is specified by *name.*
- . The *parameter-list* is a sequence of type and identifier pairs separated by commas.

```
class Box
{
	double width;
	double height;
	double depth;
	//
	void volume()
	 {
		System.out.print("Volume is ");
		System.out.println(width * height * depth);
	}
}

class BoxDemo3
{
	public static void main(String args[])
	 {
		Box mybox1 = new Box();
		Box mybox2 = new Box();

		mybox1.width = 10;
		mybox1.height = 20;
		mybox1.depth = 15;

		mybox2.width = 3;
		mybox2.height = 6;
		mybox2.depth = 9;

		// display volume of first box
		mybox1.volume();

		// display volume of second box
		mybox2.volume();
		}
}
```

This program generates the following output, which is the same as the previous version.

Volume is 3000.0
Volume is 162.0

Returning a Value

```
class Box
 {
	double width;
	double height;
	double depth;
```


notes4free.in

```
        // compute and return volume
        double volume()
        {
        return width * height * depth;
        }
}
class BoxDemo4
{
        public static void main(String args[])
         {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;

                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;

                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;

                // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);

                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
                }
}
```

Adding a Method That Takes Parameters

- Parameters allow a method to be generalized.
- That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.
- Here is a method that returns the square of the number 10:

```
int square()
{
return 10 * 10;
}
```

- While this method does, indeed, return the value of 10 squared, its use is very limited.
- However, if we modify the method so that it takes a parameter, as shown next, then we can make **square( )** much more useful.

notes4free.in

```
int square(int i)
{
return i * i;
}
```

- Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.

```
// This program uses a parameterized method.

class Box
 {
        double width;
        double height;
        double depth;

        double volume()
         {
                return width * height * depth;
        }

        void setDim(double w, double h, double d)
        {
        width = w;
        height = h;
        depth = d;
        }
}

class BoxDemo5
{
        public static void main(String args[])
        {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;

                mybox1.setDim(10, 20, 15);
                mybox2.setDim(3, 6, 9);

                // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);

                // get volume of second box
                vol = mybox2.volume();
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

 notes4free.in

```
        System.out.println("Volume is " + vol);
    }
}
```

Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- Thus automatic initialization is performed through the use of a constructor.
- A *constructor* initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself.

```
class Box
 {
    double width;
    double height;
    double depth;

    Box()
    {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    double volume()
    {
        return width * height * depth;
    }
}
class BoxDemo6
{
    public static void main(String args[])
     {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // get volume of first box
        vol = mybox1.volume();
```

 notes4free.in

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
                System.out.println("Volume is " + vol);

                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

Output:

```
                Constructing Box
                Constructing Box
                Volume is 1000.0
                Volume is 1000.0
```

- both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created.
- Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume.

Parameterized Constructors

- While the **Box( )** constructor in the preceding example initializes with value 10.all boxes have the same dimensions.
- **Box** objects of various dimensions can be assigned by using parameterized constructor.

```
class Box
{
        double width;
        double height;
        double depth;

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

        double volume()
        {
                return width * height * depth;
        }
}
class BoxDemo7
 {
        public static void main(String args[])
        {
```


notes4free.in

```
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);
        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
    }
}
```

output :

```
        Volume is 3000.0
        Volume is 162.0
```

The this Keyword

- **this** can be used inside any method to refer to the *current* object.
- That is, **this** is always a reference to the object on which the method was invoked.

```
// A redundant use of this.
        Box(double w, double h, double d)
         {
            this.width = w;
            this.height = h;
            this.depth = d;
        }
```

Instance Variable Hiding

- it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.

```
// Use this to resolve name-space collisions.
    Box(double width, double height, double depth)
     {
        this.width = width;
        this.height = height;
        this.depth = depth;
    }
```

 notes4free.in

Garbage Collection

- Since objects are dynamically allocated by using the **new** operator, how such objects are destroyed and their memory released for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator.
- Java handles deallocation automatically.
- The technique that accomplishes this is called *garbage collection.*
- when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- Garbage collection only occurs sporadically (if at all) during the execution of program.

The finalize( ) Method

- an object will need to perform some action when it is destroyed.
- if an object is holding some non-Java resource such as a file handle or character font, then we might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called *finalization.*
- By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- To add a finalizer to a class, simply define the **finalize( )** method.
- The Java run time calls that method whenever it is about to recycle an object of that class.
- Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed.
- The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

- Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class.
- **finalize( )** is only called just prior to garbage collection.
- It is not called when an object goes out-of-scope

A Stack Class

- Stacks are controlled through two operations traditionally called *push* and *pop.*
- To put an item on top of the stack, we will use push.
- To take an item off the stack, we will use pop.
- Here is a class called **Stack** that implements a stack for integers:

```
class Stack
{
	int stck[] = new int[10];
	int tos;
```


notes4free.in

```
        Stack()
         {
                tos = -1;
        }

        void push(int item)
         {
                if(tos==9)
                        System.out.println("Stack is full.");
                else
                        stck[++tos] = item;
        }

        int pop()
        {
                if(tos < 0)
                {
                System.out.println("Stack underflow.");
                return 0;
                }
                else
                return stck[tos--];
        }
}
class TestStack
{
        public static void main(String args[])
         {
                Stack mystack1 = new Stack();
                Stack mystack2 = new Stack();

                for(int i=0; i<10; i++) mystack1.push(i);
                for(int i=10; i<20; i++) mystack2.push(i);

                System.out.println("Stack in mystack1:");
                for(int i=0; i<10; i++)
                System.out.println(mystack1.pop());
                System.out.println("Stack in mystack2:");
                for(int i=0; i<10; i++)
                System.out.println(mystack2.pop());
                }
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
```

7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10

Inheritance

- One class can acquire the properties of another class.
- a class that is inherited is called a *superclass.*
- The class that does the inheriting is called a *subclass.* Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

Inheritance Basics

- To inherit a class, simply incorporate the definition of one class into another by using the **extends** keyword.
- The following program creates a superclass called **A** and a subclass called **B**.the keyword **extends** is used to create a subclass of **A**.

```
// Create a superclass.

class A
{
        int i, j;
        void showij()
         {
                System.out.println("i and j: " + i + " " + j);
        }
}

// Create a subclass by extending class A.
class B extends A
{
        int k;
        void showk()
         {
                System.out.println("k: " + k);
        }
        void sum()
         {
                System.out.println("i+j+k: " + (i+j+k));
        }
}

class SimpleInheritance
 {
        public static void main(String args[])
         {
                A superOb = new A();
                B subOb = new B();
                // The superclass may be used by itself.
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

 notes4free.in

```
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb: ");
        superOb.showij();
        System.out.println();

        /* The subclass has access to all public members of its superclass. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;

        System.out.println("Contents of subOb: ");
        subOb.showij();
        subOb.showk();
        System.out.println();

        System.out.println("Sum of i, j and k in subOb:");
        subOb.sum();
        }
}
```

output:

```
        Contents of superOb:
        i and j: 10 20
        Contents of subOb:
        i and j: 7 8
        k: 9
        Sum of i, j and k in subOb:
        i+j+k: 24
```

- the subclass **B** includes all of the members of its superclass, **A**. This is why **subOb** can access **i** and **j** and call **showij( )**. Also, inside **sum( )**, **i** and **j** can be referred to directly, as if they were part of **B**.
- Even though **A** is a superclass for **B**, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself.
- a subclass can be a superclass for another subclass.
- The general form of a **class** declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name
{
        // body of class
}
```

- Java does not support the inheritance of multiple superclasses into a single subclass.
- But a subclass can become a superclass of another subclass.
- However, no class can be a superclass of itself.

notes4free.in

Member Access and Inheritance

- Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.

```
// Create a superclass.

class A
{
        int i;                  // public by default
        private int j;          // private to A

        void setij(int x, int y)
         {
                i = x;
                j = y;
        }
}

// A's j is not accessible here.

class B extends A
{
        int total;

        void sum()
         {
                total = i + j; // ERROR, j is not accessible here
        }
}

class Access
 {
        public static void main(String args[])
         {
                B subOb = new B();
                subOb.setij(10, 12);
                subOb.sum();
                System.out.println("Total is " + subOb.total);
        }
}
```

- This program will not compile because the reference to **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

 notes4free.in

A More Practical Example

- the **Box** class developed will be extended to include a fourth component called **weight**.
- Thus, the new class will contain a box's width, height, depth, and weight.

```
// This program uses inheritance to extend Box.

class Box
{
	double width;
	double height;
	double depth;

	// construct clone of an object
	Box(Box ob)
	{					// pass object to constructor
		width = ob.width;
		height = ob.height;
		depth = ob.depth;
	}

	// constructor used when all dimensions specified
	Box(double w, double h, double d)
	 {
		width = w;
		height = h;
		depth = d;
	}

	// constructor used when no dimensions specified
	Box()
	{
		width = -1; // use -1 to indicate
		height = -1; // an uninitialized
		depth = -1; // box
	}

	// constructor used when cube is created
	Box(double len)
	 {
		width = height = depth = len;
	}

	// compute and return volume
	double volume()
	 {
```


notes4free.in

```
        return width * height * depth;
        }
}

// Here, Box is extended to include weight.
class BoxWeight extends Box
 {
        double weight; // weight of box

        // constructor for BoxWeight
        BoxWeight(double w, double h, double d, double m) {
        width = w;
        height = h;
        depth = d;
        weight = m;
        }
}

class DemoBoxWeight
{
        public static void main(String args[])
        {
                BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
                BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
                double vol;

                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                System.out.println("Weight of mybox1 is " + mybox1.weight);
                System.out.println();

                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
                System.out.println("Weight of mybox2 is " + mybox2.weight);
        }
}
```

output:

```
                Volume of mybox1 is 3000.0
                Weight of mybox1 is 34.3
                Volume of mybox2 is 24.0
                Weight of mybox2 is 0.076
```


notes4free.in

- the following class inherits **Box** and adds a color attribute:

```
// Here, Box is extended to include color.
class ColorBox extends Box
{
        int color; // color of box

        ColorBox(double w, double h, double d, int c)
        {
                width = w;
                height = h;
                depth = d;
                color = c;
        }
}
```

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.

```
class RefDemo
{
        public static void main(String args[])
         {
                BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
                Box plainbox = new Box();
                double vol;

                vol = weightbox.volume();
                System.out.println("Volume of weightbox is " + vol);
                System.out.println("Weight of weightbox is " + weightbox.weight);
                System.out.println();

                // assign BoxWeight reference to Box reference
                plainbox = weightbox;
                vol = plainbox.volume();                // OK, volume() defined in Box
                System.out.println("Volume of plainbox is " + vol);

                /* The following statement is invalid because plainbox does not define a weight
                member. */
                // System.out.println("Weight of plainbox is " + plainbox.weight);
                }
}
```

- Here, **weightbox** is a reference to **BoxWeight** objects, and **plainbox** is a reference to **Box** objects.

notes4free.in

- Since **BoxWeight** is a subclass of **Box**, it is permissible to assign **plainbox** a reference to the **weightbox** object.

Using super

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
- **super** has two general forms.
  - The first calls the superclass' constructor.
  - The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

- Asubclass can call a constructor defined by its superclass by use of the following form of **super**:

  super(*arg-list*);
- Here, *arg-list* specifies any arguments needed by the constructor in the superclass.
- **super( )** must always be the first statement executed inside a subclass' constructor.

```
// BoxWeight now uses super to initialize its Box attributes.

class BoxWeight extends Box
{
double weight;

        BoxWeight(double w, double h, double d, double m)
         {
                super(w, h, d); // call superclass constructor
                weight = m;
        }
}
```

- Here, **BoxWeight( )** calls **super( )** with the arguments **w**, **h**, and **d**. This causes the **Box( )** constructor to be called, which initializes **width**, **height**, and **depth** using these values.

```
class Box
 {
        private double width;
        private double height;
        private double depth;

        // construct clone of an object

        Box(Box ob)
        {
        width = ob.width;
```


notes4free.in

```
        height = ob.height;
        depth = ob.depth;
        }

        // constructor used when all dimensions specified
        Box(double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }

        // constructor used when no dimensions specified
        Box()
         {
                width = -1; // use -1 to indicate
                height = -1; // an uninitialized
                depth = -1; // box
        }

        // constructor used when cube is created
        Box(double len)
        {
                width = height = depth = len;
        }

        // compute and return volume
        double volume()
         {
                return width * height * depth;
        }
}

// BoxWeight now fully implements all constructors.

class BoxWeight extends Box
{
        double weight;

BoxWeight(BoxWeight ob)
 {
        super(ob);
        weight = ob.weight;
}

// constructor when all parameters are specified
```


notes4free.in

```
BoxWeight(double w, double h, double d, double m)
 {
        super(w, h, d); // call superclass constructor
        weight = m;
}

// default constructor
BoxWeight()
 {
        super();
        weight = -1;
}

// constructor used when cube is created
BoxWeight(double len, double m)
 {
        super(len);
        weight = m;
}
}
class DemoSuper
{
        public static void main(String args[])
         {
                BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
                BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
                BoxWeight mybox3 = new BoxWeight(); // default
                BoxWeight mycube = new BoxWeight(3, 2);
                BoxWeight myclone = new BoxWeight(mybox1);
                double vol;

                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                System.out.println("Weight of mybox1 is " + mybox1.weight);
                System.out.println();

                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
                System.out.println("Weight of mybox2 is " + mybox2.weight);
                System.out.println();

                vol = mybox3.volume();
                System.out.println("Volume of mybox3 is " + vol);
                System.out.println("Weight of mybox3 is " + mybox3.weight);
                System.out.println();
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
        vol = myclone.volume();
        System.out.println("Volume of myclone is " + vol);
        System.out.println("Weight of myclone is " + myclone.weight);
        System.out.println();

        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
        System.out.println("Weight of mycube is " + mycube.weight);
        System.out.println();
    }
}
```

output:

```
    Volume of mybox1 is 3000.0
    Weight of mybox1 is 34.3
    Volume of mybox2 is 24.0
    Weight of mybox2 is 0.076
    Volume of mybox3 is -1.0
    Weight of mybox3 is -1.0
    Volume of myclone is 3000.0
    Weight of myclone is 34.3
    Volume of mycube is 27.0
    Weight of mycube is 2.0
```

A Second Use for super

- **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

```
// Using super to overcome name hiding.

class A
{
    int i;
}

// Create a subclass by extending class A.

class B extends A
{
    int i; // this i hides the i in A

    B(int a, int b)
    {
        super.i = a; // i in A
        i = b; // i in B
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

 notes4free.in

```
        }
        void show()
         {
                System.out.println("i in superclass: " + super.i);
                System.out.println("i in subclass: " + i);
        }
}

class UseSuper
{
        public static void main(String args[])
         {
                B subOb = new B(1, 2);
                subOb.show();
        }
}
```

This program displays the following:
i in superclass: 1
i in subclass: 2

Creating a Multilevel Hierarchy

- given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**.
- In it, the subclass **BoxWeight** is used as a superclass to create the subclass called **Shipment**. **Shipment** inherits all of the traits of **BoxWeight** and **Box**, and adds a field called **cost**, which holds the cost of shipping such a parcel.

```
class Box
{
        private double width;
        private double height;
        private double depth;

        // construct clone of an object
        Box(Box ob)
         {
                width = ob.width;
                height = ob.height;
                depth = ob.depth;
        }

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

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

```
        depth = d;
        }

// constructor used when no dimensions specified
        Box()
        {
                width = -1; // use -1 to indicate
                height = -1; // an uninitialized
                depth = -1; // box
        }

        Box(double len)
         {
                width = height = depth = len;
        }

        double volume()
        {
                return width * height * depth;
        }
}

// Add weight.

class BoxWeight extends Box
{
        double weight;

        BoxWeight(BoxWeight ob)
         {
                super(ob);
                weight = ob.weight;
        }

        BoxWeight(double w, double h, double d, double m)
         {
                super(w, h, d);
                weight = m;
        }

        BoxWeight()
        {
                super();
                weight = -1;
        }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
        BoxWeight(double len, double m)
         {
                super(len);
                weight = m;
        }
}

// Add shipping costs.
class Shipment extends BoxWeight
 {
        double cost;

        Shipment(Shipment ob)
         {
                super(ob);
                cost = ob.cost;
        }

        Shipment(double w, double h, double d,double m, double c)
        {
                super(w, h, d, m); // call superclass constructor
                cost = c;
        }

        Shipment()
         {
                super();
                cost = -1;
        }

        Shipment(double len, double m, double c)
         {
                super(len, m);
                cost = c;
        }
}
class DemoShipment
{
        public static void main(String args[])
        {
                Shipment shipment1 = new Shipment(10, 20, 15, 10, 3.41);
                Shipment shipment2 = new Shipment(2, 3, 4, 0.76, 1.28);
                double vol;

                vol = shipment1.volume();
                System.out.println("Volume of shipment1 is " + vol);
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

```
                System.out.println("Weight of shipment1 is " + shipment1.weight);
                System.out.println("Shipping cost: $" + shipment1.cost);
                System.out.println();

                vol = shipment2.volume();
                System.out.println("Volume of shipment2 is " + vol);
                System.out.println("Weight of shipment2 is " + shipment2.weight);
                System.out.println("Shipping cost: $" + shipment2.cost);
                }
}
```

output :

Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41

Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28

When Constructors Are Called

- given a subclass called **B** and a superclass called **A**, is **A**'s constructor called before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass.
- Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used.

```
class A
{
        A() {
                System.out.println("Inside A's constructor.");
        }
}

class B extends A
 {
        B() {
                System.out.println("Inside B's constructor.");
        }
}

class C extends B
 {
        C() {
                System.out.println("Inside C's constructor.");
```


notes4free.in

```
        }
}
class CallingCons
 {
        public static void main(String args[])
         {
                C c = new C();
        }
}
```

output :

```
                Inside A's constructor
                Inside B's constructor
                Inside C's constructor
```

Method Overriding

- when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.

```
class A
{
        int i, j;
        A(int a, int b)
         {
                i = a;
                j = b;
        }

// display i and j
        void show()
         {
                System.out.println("i and j: " + i + " " + j);
        }
}

class B extends A
{
        int k;
        B(int a, int b, int c)
         {
                super(a, b);
                k = c;
        }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

```
        void show()
        {
                System.out.println("k: " + k);
        }
}
class Override
{
        public static void main(String args[])
        {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls show() in B
        }
}
```

output:

    k: 3

- the version of **show( )** inside **B** overrides the version declared in **A**.
- to access the superclass version of an overridden method can be called using **super**.

```
class B extends A
{
        int k;

        B(int a, int b, int c)
        {
                super(a, b);
                k = c;
        }

        void show()
        {
                super.show(); // this calls A's show()
                System.out.println("k: " + k);
        }
}
```

output:

    i and j: 1 2
    k: 3

Here, **super.show( )** calls the superclass version of **show( )**.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

- Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

```
class A
{
        int i, j;

        A(int a, int b)
         {
                i = a;
                j = b;
        }

        // display i and j
        void show()
         {
                System.out.println("i and j: " + i + " " + j);
        }
}

// Create a subclass by extending class A.

class B extends A
{
        int k;

        B(int a, int b, int c)
         {
                super(a, b);
                k = c;
        }

        // overload show()

        void show(String msg)
         {
                System.out.println(msg + k);
        }
}
class Override
{
        public static void main(String args[])
         {
                B subOb = new B(1, 2, 3);
                subOb.show("This is k: "); // this calls show() in B
```

```
        subOb.show(); // this calls show() in A
        }
}
```

The output produced by this program is shown here:
This is k: 3
i and j: 1 2

Packages and Interfaces

- *Packages* are containers for classes that are used to keep the class name space compartmentalized.
- Through the use of the **interface** keyword, Java allows to fully abstract the interface from its implementation.
- Using **interface**, we can specify a set of methods that can be implemented by one or more classes.
- The **interface**, itself, does not actually define any implementation.
- A class can implement more than one interface.

Packages

- Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.
- The package is both a naming and a visibility control mechanism.
- It is possible to define classes inside a package that are not accessible by code outside that package.
- We can define class members that are only exposed to other members of the same package.

Defining a Package

- To create a package ,simply include a **package** command as the first statement in a Java source file.
- Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored.
- If we skip the **package** statement, the class names are put into the default package, which has no name.
- The general form of the **package** statement:
  package *pkg*;
- Here, *pkg* is the name of the package.
- For example, the following statement creates a package called **MyPackage**.
  package MyPackage;
- The general form of a multileveled package statement is shown here:
  package *pkg1*[.*pkg2*[.*pkg3*]];

notes4free.in

Finding Packages and CLASSPATH

- How does the Java run-time system know where to look for packages that we create?
- The answer has three parts.
- First, by default, the Java run-time system uses the current working directory as its starting point.
- Second, we can specify a directory path or paths by setting the **CLASSPATH** environmental variable.
- Third, we can use the **-classpath** option with **java** and **javac** to specify the path to our classes.

A Short Package Example

```
package MyPack;

class Balance
 {
        String name;
        double bal;

        Balance(String n, double b)
         {
                name = n;
                bal = b;
        }
        void show()
         {
                if(bal<0)
                System.out.print("--> ");
                System.out.println(name + ": $" + bal);
        }
}

class AccountBalance
{
        public static void main(String args[])
         {
                Balance current[] = new Balance[3];

                current[0] = new Balance("K. J. Fielding", 123.23);
                current[1] = new Balance("Will Tell", 157.02);
                current[2] = new Balance("Tom Jackson", -12.33);

                for(int i=0; i<3; i++)
                 current[i].show();
        }
}
```


notes4free.in

- Call this file **AccountBalance.java** and put it in a directory called **MyPack**.

Access Protection

- Packages add another dimension to access control.
- Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods.
- Packages act as containers for classes and other subordinate packages.
- Classes act as containers for data and code.
- Java addresses four categories of visibility for class members:
  - • Subclasses in the same package
  - • Non-subclasses in the same package
  - • Subclasses in different packages
  - • Classes that are neither in the same package nor subclasses
- The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.
- Anything declared **public** can be accessed from anywhere.
- Anything declared **private** cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the **default access.**
- If we want to allow an element to be seen outside our current package, but only to classes that subclass our class directly, then declare that element **protected**.

	Private	No Modifier	Protected	Public
Same class	yes	yes	yes	yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

notes4free.in

An Access Example

- This has two packages and five classes.
- Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**.

This is file **Protection.java**:

```
package p1;

public class Protection
{
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;

        public Protection()
         {
                System.out.println("base constructor");
                System.out.println("n = " + n);
                System.out.println("n_pri = " + n_pri);
                System.out.println("n_pro = " + n_pro);
                System.out.println("n_pub = " + n_pub);
        }
}
```

This is file **Derived.java**:

```
package p1;

class Derived extends Protection
{
        Derived()
        {
        System.out.println("derived constructor");
        System.out.println("n = " + n);

        // System.out.println("n_pri = "4 + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

This is file **SamePackage.java**:

```
package p1;

class SamePackage
 {
        SamePackage()
        {
                Protection p = new Protection();
                System.out.println("same package constructor");
                System.out.println("n = " + p.n);

                // System.out.println("n_pri = " + p.n_pri);

                System.out.println("n_pro = " + p.n_pro);
                System.out.println("n_pub = " + p.n_pub);
        }
}
```

- Following is the source code for the other package, **p2**.
- The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection.
- the default only allows access from within the class or the package, not extra-package subclasses.
- the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.

This is file **Protection2.java**:

```
package p2;

class Protection2 extends p1.Protection
{
        Protection2()
        {
        System.out.println("derived other package constructor");

        // System.out.println("n = " + n);

        // System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
```


notes4free.in

```
}
```

This is file **OtherPackage.java**:

```
package p2;

class OtherPackage
 {
        OtherPackage()
        {

                p1.Protection p = new p1.Protection();
                System.out.println("other package constructor");


                // System.out.println("n = " + p.n);

                // System.out.println("n_pri = " + p.n_pri);

                // System.out.println("n_pro = " + p.n_pro);

                System.out.println("n_pub = " + p.n_pub);
        }
}
.
package p1;

// Instantiate the various classes in p1.

public class Demo
 {
        public static void main(String args[])
         {
                Protection ob1 = new Protection();
                Derived ob2 = new Derived();
                SamePackage ob3 = new SamePackage();
        }
}


// Demo package p2.

package p2;

public class Demo
 {
```


notes4free.in

```
        public static void main(String args[])
         {
                Protection2 ob1 = new Protection2();
                OtherPackage ob2 = new OtherPackage();
        }
}
```

Importing Packages

- the **import** statement is used to bring certain classes, or entire packages, into visibility.
- **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.
- This is the general form of the **import** statement:
  import *pkg1*[.*pkg2*].(*classname*|*);
- Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**).

This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;.
```

- All of the standard Java classes included with Java are stored in a package called **java**.
- The basic language functions are stored in a package inside of the **java** package called **java.lang**.

```
                import java.lang.*;

        import java.util.*;
        class MyDate extends Date
        {
        }

        class MyDate extends java.util.Date
        {
        }
```

- if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**,

```
public class Balance
 {
        String name;
        double bal;

        public Balance(String n, double b)
        {
                name = n;
```


notes4free.in

```
            bal = b;
        }
        public void show()
         {
            if(bal<0)
            System.out.print("--> ");
            System.out.println(name + ": $" + bal);
        }
}
```

- the **Balance** class is now **public**. Also, its constructor and its **show( )** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package.
- **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;

class TestBalance
{
        public static void main(String args[])
         {
            class and call its constructor. */
            Balance test = new Balance("J. J. Jaspers", 99.88);
            test.show(); // you may also call show()
        }
}
```

- Using the keyword **interface**, you can fully abstract a class' interface from its implementation.
- Once interface is defined, any number of classes can implement an **interface**.
- Also, one class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface.

Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name
 {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        return-type method-nameN(parameter-list);
        type final-varnameN = value;
}
```


notes4free.in

- When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.
- When it is declared as **public**, the interface can be used by any other code.
- the methods that are declared have no bodies. They end with a semicolon after the parameter list.
- They are abstract methods; there can be no default implementation of any method specified within an interface.
- Each class that includes an interface must implement all of the methods.
- Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized.
- All methods and variables are implicitly **public**.
- Here is an example of a simple interface that contains one method called **callback( )** that takes a single integer parameter.

```
interface Callback
{
        void callback(int param);
}
```

Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the **implements** clause:

class *classname* [extends *superclass*] [implements *interface* [,*interface...*]]
{
        // class-body
}

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- The methods that implement an interface must be declared **public**.
- the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

- Here is a small example class that implements the **Callback** interface shown earlier.

```
class Client implements Callback
 {
// Implement Callback's interface
```


notes4free.in

```
        public void callback(int p)
         {
                System.out.println("callback called with " + p);
        }
}
```

- Notice that **callback( )** is declared using the **public** access specifier.
- It is both permissible and common for classes that implement interfaces to define additional members of their own.
- For example, the following version of **Client** implements **callback( )** and adds the method **nonIfaceMeth( )**:

```
class Client implements Callback
{
// Implement Callback's interface

        public void callback(int p)
        {
                System.out.println("callback called with " + p);
        }
        void nonIfaceMeth()
         {
                System.out.println("Classes that implement interfaces " + "may also define other
                members, too.");
        }
}
```

Accessing Implementations Through Interface References

- we can declare variables as object references that use an interface rather than a class type.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When we call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to

The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface
{
        public static void main(String args[])
         {
                Callback c = new Client();
                c.callback(42);
        }
}
```


notes4free.in

output :
callback called with 42

- variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**.
- Although **c** can be used to access the **callback( )** method, it cannot access any other members of the **Client** class.
- **c** could not be used to access **nonIfaceMeth( )** since it is defined by **Client** but not **Callback**.

the second implementation of **Callback**, shown here to show the polymorphic behavior:

```
// Another implementation of Callback.

class AnotherClient implements Callback
 {
        public void callback(int p)
        {
                System.out.println("Another version of callback");
                System.out.println("p squared is " + (p*p));
        }
}


class TestIface2
 {
        public static void main(String args[])
        {
                Callback c = new Client();
                AnotherClient ob = new AnotherClient();
                c.callback(42);
                c = ob; // c now refers to AnotherClient object
                c.callback(42);
        }
}
```

output:

callback called with 42
Another version of callback
p squared is 1764

the version of **callback( )** that is called is determined by the type of object that **c** refers to at run time.

 notes4free.in

Partial Implementations

- If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**.
- For example:

```
abstract class Incomplete implements Callback
 {
int a, b;
void show()
{
        System.out.println(a + " " + b);
}
// ...
}
```

- the class **Incomplete** does not implement **callback( )** and must be declared as abstract.
- Any class that inherits **Incomplete** must implement **callback( )** or be declared **abstract** itself.

Nested Interfaces

- An interface can be declared a member of a class or another interface. Such an interface is called a *member interface* or a *nested interface*.
- A nested interface can be declared as **public**, **private**, or **protected**. This differs from a top-level interface, which must either be declared as **public** or use the default access level,

```
class A
 {
        // this is a nested interface
        public interface NestedIF
         {
                boolean isNotNegative(int x);
        }
}

class B implements A.NestedIF
{
        public boolean isNotNegative(int x)
        {
                return x < 0 ? false : true;
        }
}

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


notes4free.in

```
        {
            A.NestedIF nif = new B();

            if(nif.isNotNegative(10))
            System.out.println("10 is not negative");
            if(nif.isNotNegative(-12))
            System.out.println("this won't be displayed");
            }
}
```

- **A** defines a member interface called **NestedIF** and that it is declared **public**.
- **B** implements the nested interface by specifying implements A.NestedIF

Applying Interfaces

- a class called **Stack** that implemented a simple fixed-size stack.
- the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation.
- First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**.

This interface will be used by both stack implementations.

```
interface IntStack
 {
void push(int item);
int pop();
}
```

- The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.

class FixedStack implements IntStack
 {
        private int stck[];
        private int tos;

        FixedStack(int size)
         {
                stck = new int[size];
                tos = -1;
        }

        public void push(int item)
         {
```

(SOURCE DIGINOTES)

```
        if(tos==stck.length-1) // use length member
        System.out.println("Stack is full.");
        else
        stck[++tos] = item;
    }

    public int pop()
    {
        if(tos < 0)
        {
        System.out.println("Stack underflow.");
        return 0;
        }
        else
        return stck[tos--];
    }
}

class IFTest
{
    public static void main(String args[])
     {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    for(int i=0; i<5; i++)
     mystack1.push(i);

    for(int i=0; i<8; i++)
    mystack2.push(i);

    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
    System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
    System.out.println(mystack2.pop());
    }
}
```

- another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition.

```
class DynStack implements IntStack
{
```


notes4free.in

```
        private int stck[];
        private int tos;

        DynStack(int size)
         {
                stck = new int[size];
                tos = -1;
        }
// Push an item onto the stack
        public void push(int item)
        {

                if(tos==stck.length-1)
                {
                        int temp[] = new int[stck.length * 2];
                                // double size
                        for(int i=0; i<stck.length; i++)
                         temp[i] = stck[i];
                        stck = temp;
                        stck[++tos] = item;
                }
                else
                        stck[++tos] = item;
        }

        public int pop()
         {
                if(tos < 0)
                 {
                        System.out.println("Stack underflow.");
                        return 0;
                }
                else
                        return stck[tos--];
                }
}
class IFTest2
{
        public static void main(String args[])
         {
        DynStack mystack1 = new DynStack(5);
        DynStack mystack2 = new DynStack(8);

        for(int i=0; i<12; i++) mystack1.push(i);
        for(int i=0; i<20; i++) mystack2.push(i);
```


CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

```
        System.out.println("Stack in mystack1:");
        for(int i=0; i<12; i++)
        System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<20; i++)
        System.out.println(mystack2.pop());
        }
}
```

- The following class uses both the **FixedStack** and **DynStack** implementations. It does so through an interface reference. This means that calls to **push( )** and **pop( )** are resolved at run time rather than at compile time.

```
class IFTest3
 {
        public static void main(String args[])
         {
        IntStack mystack; // create an interface reference variable
        DynStack ds = new DynStack(5);
        FixedStack fs = new FixedStack(8);

        mystack = ds; // load dynamic stack
        // push some numbers onto the stack
        for(int i=0; i<12; i++) mystack.push(i);
        mystack = fs; // load fixed stack

        for(int i=0; i<8; i++) mystack.push(i);
        mystack = ds;
        System.out.println("Values in dynamic stack:");

        for(int i=0; i<12; i++)
        System.out.println(mystack.pop());

        mystack = fs;
        System.out.println("Values in fixed stack:");

        for(int i=0; i<8; i++)
        System.out.println(mystack.pop());
        }
}
```

- **mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push( )** and **pop( )** defined by the **DynStack** implementation.
- When it refers to **fs**, it uses the versions of **push( )** and **pop( )** defined by **FixedStack**.
- Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

Variables in Interfaces

- we can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.

```
import java.util.Random;

interface SharedConstants
{
        int NO = 0;
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int SOON = 4;
        int NEVER = 5;
}
class Question implements SharedConstants
 {
        Random rand = new Random();
        int ask()
         {
        int prob = (int) (100 * rand.nextDouble());

        if (prob < 30)
        return NO;
        else if (prob < 60)
        return YES;
        else if (prob < 75)
        return LATER;
        else if (prob < 98)
        return SOON;
        else
        return NEVER;
        }
}
class AskMe implements SharedConstants
 {
        static void answer(int result)
        {
                switch(result)
                 {
                case NO:
                                System.out.println("No");
                                break;
                case YES:
                                System.out.println("Yes");
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

 notes4free.in

```
                        break;
                case MAYBE:
                        System.out.println("Maybe");
                        break;
                case LATER:
                        System.out.println("Later");
                        break;
                case SOON:
                        System.out.println("Soon");
                        break;
                case NEVER:
                        System.out.println("Never");
                        break;
        }
}

        public static void main(String args[])
        {
        Question q = new Question();
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        }
}
```

Note that the results are different each time it is run.

Later
Soon
No
Yes

Interfaces Can Be Extended

- One interface can inherit another by use of the keyword **extends**.
- The syntax is the same as for inheriting classes

```
interface A
 {
        void meth1();
        void meth2();
}

interface B extends A
{
        void meth3();
```


notes4free.in

```
}

class MyClass implements B
{
        public void meth1()
        {
                System.out.println("Implement meth1().");
        }

        public void meth2()
        {
                System.out.println("Implement meth2().");
        }
        public void meth3()
        {
                System.out.println("Implement meth3().");
        }
}
class IFExtend
{
        public static void main(String arg[])
        {
                MyClass ob = new MyClass();
                ob.meth1();
                ob.meth2();
                ob.meth3();
        }
}
```

- any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

 notes4free.in

Exception Handling

- an exception is a run-time error.
- languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on.
- Java's exception handling avoids handling problems manually and, in the process, brings run-time error management into the object toriented world.

Exception-Handling Fundamentals

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- That method may choose to handle the exception itself, or pass it on.
- Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system,
- or they can be manually generated by your code.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Briefly, here is how they work. Program statements that create exceptions are contained within a **try** block.
- If an exception occurs within the **try** block, it is thrown.we can catch this exception (using **catch**) and handle it .
- System-generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
```


notes4free.in

}

- Here, *ExceptionType* is the type of exception that has occurred.

Exception Types

- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch.
- There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.
- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error

Uncaught Exceptions

. This program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0
 {
        public static void main(String args[])
         {
                int d = 0;
                int a = 42 / d;
        }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception.
- This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.
- Here we don't have any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.
- Any exception that is not caught by our program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.
- Here is the exception generated when this example is executed:
  java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)

Using try and catch

- Although the default exception handler provided by the Java run-time system is useful for

notes4free.in

debugging,we should handle an exception ourself.

- Doing so provides two benefits.
- First, it allows you to fix the error.
- Second, it prevents the program from automatically terminating.
- To handle a run-time error, simply enclose the code inside a **try** block.
- Immediately following the **try** block, include a **catch** clause that specifies the exception type to catch

```
class Exc2
{
	public static void main(String args[])
	{
		int d, a;

		try
		 {
			d = 0;
			a = 42 / d;
			System.out.println("This will not be printed.");
		}
		 catch (ArithmeticException e)
		{
			System.out.println("Division by zero.");
		}
		System.out.println("After catch statement.");
	}
}
```

This program generates the following output:
Division by zero.
After catch statement.

- A **try** and its **catch** statement form a unit.
- The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- A **catch** statement cannot catch an exception thrown by another **try** statement.

```
class HandleError
{
	public static void main(String args[])
	{
		int a=0, b=0, c=0;
		Random r = new Random();

			for(int i=0; i<32000; i++)
			{
				try
```


notes4free.in

```
                        {
                                b = r.nextInt();
                                c = r.nextInt();
                                a = 12345 / (b/c);
                        }
                        catch (ArithmeticException e)
                        {
                                System.out.println("Division by zero.");
                                a = 0; // set a to zero and continue
                        }
                System.out.println("a: " + a);
                }
        }
}
```

Multiple catch Clauses

- more than one exception could be raised by a single piece of code.
- To handle this type of situation, we can specify two or more **catch** clauses, each catching a different type of exception.
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.

The following example traps two different exception types:

```
// Demonstrate multiple catch statements.

class MultiCatch
 {
        public static void main(String args[])
        {
                try
                 {
                        int a = args.length;
                        System.out.println("a = " + a);
                        int b = 42 / a;
                        int c[] = { 1 };
                        c[42] = 99;
                }
                catch(ArithmeticException e)
                 {
                        System.out.println("Divide by 0: " + e);
                }
                catch(ArrayIndexOutOfBoundsException e)
                 {
                        System.out.println("Array index oob: " + e);
```


notes4free.in

```
        }
        System.out.println("After try/catch blocks.");
        }
}
```

output:

```
        C:\>java MultiCatch
        a = 0
        Divide by 0: java.lang.ArithmeticException: / by zero
        After try/catch blocks.
        C:\>java MultiCatch TestArg
        a = 1
        Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
        After try/catch blocks.
```

```
class SuperSubCatch
{
    public static void main(String args[])
     {
        try
        {
            int a = 0;
            int b = 42 / a;
        }
         catch(Exception e)
         {
            System.out.println("Generic Exception catch.");
        }

        catch(ArithmeticException e)
        {
            System.out.println("This is never reached.");
        }
    }
}
```

- If this program is compiled , we will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught.
- Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**.
- This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

Nested try Statements

- The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**.

notes4free.in

```
class NestTry
 {
        public static void main(String args[])
         {
                try
                 {
                        int a = args.length;
                        int b = 42 / a;
                        System.out.println("a = " + a);
                                try
                                 {
                                        if(a==1) a = a/(a-a);
                                        if(a==2)
                                        {
                                        int c[] = { 1 };
                                        c[42] = 99; // generate an out-of-bounds exception
                                        }
                                }
                                 catch(ArrayIndexOutOfBoundsException e)
                                 {
                                        System.out.println("Array index out-of-bounds: " + e);
                                }
                }
                 catch(ArithmeticException e)
                {
                        System.out.println("Divide by 0: " + e);
                }
        }
}
```

- When we execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block.
- Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested **try** block.
- Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled.
- If we execute the program with two command-line arguments, an array boundary exception is generated from within the inner **try** block.

```
C:\>java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One Two
a = 2
Array index out-of-bounds:
```

 notes4free.in

java.lang.ArrayIndexOutOfBoundsException:42

throw

- it is possible for your program to throw an exception explicitly, using the **throw** statement.
- The general form of **throw** is shown here:
  throw *ThrowableInstance*;
- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.
- Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.

```
class ThrowDemo
{
	static void demoproc()
	 {
		try
		{
			throw new NullPointerException("demo");
		}
		catch(NullPointerException e)
		{
			System.out.println("Caught inside demoproc.");
			throw e; // rethrow the exception
		}
	}
	public static void main(String args[])
	{
		try
		{
			demoproc();
		}
		 catch(NullPointerException e)
		 {
			System.out.println("Recaught: " + e);
		}
	}
}
```

- First, **main( )** sets up an exception context and then calls **demoproc( )**.
- The **demoproc( )** method then sets up another exceptionhandling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line.
- The exception is then rethrown.
- Here is the resulting output:

Caught inside demoproc.

 notes4free.in

Recaught: java.lang.NullPointerException: demo

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception.
- We can do this by including a **throws** clause in the method's declaration.
- A **throws** clause lists the types of exceptions that a method might throw

.

- This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
// body of method
}
```

```
class ThrowsDemo
{
	static void throwOne() throws IllegalAccessException
	{
		System.out.println("Inside throwOne.");
		throw new IllegalAccessException("demo");
	}
	public static void main(String args[])
	 {
		try
		{
			throwOne();
		}
		 catch (IllegalAccessException e)
		 {
			System.out.println("Caught " + e);
		}
	}
}
```

Here is the output generated by running this example program:

inside throwOne

caught java.lang.IllegalAccessException: demo

finally

- **finally** creates a block of code that will be executed after a **try**/**catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception

```
class FinallyDemo
{
	static void procA()
```

```
 {
          try {
          System.out.println("inside procA");
          throw new RuntimeException("demo");
          }
           Finally
           {
                    System.out.println("procA's finally");
          }
}

static void procB()
 {
          try {
                    System.out.println("inside procB");
                    return;
          }
           finally {
                    System.out.println("procB's finally");
          }
}

static void procC()
{
          try
          {
                    System.out.println("inside procC");
          }
          Finally
           {
                    System.out.println("procC's finally");
          }
}
public static void main(String args[])
 {
          try
          {
          procA();
          }
          catch (Exception e)
          {
          System.out.println("Exception caught");
          }
procB();
procC();
}
```


notes4free.in

}

- Here is the output generated by the preceding program:

inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally

Java's Built-in Exceptions

- Inside the standard package **java.lang**, Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type **RuntimeException**
- if the method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions.*

Java's Unchecked RuntimeException Subclasses Defined in java.lang

Exception Meaning

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
NullPointerException	Invalid use of a null reference.

- Java's Checked Exceptions Defined in java.lang

ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.

notes4free.in

InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Creating Your Own Exception Subclasses

- It is possible to create to create our own exception types to handle situations specific to your applications.
- just define a subclass of **Exception**
- Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.
- The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**.
- Thus, all exceptions, including those that we create, have the methods defined by **Throwable** available to them.

Method	Description
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of

Chained Exceptions

- The chained exception feature allows you to associate another exception with an exception.
- This second exception describes the cause of the first exception.
- For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero.
- However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly.
- To allow chained exceptions, two constructors and two methods were added to **Throwable**.

  The constructors are shown here:
  Throwable(Throwable *causeExc*)
  Throwable(String *msg*, Throwable *causeExc*)
- These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.
- The chained exception methods added to **Throwable** are **getCause( )** and **initCause( )**.
- These methods are shown

notes4free.in

```
Throwable getCause( )
Throwable initCause(Throwable causeExc)
```

- The **getCause( )** method returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned.
- The **initCause( )** method associates *causeExc* with the invoking exception and returns a reference to the exception.

```
class ChainExcDemo
 {
        static void demoproc()
        {
                // create an exception
                NullPointerException e =
                new NullPointerException("top layer");
                // add a cause
                e.initCause(new ArithmeticException("cause"));
                throw e;
        }
        public static void main(String args[])
        {
        try
        {
                demoproc();
        }
        catch(NullPointerException e)
         {
                // display top level exception
                System.out.println("Caught: " + e);
                // display cause exception
                System.out.println("Original cause: " +
                e.getCause());
        }
        }
}
```

The output from the program is shown here:
Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause

- In this example, the top-level exception is **NullPointerException**.
- To it is added a cause exception, **ArithmeticException**. When the exception is thrown out of **demoproc( )**, it is caught by **main( )**.
- There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling **getCause( )**.

 notes4free.in

Multithreaded programming

UNIT 4

*) Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently.

*) Each part of such a program is called a thread, and each thread defines a separate path of execution.

*) There are two distinct types of multitasking: process based and thread-based.

*) A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently.

*) For example, process-based multitasking enables us to run the Java compiler at the same time that we are using a text editor.

*) In process based multitasking; a program is the smallest unit of code that can be dispatched by the scheduler.

notes4free.in

*) In, a thread based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.

*) For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Note: process - based

*) Each process have its own address in memory i.e each process allocates separate memory area.

*) process is heavy weight.

*) cost of communication between the process is high (switching from one process to another requires some time for saving & loading registers, memory maps, updating lists etc)

*) Thread - Based

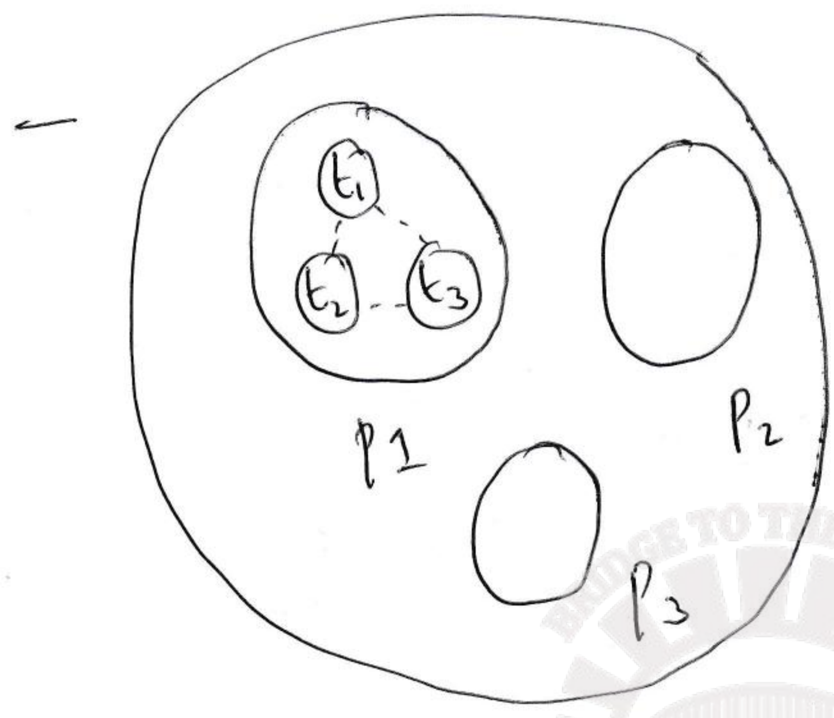
*) Threads share the same address space.

*) Thread is lightweight

(threads within the process share the same address space, code section, data section, os resources & communication

(SOURCE DIGINOTES)
notes4free.in

between threads is low so) It is light weight



*) cost of communication b/w the thread is low, so we always prefer more multithreading than multi processing.

→ The Java Thread Model:

*) The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.

*) In fact, Java uses threads to enable the entire environment to be asynchronous.

notes4free.in

*) single threaded systems use an approach called an event loop with polling.

*) polling is a single event queue to decide what to do next. once this polling mechanism returns with, say a file is ready to be read, then the event loop dispatches control to the event handler.

polling (single event queue) → event loop —dispatches control→ event handler.

*) until the event handler returns, nothing else can happen in the system. This wastes CPU time.

*) when a thread blocks because it is waiting for some resources, the entire program stops running.

*) The benefit of Java's multithreading is that the polling mechanism is eliminated.

*) one thread can pause without stopping other parts of our programs. for example, the idle time created when a thread reads data from a N/w or waits for user i/p can be utilized somewhere else.

notes4free.in

Threads exist in several states:

*) A thread can be running.

*) A running thread can be suspended.

*) A suspended thread can then be resumed.

*) A thread can be blocked

*) A thread can be terminated

*) A thread can be resumed

→ Thread Priorities

*) Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.

*) The thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch.

*) The rules that determine when a context switch takes place are simple:

– A thread can voluntarily claim control. This is done by explicitly yielding, sleeping or blocking on pending I/O. In this scenario, all other threads are examined, and the

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

notes4free.in

highest-priority thread that is ready to run is given the cpu.

*) A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted.

*) In cases where two threads with the same priority are competing for cpu cycles, the situation is a bit complicated. For operating systems such as windows, threads of equal priority are time-sliced automatically in round-robin fashion.

→ Synchronization

*) Because multithreading introduces an asynchronous behaviour to our programs, there must be a way to enforce synchronicity when you need it.

*) For example, if you want two threads to communicate and share a complicated data structure, such as linked list, you need some way to ensure that they don't conflict with eachother. i.e we must prevent one thread from writing data while another is in the middle of reading.

notes4free.in

*) For this purpose, java implements an elegant twist on an age-old model of interprocess synchronization: the monitor

*) A monitor is a very small box that can hold only one thread. once a thread enters a monitor, all other threads must wait until that thread exits the monitor.

*) In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

→ Messaging

*) Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have.

*) Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

→ The Thread class and the Runnable Interface.

*) Java multithreading system is built upon the Thread class, its methods, Runnable.

*) The Thread class defines several methods that help manage threads.

Method	Meaning.
getName	obtains a thread's name.
getPriority	obtains a thread's priority
isAlive	Determine if a thread is still running.
join	wait for a thread to terminate.
run	Entry point for the thread
sleep	suspend a thread for a period of time.
start	start a thread by calling its run method.

CAMBRIDGE INSTITUTE OF TECHNOLOGY

(SOURCE DIGINOTES)

notes4free.in

Thread Priorities:

*) Thread priorities are used by the thread schedular to decide when each thread should be allowed to run. In theory, higher priority threads get more cpu time then lower-priority threads.

*) In theory, threads of equal priority should get equal access to the cpu. But you need to be careful. Remember, Java is designed to work in a wide range of environment. Some of those environments implement multitasking differently than others.

*) To set a thread's priority, use the setpriority() method, which is a member of thread

general form: final void setPriority(int level)

*) The value of level must be within the range MIN_PRIORITY & MAX_PRIORITY. Currently, these values are 1 and 10 respectively.

To return a thread to default priority, specify NORM_PRIORITY, which is currently 5.

notes4free.in

These priorities are defined as final variables within Thread.

```
// Demonstrate thread priorities.

Class clicker implements Runnable
{
    int click=0;
    Thread t;
    Private volatile boolean running = true;
    public clicker (int p)
    {
        t =new Thread (this);
        t.setpriority (p);
    }

public void run()
{
    while (running)
    {
    Click++;
    }
}

Public void stop()
{
    running = false;
}
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

```
public void start()
 {
   t.start();
 }
}

class HiLoPri
{
  public static void main(string args[])
 {
  Thread.currentThread().setpriority(Thread.MAX_PRIORITY);
  clicker hi = new clicker(Thread.NORM_PRIORITY +2);
  clicker lo = new clicker(Thread.NORM_PRIORITY -2);
  lo.start();
  hi.start();
 try
 {
  Thread.sleep(10000);
 }
catch(InterruptedException e)
{
  System.out.println("Main thred interrupted");
 }
lo.stop();
hi.stop();
// wait for child threads to terminate
try{
    hi.t.join();
 }  lo.t.join();

catch(InterruptException e)
{
S.o.p("Interrupted Exception");
}
S.o.p("low-priority:"
            + lo.click);
S.o.p("high-priority:"
            + hi.click);
}
}
```

O/p low.priority: 4408112
    high.priority: 589626904

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

Thread priorities

```
class TestMultiPriority1 extends Thread
{
    public void run()
    {
        s.o.pln ("running thread" + Thread.currentThread().getName());

        s.o.pln ("its priority is" + Thread.currentThread().getPriority());
    }

    public static void main (String args[])
    {
        TestMultiPriority1 m1 = new TestMultiPriority1();
        TestMultiPriority1 m2 = new TestMultiPriority1();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

O/p running thread thread0
Its priority is 10
running thread thread1
Its priority is 10

CAMBRIDGE INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

→ synchronization

*) when two or more threads need access to a shared resources, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

*) Key to synchronization is the concept of the monitor. A monitor is an object that is used as a mutually exclusive lock or mutex. only one thread can own a monitor at a given time.

*) when a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

```
// This program is not synchronized.
class callme
{
    void call(string msg)
    {
     system.out.print("I" + msg);
     try {
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

```
Thread.sleep(1000);
}
catch(InterruptedException e)
{
System.out.println("Interrupted");
}
System.out.println("]");
}
}

class caller implements Runnable
{
String msg;
Callme target;
Thread t;
public caller(callme targ, string s)
{
target=targ;
msg=s;
t=new Thread(this);
t.start();
}
public void run()
{
target.call(msg);
}
}
```

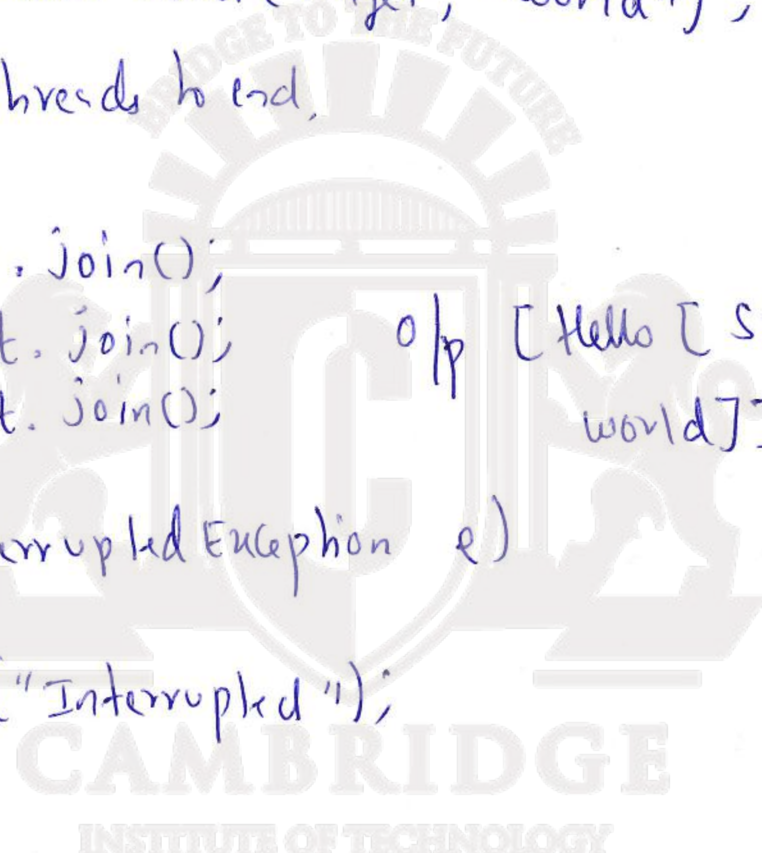
```
public void run()
// to synchronize
{
synchronized(target)
{
target.call(msg);
}
}
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

```
class synch
{
 public static void main(String args[])
 {
   callme target = new callme();
   caller ob1 = new caller(target, "Hello");
   caller ob2 = new caller(target, "synchronized");
   caller ob3 = new caller(target, "world");

   // wait for threads to end
    try{
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
    }
   catch (InterruptedException e)
    {
      s.o.p("Interrupted");
    }
  }
}
```

o/p [Hello [Synchronized [world]]]



CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

*) To fix the preceding program, we must serialize access to call(). That is, we must restrict its access to only one thread at a time.

*) To do this, we simply need to precede call() definition with the keyword synchronized as shown here:

notes4free.in

```
class callme
{
  synchronized void call(string msg)
  {
     . . .
  }
```

o/p [Hello]
[Synchronized]
[world]

→ **Interthread Communication**

*) polling is usually implemented by a loop that is used to check some condition repeatedly.

*) once the condition is true, appropriate action is taken. This wastes cpu time. for example, consider the classic queuing problem,

*) where one thread is producing some data and another is consuming it, To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.

*) In polling system, the consumer would waste many cpu cycles while it waited for the producer to produce.

notes4free.in

```
class callme
{
  synchronized void call(string msg)
  {
     ...
  }
```

o/p [Hello]
[Synchronized]
[world]

→ Interthread Communication

*) polling is usually implemented by a loop that is used to check some condition repeatedly.

*) once the condition is true, appropriate action is taken. This wastes cpu time. for example, consider the classic queuing problem,

*) where one thread is producing some data and another is consuming it, To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.

*) In polling system, the consumer would waste many cpu cycles while it waited for the producer to produce.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES) BRIDGE TO THE FUTURE
notes4free.in

once the producer was finished, it would start polling, wasting more cpu cycles waiting for the consumer to finish, so on.

*) To avoid polling, Java includes an interprocess communication mechanism via the wait(), notify() and notifyAll() methods.

1) wait() : tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().

calling thread
↓ to give up
monitor
↓ go to sleep
until some other thread enters monitor (notify())

2) notify() : wakes up the first thread that called wait() on same object.

3) notifyAll() : wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

// incorrect implementation of a producer & consumer.

```
class Q
{
 int n;
 synchronized int get()
 {
   System.out.println("got:" +n);
 } return n;
```

notes4free.in

```
synchronized void put (int n)
 {
    this.n=n;
    system.out.println("put:"+n);
  }
}
```

```
class producer implements Runnable
{
  Q q;
  producer(Q q)
  {
   this.q=q;
   new Thread (this,"producer").start();
  }
  public void run()
  {
    int i=0;
    while (true)
    {
     q.put(i++);
    }
  }
}
```

```
class Consumer implements Runnable
{
   Q q;
   consumer(Q q)
   {
    this.q=q;
    new Thread (this,"consumer").start();
   }
```

BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

```
public void run()
{
 while(true)
 {
  q.get();
 }
}
}
```

```
class PC
{
 public static void main(string args[])
 {
   Q q = new Q();
   new Producer(q);
   new consumer(q);
   system.out.println("press control-c to stop");
 }
}
```

*) Although the put() & get() methods on Q are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue (values will vary with processor speed & load)

Put: 1
Got: 1
Got: 1
Got: 1
Got: 1

Put: 2
Put: 3
put: 4
Put: 5
Put: 6
Put: 7
got: 7

*)

CAMBRIDGE INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

A) The proper way to write the program in java is to use wait() and notify() to signal in both directions, as shown.

// A correct implementation of a producer & consumer.

```
Class Q
{
  int n;
  boolean valueset = false;
  synchronized int get()
  {
    if(!valueset)
    {
      try{
          wait();
         }
      catch (InterruptedException e)
      {
        System.out.println("Interrupt caught");
      }
    System.out.println("Got:" +n);
    valueset = false;
    notify(); // wakes up the first thread that calls wait.
    return n;
  }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

```
synchronized void put(int n)
{
    if(valueset)
    {
        try{
            wait();
        }
        catch(InterruptedException e)
        {
            S.o.p("Interrupt caught");
        }
    this.n = n;
    valueset = true;
    S.o.p("put:" + n);
    notify();
}
}
```

```
Class producer implements Runnable
{
    Q q;
    producer(Q q)
    {  this.q = q;
       new thread(this, "producer").start();
    }
    public void run()
    {
        int i=0;
        while(true)
        {  q.put(i++);
        }
    }
}
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY
BRIDGE TO THE FUTURE
(SOURCE DIGINOTES)
notes4free.in

```
class consumer implements Runnable
{
  Q q;
  Consumer (Q q)
  {
    this.q = q;
    new Thread (this, "Consumer").start();
  }
  public void run()
  {
   while (true)
   {
     q.get();
   }
  }
}

class PCFixed()
{
  public static void main (String args[])
  {
    Q q = new Q();
    new producer (q);
    new Consumer (q);
    S.o.p ("press Control C to stop");
  }
}
```

O/p
```
put : 1
Got : 1
put : 2
Got : 2
put : 3
Got : 3
```

(SOURCE DIGINOTES)

notes4free.in

Event handling

*) Any program that uses a graphical user interface, such as java application written for windows, is event driven.

*) Events are supported by a number of packages, including java.util, java.awt and java.awt.event.

*) most events to which your program will respond are generated when the user interacts with a GUI-based program.

*) There are several types of events, including those generated by the mouse, the keyboard, and various GUI controls, such as a push button, scroll bar or check box.

*) This chapter begins with an overview of Java's event handling. It then examines the main event classes and interfaces used by AWT and demonstrates several fundamentals of event handling.
(ABSTRACT WINDOW TOOLKIT) → provides the interface to create GUI such as buttons.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

→ TWO Event Handling Mechanisms.

*) The way in which events are handled changed significantly between the original version of Java (1.0) and modern version of Java, beginning with version 1.1

notes4free.in

*) The 1.0 method of event handling is still supported, but it is not recommended for new programs. The modern approach is the way that events should be handled by all new programs and thus is the method employed by programs.

→ The Delegation Event Model

Note:

1) Register the component with the Listner

Syntax: addTypeListner(TypeListner a);

2) If user performs any event like clicking the mouse button, scrolling etc, the object for ~~concerned~~ Concerned event class is created automatically and information about the source & the event get populated.

3) Event object is forwarded to the method of Registered Listener class.

4) method is now gets executed and returned.

*)

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

→ The Main Thread

```
Class CurrentThreadDemo
{
  public static void main(string args[])
  {
      Thread t = Thread.currentThread();
      system.out.println("current thread :" +t);
    // change the name of the thread
      t.setName("My Thread");
    System.out.println(" After name change " +t);
    try
    {
      for(int i=5; i>0; i--)
      {
          system.out.println(i);
          Thread.sleep(1000);
      }
    }
    Catch(InterruptedException e)
    {
      System.out.println("MAIN THREAD INTERRUPTED");
    }
  }
}
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

*) when a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program. This is the one that is executed when your program begins.

A) The main thread ~~from~~ is important for two reasons:

1) It is the thread from which other "child" threads will be spawned.

2) often, it must be the last thread to finish execution because it performs various shutdown actions.

*) main thread can be automatically created when your program is started, it can be controlled through a Thread object. To do so, we must obtain a reference to it by calling the method currentThread().

general form is: static Thread currentThread()

*) o/p of the program:

Current thread: Thread [main, 5, main]

After name change: Thread [my Thread, 5, main]

5
4
3
2
1

(SOURCE DIGINOTES)

notes4free.in

Notice the output produced when t is used as an argument to println(). This displays, in order, the name of the thread, its priority, and the name of its group.

*) By default, the name of the main thread is main. Its priority is 5, which is the default value, and main is also the name of the group of threads to which this thread belong.

*) The sleep() method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

general form: static void sleep(long milliseconds) throws InterruptedException.

→ Creating a Thread.

*) Java defines two ways in which this can be accomplished.

1) we can implement the Runnable interface.

2) we can extend the Thread class, itself.

(SOURCE DIGINOTES)
CAMBRIDGE INSTITUTE OF TECHNOLOGY
notes4free.in

Implementing Runnable:

*) The easiest way to create a thread is to create a class that implements the Runnable interface.

*) Runnable abstracts a unit of executable code. We can construct a thread on any objects that implements Runnable

*) To implement Runnable, a class need only implement a single method called run()

public void run()

*) Inside run(), we will define the code that constitutes the new thread.

*) After you create a class that implements Runnable, we will instantiate an object of type Thread from within that class.

*) Thread defines several constructors, The one that we will use is shown here:

Thread(Runnable threadob, string threadName)

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

*) In this constructor, threadob is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

*) After the new thread is created, it will not start running until you call its ~~star~~ start() method.

Void start()

*) Example that creates a new thread and starts its running?

```
Class NewThread implements Runnable
{
  Thread t;
  NewThread()
  {
     // Create a new, Second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("child thread" + t);
    t.start(); // start the thread
  }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

```
//This is the entry point for the Second thread.
 public void run()
  {
   try
    {
     for(int i=5; i>0; i--)
      {
       system.out.println("child thread:" +i);
       Thread.sleep(500);
      }
    }
   Catch (InterruptedException e)
   {
    System.out.println("child Interrupted");
   }
   System.out.println("Exiting child thread");
  }
}

Class ThreadDemo
{
  public static void main(String args[])
  {
  new NewThread(); // create a new thread
  (or)
  NewThread P = new NewThread();
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

```
try
{
  for(int i=5; i>0; i--)
  {
   System.out.println("main thread: " + i);
   Thread.sleep(1000);
  }
}
 catch(InterruptedException e)
{
 System.out.println("main thread interrupted");
}
System.out.println("main thread exiting");
}
}
```

Note: Inside NewThread's constructor, a new Thread object is created by t = new Thread(this, "Demo Thread");

O/p : Child thread: Thread[Demo thread, 5, main]

main thread: 5
Child thread: 5
child thread: 4
Main thread: 4
Child thread: 3
" : 2
Main thread: 3
Child thread: 1
Exiting child thread
Main thread: 2
: 1
Main thread exiting

BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

→ Extending Thread

*) The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.

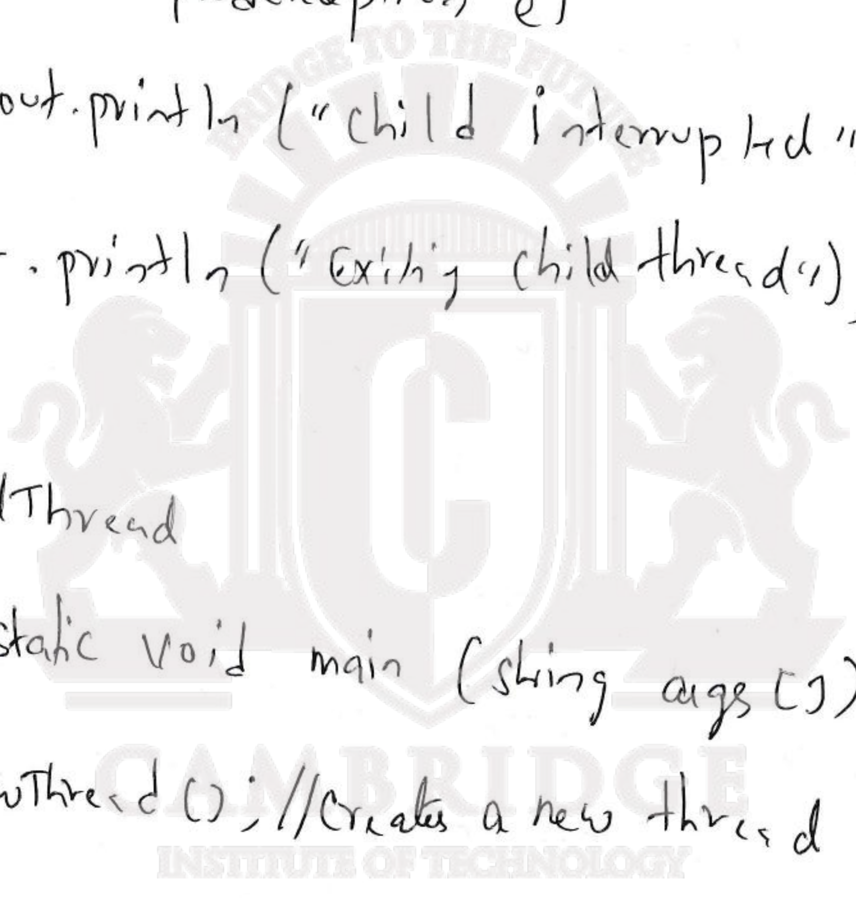
*) The extending class must override the run() method which is the entry point for the new thread. It must also call start() to begin execution of the new thread

```
// Create a second thread by extending Thread.
class NewThread extends Thread
{
 NewThread()
                    // Create a new, second thread.
 {
   super("Demo Thread");
   System.out.println("child thread " + this);
   start();
 }
// This is the entry point for the second thread.
 public void run()
 {
  try
  {
    for (int i=5; i>0; i--)
    {
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

```
try
{
  for(int i=5; i>0; i--)
  {
    system.out.println("Child Thread"+i);
    Thread.sleep(500);
  }
} catch(InterruptedException e)
{
  System.out.println("child interrupted");
}
system.out.println("Exiting child thread");
}
}

class ExtendThread
{
  public static void main (String args[])
  {
    new NewThread(); //Creates a new thread
try {
    for(int i=5; i>0; i--)
    {
      system.out.println("mainThread : "+i);
      Thread.sleep(1000);
    }
  }
  catch(InterruptedException e)
  {
    system.out.println("main thread Interrupted");
  }
  system.out.println("Main thread exiting");
}
}
```



BRIDGE TO THE FUTURE
CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

Choosing an Approach

*) which approach is better. The Thread Class defines several methods that can be overridden by a derived Class, of these method the only one that must be overridden in run().

*) The same method required when you implement Runnable. many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. so if you will not be overriding any of Threads other methods.

→ Creating multiple Threads

*) So far, we have been using only two threads: the main thread and one child thread. However your program can spawn as many threads as it needs.

*) The following program creates three child threads:

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

notes4free.in

```
// create multiple threads

class NewThread implements Runnable
{
    String name; // name of thread
    Thread t;

  NewThread(String threadname)
  {
    name = threadname;
    t = new Thread(this, name);
    system.out.println("New thread" +t);
    t.start(); // start the thread.
  }
// This is the entry point for thread.
public void run()
{
    try{
        for(int i=5; i>0; i--)
        {
          system.out.println(name + ":" + i);
          Thread.sleep(1000);
        }
    }
     catch(InterruptedException e)
     {
     system.out.println(name + "Interrupted");
     }
   system.out.println(name + "exiting");
 } }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

```
Class MultiThreadDemo
{
  public static void main (String args[])
  {
    new NewThread ("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");
  try
  {
     // wait for other threads to end
     Thread.sleep (10000);
  }
  catch( InterruptedException e)
  {
    System.out.println("Main thread exiting");
  }
}
```

<u>o/p:</u>

```
New thread: Thread(One,5,main)
New thread: Thread(Two,5,main)
New thread: Thread(Three,5,main)
One : 5
Two : 5
Three : 5
One : 4
Two : 4
Three : 4
One : 3
Two : 3
Three : 3
One : 2
Two : 2
Three : 2
One : 1
Two : 1
Three : 1
One exiting
Two exiting
Three exiting
Main thread exiting.
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

→ Using isAlive() and join()

*) Two ways exist to determine whether a thread has finished. first, we can call isAlive() on the thread. This method is defined by Thread.

general form: final boolean isAlive();

*) The isAlive() method returns true if the thread upon which it is called is still running. It returns false otherwise.

*) while isAlive() is occasionally useful, the method that we will more commonly use to wait for a thread to finish is called join().

final void join() throws InterruptedException.

*) This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it.

[Join - waits for the thread to terminate]

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)
notes4free.in

//using join() to wait for threads to finish.

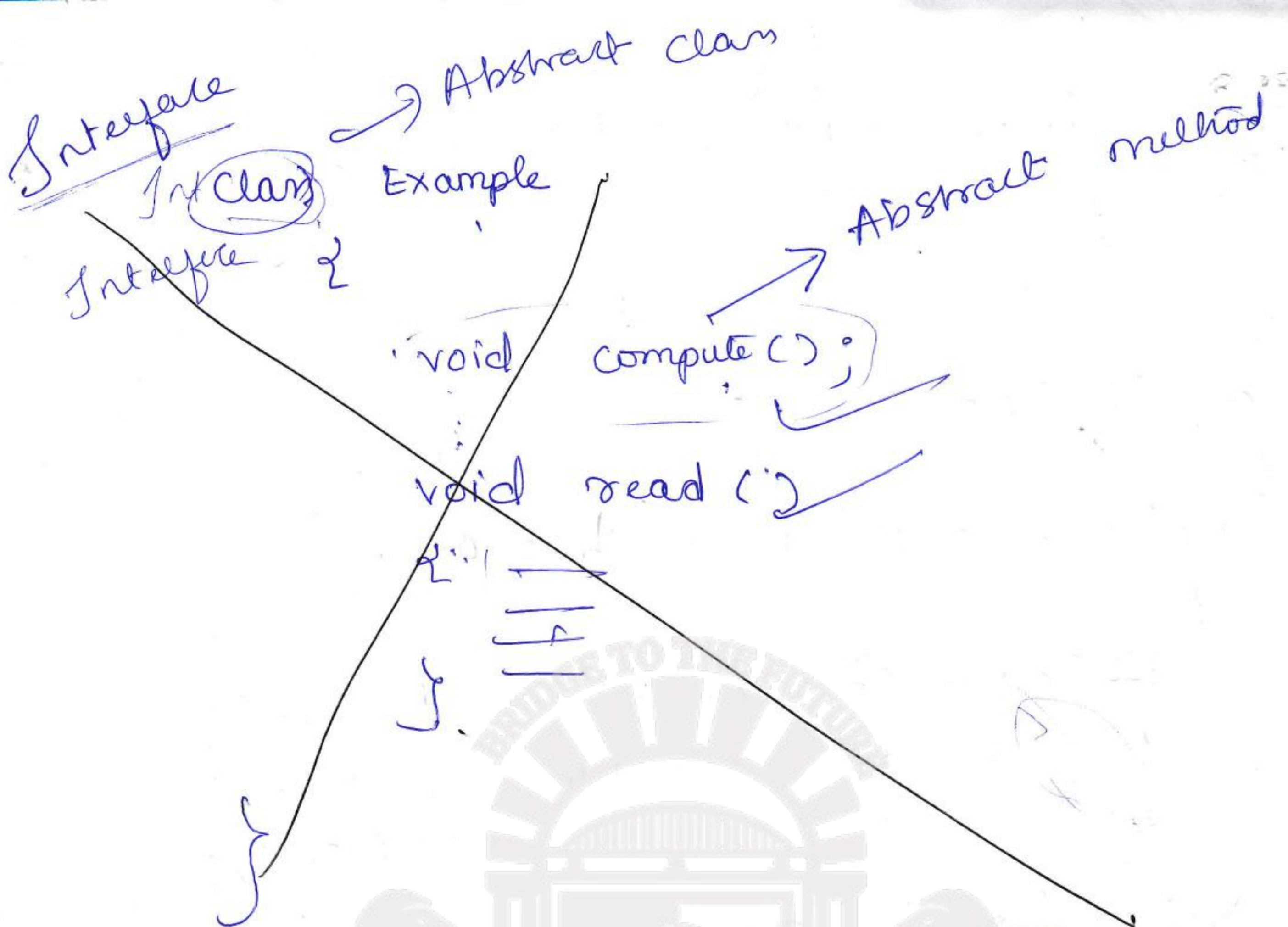
```
Class NewThread implements Runnable
{
   String name;
   Thread t;

   NewThread (String threadname)
   {
      name = threadname;
      t = new Thread(this, name);
      System.out.println("New Thread:" +t);
      t.start(); // start the thread.
   }

   public void run()
   {
      try
      {
         for(int i=5; i>0; i--)
         {
            System.out.println(name + ":" + i);
            Thread.sleep(1000);
         }
      }
      catch (InterruptedException e)
      {
         System.out.println(name + "interrupted");
      }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)
notes4free.in

```
System.out.println(name + " exiting");
}
}

class DemoJoin
{
  public static void main(String args[])
  {
     NewThread ob1 = new NewThread("One");
     NewThread ob2 = new NewThread("Two");
     NewThread ob3 = new NewThread("Three");
  System.out.println("Thread one is alive " + ob1.t.isAlive());
  System.out.println("Thread two is alive " + ob2.t.isAlive());
  System.out.println("Thread three is alive " + ob3.t.isAlive());
  try   // wait for thread to finish.
    {
     System.out.println("waiting for thread to finish");
     ob1.t.join();
     ob2.t.join();
     ob3.t.join();
    }
  catch (InterruptedException e)
  {
   System.out.println("Main thread Interrupted");
  }
  }
```

(SOURCE DIGINOTES)

notes4free.in



System.out.println("Thread one is alive: " + ob1.t.isAlive());
System.out.println("Thread two is alive" + ob2.t.isAlive());
System.out.println("Thread three is alive" + ob3.t.isAlive());

System.out.println("Main thread exiting");
}
}

O/p

New thread: Thread(one, 5, main)
New thread: Thread (Two, 5, main)
New thread: Thread (Three, 5, main)

Thread one is alive: true
" two is alive : true
" three is alive ; true

waiting for threads to finish

One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
.
.
.
One: 1
Two: 1
Three: 1

One exiting
Two exiting
Three exiting.

Thread one is alive: false
Thread two is alive: false
Thread three: false
Main thread exiting.

notes4free.in

→ The ActionEvent class

*) An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.

*) ActionEvent class defines four integer constants that can be used to identify any modifiers associated with action event.
ALT_MASK, CTRL_MASK, META_MASK, SHIFT_MASK.

*) ActionEvent has these two constructors:

ActionEvent (object src, int type, string cmd)
ActionEvent (object src, int type, string cmd, int modifiers)

src → Reference to the object that generated this event.
Type → type of the event.
cmd → command string.
modifier → indicates which modifier keys (ALT, CTRL, SHIFT) were pressed when an event was generated.

*) We can also obtain the command name for the invoking, ActionEvent object by using getActionCommand().

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

notes4free.in

*) The getModifiers() method returns a value that indicates which modifier keys were pressed when an event has generated.

→ The AdjustmentEvent class.

*) An AdjustmentEvent is generated by a scroll bar. There are five types of adjustment events.

*) The AdjustmentEvent class defines integer constants that can be used to identify them.

*) The constants and their meanings are: (ID)

BLOCK_DECREMENT → The user clicked inside the scroll bar to decrement (decrease) its value.

BLOCK_INCREMENT → The user clicked inside the scroll bar to increase its value

TRACK → The slider was dragged.

UNIT_DECREMENT → The button at the end of the scroll bar was clicked to decrease its value

UNIT_INCREMENT → The button at the top of the scroll bar was clicked to increase its value.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

*) AdjustmentEvent has this constructor:

AdjustmentEvent (Adjustable src, int id, int type, int data) → How much scroll down

(→ adjustment event)

Here src → is a reference to the object that generated this event.

Note: There is ADJUSTMENT_VALUE_CHANGED, that indicates that a change has occurred, which is an integer constant.

id → id equals ADJUSTMENT_VALUE_CHANGED.

type & data → type of event & associated data.

*) getAdjustable() method returns the object that generated the event.

*) getAdjustmentType() → Type of the event is obtained.

*) getValue() → amount of Adjustment is obtained.

CAMBRIDGE INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

→ The ComponentEvent class

Component Event (superclass) → mouse, Keyboard, window

*) ComponentEvent is generated when the size, position or visibility of a component is changed.

*) There are four types of component events.

COMPONENT_HIDDEN → The component was hidden.
COMPONENT_MOVED → The component was moved
COMPONENT_RESIZED → The component was resized
COMPONENT_SHOWN → The component became visible.

*) ComponentEvent Constructor is

ComponentEvent(Component src, int type)

*) ComponentEvent is the superclass of ContainerEvent, FocusEvent, KeyEvent, MouseEvent & WindowEvent.

*) getComponent() method returns the component that generated the event.

Component getComponent()

CAMBRIDGE INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

→ The ContainerEvent class

*) A ContainerEvent is generated when a component is added to or removed from a container.

*) There are two types of container events. The ContainerEvent class defines int constants that can be used to identify them:

COMPONENT_ADDED and COMPONENT_REMOVED.

*) Constructor defined as follows:

ContainerEvent(Component src, int type, Component comp)

(src → button / list / panel; type → component add / remove; comp → which contains ?)

comp → component that is added or removed from the container

src → reference to the container which generates this event.

*) getChild() method returns a reference to the component that was added to or removed from the container. (what has added it)

*) We can obtain a reference to the container that generated this event by using getContainer() method.

Note:- Components include lists, buttons, panels & windows, to use components, we need to place them in container. Container is a component that holds & manages other components.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

→ The FocusEvent class : (permanent / temporary focus)

*) A focus event is generated when a component gains or loses input focus.

*) These events are identified by the integer constants FOCUS_GAINED & FOCUS_LOST

*) FocusEvent is a subclass of ComponentEvent and has these constructors.

FocusEvent(Component src, int type)

FocusEvent(Component src, int type, boolean temporaryFlag)

FocusEvent(Component src, int type, boolean temporaryFlag, Component other)

(→ focus gain / focus lost; default = true / false; → permanent focus)

*) The argument temporary flag is set to true if the focus event is temporary. Otherwise, it is set to false (For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, focus is temporarily lost)

(← A → gain the focus; loses; B other unit is other will gain focus)

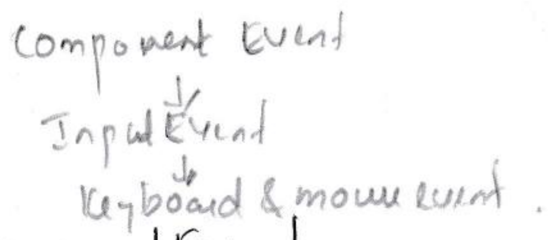
*) getOppositeComponent() gives information about opposite components.

*) boolean isTemporary()

The method returns true if the change is temporary.

notes4free.in

→ The InputEvent class



*) InputEvent is a subclass of ComponentEvent

*) It is the superclass of component input event. Its subclasses are KeyEvent and MouseEvent.

*) InputEvent class defines the following eight integer constants that can be used to obtain information about any modifiers associated with this event:

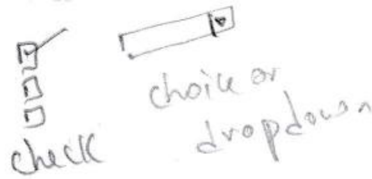
ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

*) isAltDown(), isAltGraphDown(), isControlDown(), ismetaDown() & isshiftDown() methods test if these modifiers were pressed at the time this event was generated.

(SOURCE DIGINOTES)

notes4free.in

→ ItemEvent Class :



*) An ItemEvent is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.

*) There are two types of item events.

DESELECTED → The user deselected an item.

SELECTED. → The user selected an item.

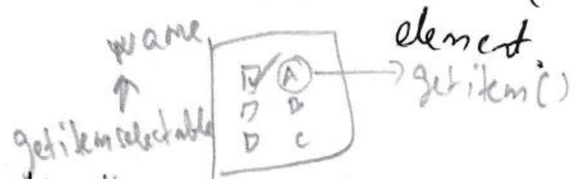
*) ItemEvent defines one integer constant, ITEM_STATE_CHANGED, that signifies a change of state.

*) ItemEvent has this constructor. select / Deselected — what have to select

ItemEvent (ItemSelectable src, int type, Object entry, int state) — selectable & deselectable

*) src → reference to the component that generated this event. ex:- this might be a list or choice element.

type → Type of event.



The specific item that generated the item event is passed in entry

state → Current state of the item.

notes4free.in

*) getItem() → used to obtain a reference to the item that generated an event.

getItemSelectable() → can be used to obtain a reference to the ItemSelectable object that generates an event.

getStateChange() → method returns the state change for the event (selected or not selected).

→ The KeyEvent Class.

*) A KeyEvent is generated when keyboard input occurs.

*) There are three types of key events, which are identified by these integer constants.

virtual key code

KEY_PRESSED, KEY_RELEASED (Ctrl, Alt) & KEY_TYPED (A-Z).

*) The first two events are generated when key is pressed and released.

*) The last event occurs only when a character is generated.

*) There are many other integer constants that are defined by KeyEvent. For example VK_0 through VK_9 and

BRIDGE TO THE FUTURE
CAMBRIDGE INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)
notes4free.in

→ The MouseEvent class (Cont)

int getButton()

It returns a value that represents the button that causes the event. The return value will be one of these constants defined by MouseEvent

NOBUTTON	BUTTON 1	BUTTON 2	BUTTON 3

*) NOBUTTON → value indicates that no button was pressed or released.

*) Java SE 6 added three methods to MouseEvent that obtain the coordinates of the mouse relative to the screen rather than the component. They are

point getLocationOnScreen()
int getXOnScreen()
int getYOnScreen()

*) The getLocationOnScreen() method returns a Point object that contains both the x and y coordinate.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

→ The MouseWheelEvent class.

*) The MouseWheelEvent class, it is a subclass of MouseEvent. Not all mice have wheels. Mouse wheels are used for scrolling.

*) Two integer constants defined are:

WHEEL_BLOCK_SCROLL → A page-up or page down scroll event occurred.

WHEEL_UNIT_SCROLL → A line up or line-down scroll event occurred.

Constructor: MouseWheelEvent(Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup, int scrollHow, int amount, int count)

*) int getWheelRotation()

It returns the number of rotational units. If the value is positive, the wheel moved in counterclockwise. Otherwise clockwise.

*) int getScrollType()

It returns either WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

notes4free.in

→ The MouseEvent Class

*) There are seven types of mouse events.

MOUSE_CLICKED → The user clicked the mouse.
MOUSE_DRAGGED → The user dragged the mouse.
MOUSE_ENTERED → The mouse entered a component
MOUSE_EXITED → mouse exited from a component.
MOUSE_MOVED → mouse was moved
MOUSE_PRESSED → mouse was pressed
MOUSE_RELEASED → The mouse was released

*) Constructor: MouseEvent(Component src, int type, long when, int modifiers, int x, int y, int clicks, boolean triggersPopup)

*) Modifiers → The modifiers argument indicates which modifiers were pressed when a mouse event occurred.

*) The coordinates of the mouse are passed in x & y.

*) The click count is passed in clicks.

*) The triggerspopup flag indicates if this event causes a pop-up menu to appear on this platform.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

*) getx() & gety() methods are used to return the x & y coordinates of the mouse when the event as occured.

*) getpoint() - Alternatively, this method can be used to obtain the coordinates.

*) translatePoint() (int x, int y) method changes the location of the event.

*) getclick count() method obtains the number of mouse clicks for this event. i.e int getclickCount()

*) The TextEvent class

*) These are generated by text fields and text areas when characters are entered by a user or program.

*) constructor :- TextEvent (object src, int type)

Here src is a reference to the object that generated this event, The type of the event is specified by type.

*) TextEvent defines the integer constant TEXT_VALUE_CHANGED.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

→ The WindowEvent Class

*) There are seven types of window events. The constants and their meanings are shown here;

WINDOW_ACTIVATED → The window was activated
WINDOW_CLOSED → window has been closed
WINDOW_CLOSING → user requested that the window be closed.
WINDOW_DEACTIVATED → window was deactivated.
WINDOW_ICONIFIED → The window was iconified.
WINDOW_OPENED → The window was opened.

*) constructor; WindowEvent(Window src, int type)

src → is a reference to the object that generated this event.
type → type of event

*) getWindow() method It returns the window object that generated the event.

→ Sources of Events

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

Event source	Description
1) Button	→ Generates action events when the button is pressed.
2) Checkbox	→ Generates item events when the check box is selected or deselected.
3) choice	→ Generates item events when the choice is changed.
4) list	→ Generates action events when an item is double-clicked, generates item events when item is selected or deselected.
5) Menu item	→ Generates action events when a menu item is selected. Generates item events when a checkable menu item is selected.
6) Scrollbar	→ Generates adjustment events when the scroll bar is manipulated.
7) Text components	→ Generates text events when the user enters the character.
8) window	→ Generates window events when a window is activated, closed, deactivated etc.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)
notes4free.in

→ Event Listener Interfaces

Interface	Description
1) ActionListener →	Defines one method to receive action events.
2) Adjustment listener →	Defines one method to receive adjustment events.
3) ComponentListener →	Defines four methods to recognize when a component is hidden, moved, resized or shown.
4) ContainerListener →	Defines two methods to recognize when a component is added to or removed from a container.
5) FocusListener →	Defines two methods to recognize when a component gains or loses focus.
6) ItemListener →	Defines one method to recognize when ~~a component~~ the state of an item changes
7) KeyListener →	Define three methods to recognize when a key is pressed, released or typed.
8) Mouse listener →	Define five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed or released.

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)
notes4free.in

9). MouseMotionListener → Defines two methods to recognize when the mouse is dragged or moved.

10) TextListener → Defines one method to recognize when a text value changes.

11) WindowListener → Defines seven methods to recognize when a window is activated, closed, deactivated, opened or quit.

Assignment : Event Listener Interfaces

→ Handling Mouse Events.

*) To handle mouse events, we must implement the MouseListener and the MouseMotionListener interface.

*) The following applet demonstrate the process. It displays the current coordinate of the mouse in the applet's status window with various mouse method().

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)
notes4free.in

// Demonstrate the mouse event handlers.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class MouseEvents extends Applet
        implements MouseListener, MouseMotionListener
{
    String msg = " ";
    int x=0, y=0;
  public void init()
  {
    addMouseListener(this);
    addMouseMotionListener(this);
  }
// Mouse Clicked
  public void mouseClicked(MouseEvent me)
  {
    x=0;
    y=10;
    msg= "Mouse clicked";
    repaint();
  }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY
BRIDGE TO THE FUTURE
(SOURCE DIGINOTES)
notes4free.in

```
// Mouse entered
public void mouseEntered (MouseEvent me)
{
  x = 10;
  y = 10;
  msg = "mouse entered";
  repaint();
}
// mouse exited
public void mouseExited (MouseEvent me)
{ x = 10;
  y = 10;
  msg = "Mouse exited";
  repaint();
}
// button pressed.
public void mousePressed (MouseEvent me)
{
  x = me.getX();
  y = me.getY();
  msg = "Down";
  repaint();
}
// button released
public void mouseReleased (MouseEvent me)
{ x = me.getX();
  y = me.getY();
  msg = "Up";
} repaint();
```

```
// mouse dragged.
public void mouseDragged
      (MouseEvent me)
{
  x = me.getX();
  y = me.getY();
  msg = "*";
  showStatus("Dragging mouse at
    " + me.getX() + "," + me.getY());
}
  // Display
public void paint (Graphics g)
{
  g.drawString (msg, x, y);
}
}
```

notes4free.in

→ Handling Keyboard Events

*) To handle keyboard events, we use the same general architecture as shown in mouse event.

*) we have to implement a KeyListener interface.

*) Key events are generated when KEY_PRESSED, KEY_RELEASED & KEY_TYPED.

// Demonstrate the key event handlers

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class SimpleKey extends Applet
    implements KeyListener
{
  String msg = " ";
  int x=10, y=20;
  public void init()
  {
    addKeyListener(this);
  }
```

CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)
notes4free.in

```
public void keypressed (KeyEvent ke)
{
  showstatus ("Key Down");
}
public void KeyReleased (Keyevent ke)
{
  showstatus ("Key up");
}
public void keyTyped (KeyEvent ke)
{
   msg = msg + ke.getKeychar();
   repaint();
}

Display
public void paint (Graphics g)
{
   g.drawstring (msg, x, y);
}
}
```

CAMBRIDGE
INSTITUTE OF TECHNOLOGY
(SOURCE DIGINOTES)

notes4free.in

Applets

- Applet is a small program that
  - can be placed on a web page
  - will be executed by the web browser
  - give web pages "dynamic content".
  - Java Applets enable user interaction with GUI elements
  - Applets are Java programs that can be embedded in HTML documents
  - When browser loads Web page containing applet,Applet downloads into **Web browser** and begins execution or applets can be executed in **appletviewer.**
  - Applets are not stand alone programs.
  - Applets are specified in html document by using applet tag
  - /* <applet code="MyApplet" width=200 height=100>
    </applet> */
  - [thus applet will be executed in java enabled web browser when it encounters applet tag within the html file.]
- **The Applet class**
  - Applet class provides all necessary methods to start and stop the applet program.
  - It also provides methods to load and display images,and play audio clips.
  - Applet extends the AWT class Panel.
  - Panel extends Container which extends Component

Method	**Description**
void destroy( )	Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction.
AccessibleContext getAccessibleContext( )	Returns the accessibility context for the invoking object.
AppletContext getAppletContext( )	Returns the context associated with the applet.
String getAppletInfo( )	Returns a string that describes the applet.
AudioClip getAudioClip(URL url)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.

notes4free.in

Method	Description
AudioClip getAudioClip(URL url, String clipName)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url and having the name specified by clipName.
URL getCodeBase( )	Returns the URL associated with the invoking applet.
URL getDocumentBase( )	Returns the URL of the HTML document that invokes the applet.
Image getImage(URL url)	Returns an Image object that encapsulates the image found at the location specified by url.
Image getImage(URL url, String imageName)	Returns an Image object that encapsulates the image found at the location specified by url and having the name specified by imageName.
Locale getLocale( )	Returns a Locale object that is used by various localesensitive classes and methods.
String getParameter(String paramName)	Returns the parameter associated with paramName. null is returned if the specified parameter is not found.
String[ ] [ ] getParameterInfo( )	Returns a String table that describes the parameters recognized by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description of its type and/or range, and an explanation of its purpose.
void init( )	Called when an applet begins execution. It is the first method called for any applet.
boolean isActive( )	Returns true if the applet has been started. It returns false if the applet has been stopped.
static final AudioClip newAudioClip(URL url)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url. This method is similar to getAudioClip( ) except that it is static and can be executed without the need for an Applet object.

notes4free.in

void play(URL url)	If an audio clip is found at the location specified by url, the clip is played.
void play(URL url, String clipName)	If an audio clip is found at the location specified by url with the name specified by clipName, the clip is played.
void resize(Dimension dim)	Resizes the applet according to the dimensions specified by dim. Dimension is a class stored inside java.awt. It contains two integer fields: width and height.
void resize(int width, int height)	Resizes the applet according to the dimensions specified by width and height.
final void setStub(AppletStub stubObj)	Makes stubObj the stub for the applet. This method is used by the run-time system and is not usually called by your applet. A stub is a small piece of code that provides the linkage between your applet and the browser.
void showStatus(String str)	Displays str in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place.
void start( )	Called by the browser when an applet should start (or resume) execution. It is automatically called after init( ) when an applet first begins.
void stop( )	Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls start().

➢ **Applet Architecture**

- An applet is a window-based program. As such, its architecture is different from the console-based programs
- . Applets are event driven.
- . An applet waits until an event occurs.
- The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet.
- Once this happens, the applet must take appropriate action and then quickly return.
- Applet must perform specific actions in response to events and then return control to the run-time system.
- In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), an additional thread of execution must be started.

notes4free.in

- the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond.
- For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated.
- If the user presses a key while the applet's window has input focus, a keypress event is generated.
- Applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

➢ **An Applet Skeleton.**

- It defines init(),start(),stop(),destroy() methods
- AWT-based applets will override paint() method defined by AWT Component class.This method is called when applet's output must be redisplayed.

```
import java.awt.*;
 import java.applet.*;
/*
<applet code="AppletSkel"       width=300  height=200>
</applet>
*/
 public class AppletSkel extends Applet
 {
 // Called first.
 public void init()
 { // initialization }
 /* Called second, after init(). Also called whenever the applet is restarted. */
public void start()
{ // start or resume execution }
 // Called when the applet is stopped.
 public void stop()
 { // suspends execution }
 /* Called when applet is terminated. This is the last method executed. */
 public void destroy()
 { // perform shutdown activities }
 // Called when an applet's window must be restored.
 public void paint(Graphics g)
 {
 // redisplay contents of window
}
 }
```

When run, it generates the following window when viewed with an applet viewe

notes4free.in

- **Applet Initialization and Termination**
  - Applet methods are called in the following order
  - When an applet begins, the following methods are called, in this sequence:
    - 1. init( )
    - 2. start( )
    - 3. paint( )
  - When an applet is terminated, the following sequence of method calls takes place
    - : 1. stop( )
    - 2. destroy( )

. **init( )**

- The init( ) method is the first method to be called. Initialization of variables is done here. This method is called only once during the run time of applet.

start( )

- The start( ) method is called after init( ). It is also called to restart an applet after it has been stopped. Whereas init( ) is called once—the first time an applet is loaded
- start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ).

paint( )

- The paint( ) method is called each time when applet's output must be redrawn.
- For example, the window in which the applet is running may be overwritten by another window and then uncovered.
- Or the applet window may be minimized and then restored.
- paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called.
- The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running.

stop( )

- The stop( ) method is called when a web browser leaves the HTML document containing the applet—when it goes to another page
- , For example. stop( ) is called,when the applet is probably running. stop( )is used to suspend threads that don't need to run when the applet is not visible.

notes4free.in

destroy( )

- The destroy( ) method is called when the environment determines that the applet needs to be removed completely from memory.
- We should free up any resources the applet may be using.
- The stop( ) method is always called before destroy( ).

Overriding update( )

- AWT, defines a method called update( ). This method is called when applet has requested that a portion of its window be redrawn.
- update( ) method simply calls paint( ).

- **Simple Applet Display Methods**
  - AWT-based applets use AWT to perform input and output.
  - to output a string to an applet, drawString( ) method is used, which is a member of the Graphics class. Typically, it is called from within either update( ) or paint( ).
  - It has the following general form:
    - void drawString(String message, int x, int y) Here, message is the string to be displayed at x,y location. In a Java window, the upper-left corner is location 0,0.
- To set the background color of an applet's window, setBackground( ) method is used.
- To set the foreground color setForeground( ) method is used.
- These methods are defined by Component, and they have the following general forms:
  - void setBackground(Color newColor)
  - void setForeground(Color newColor)
  - Here, newColor specifies the new color.

The class Color defines the constants shown here that can be used to specify colors:

Color.black Color.magenta Color.blue Color.orange Color.cyan
Color.pink Color.darkGray Color.red Color.gray Color.white
Color.green Color.yellow Color.lightGray

- The following example sets the background color to green and the text color to red:
  setBackground(Color.green);
  setForeground(Color.red);
- A good place to set the foreground and background colors is in the init( ) method.
- We can obtain the current settings for the background and foreground colors by calling getBackground( ) and getForeground( ), respectively.

notes4free.in

- They are also defined by Component ,they are:
  Color getBackground( ) Color getForeground( )
- Here is a very simple applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the init( ), start( ), and paint( ) methods are called when an applet starts up:

```
/* A simple applet that sets the foreground and background colors and outputs a string. */
 import java.awt.*;
import java.applet.*;
 /*<applet code="Sample" width=300 height=200>
</applet> */
public class Sample extends Applet
{
 String msg;
 // set the foreground and background colors.
 public void init()
 {
 setBackground(Color.cyan);
setForeground(Color.red);
 msg = "Inside init( ) --";
}
 // Initialize the string to be displayed.
 public void start()
 {
 msg += " Inside start( ) --";0
 }
 // Display msg in applet window.
public void paint(Graphics g)
 {
 msg += " Inside paint( ).";
 g.drawString(msg, 10, 30);
}
 }
```

This applet generates the window shown here: The methods stop( ) and destroy( ) are not overridden, because they are not needed by this simple applet

 notes4free.in

➢ Requesting Repainting

- As a general rule, an applet writes to its window only when its update( ) or paint( ) method is called by the AWT.
- An applet must quickly return control to the run-time system.(constraint)
- It cannot create a loop inside paint( ) that repeatedly scrolls the banner. This would prevent control from passing back to the AWT.
- If applet needs to update the information displayed in the window, it simply calls repaint( ).
- The repaint( ) method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's update( ) method, which by default, calls paint( ).
- The AWT will then execute a call to paint( ), which can display the stored information.
- The repaint( ) method has four forms.
  The simplest version of repaint( ) is shown here:
  void repaint( )
  This version causes the entire window to be repainted.
- The following version specifies a region that will be repainted: void repaint(int left, int top, int width, int height)
- These dimensions are specified in pixels.
- Update may not be called immediately if system is slow or busy. In this case the following forms of repaint( ) are used:
  - void repaint(long maxDelay)
  - void repaint(long maxDelay, int x, int y, int width, int height)
- Here, maxDelay specifies the maximum number of milliseconds that can elapse before update( ) is called.
- It is possible for a method other than paint( ) or update( ) to output to applet window,to do so it must obtain a graphics context by calling getGraphics( ) (defined by Component) and then use this context to output to the window.

➢ A Simple Banner Applet

- To demonstrate repaint( ), a simple banner applet is developed. This applet scrolls a message, from right to left, across the applet's window.
- The scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized.

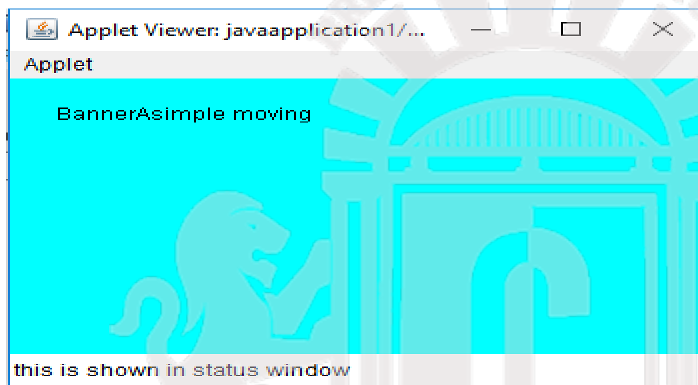
```
import java.awt.*;
import java.applet.*;
 /*<applet code=SimpleBanner  width=300 height=200>
</applet> */
public class SimpleBanner extends Applet implements Runnable
 {
        String msg = " A Simple Moving Banner.";
```


notes4free.in

```
        Thread t = null;
        int state;
        boolean stopFlag;
// Set colors and initialize thread.
        public void init()
        {
                setBackground(Color.cyan);
                 setForeground(Color.red);

        }

// Start thread
        public void start()
        {
                t = new Thread(this);
                stopFlag = false;
                t.start();

        }

// Entry point for the thread that runs the banner.
public void run()

        {

                char ch;
// Display banner
                for( ; ; )
                {
                        try
                        {
                                repaint();
                                 Thread.sleep(250);
                                ch = msg.charAt(0);
                                msg = msg.substring(1, msg.length());
                                 msg += ch;
                                if(stopFlag) break;
                        }
                        catch(InterruptedException e) { }
                }
        }
// Pause the banner.
        public void stop()
```


notes4free.in

```
        {
                stopFlag = true;
                t = null;
        }
// Display the banner.
        public void paint(Graphics g)
        {
                g.drawString(msg, 50, 30);
        }
}
```



- SimpleBanner extends Applet and implements Runnable Interface.
- It is necessary to implement Runnable interface since the applet will be creating a second thread of execution that will be used to scroll the banner.
- Inside init( ), the foreground and background colors of the applet are set.
- After initialization, the run-time system calls start( ) to start the applet running.
- Inside start( ), a new thread of execution is created and assigned to the Thread variable t.
- Then, the boolean variable stopFlag, which controls the execution of the applet, is set to false.
- the thread is started by a call to t.start( ).
- t.start( ) calls run( ) to begin executing.
- Inside run( ), the characters in the string contained in msg are repeatedly rotated left.
- Between each rotation, a call to repaint( ) is made. This eventually causes the paint( ) method to be called, and the current contents of msg are displayed.
- Between each iteration, run( ) sleeps for a quarter of a second.
- The stopFlag variable is checked on each iteration. When it is true, the run( ) method terminates.
- If a browser is displaying the applet when a new page is viewed, the stop( ) method is called, which sets stopFlag to true, causing run( ) to terminate.

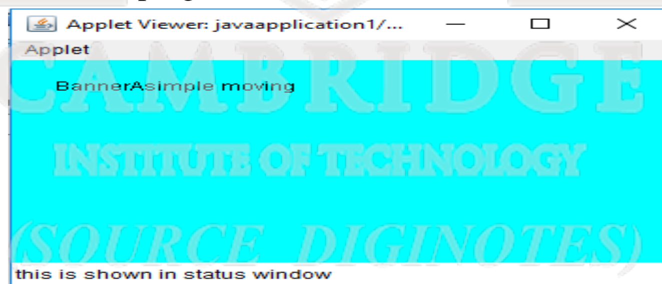
notes4free.in

➢ Using the Status Window

- An applet can also output a message to the status window of the browser or applet viewer on which it is running.
- showStatus( ) method displays the msg in the status window which is passed as a parameter to it.
- The following applet demonstrates showStatus( ):

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/* <applet code StatusWindow width=300 height=100>
</applet>*/
public class StatusWindow extends Applet
        {
                public void init()
                {
                setBackground(Color.cyan);
                }
// Display msg in applet window.
                public void paint(Graphics g)
                {
                g.drawString("This is in the applet window.", 10, 20);
                 showStatus("This is shown in the status window.");
                }
        }
```

Sample output from this program is shown her



➢ The HTML APPLET Tag

- the APPLET tag can be used to start an applet from both an HTML document and from an applet viewer.
- An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page.
- Bracketed items are optional.

```
< APPLET
[CODEBASE = codebaseURL]
```

 notes4free.in

CODE = appletFile
[ALT = alternateText]
[NAME = appletInstanceName]
WIDTH = pixels HEIGHT = pixels
[ALIGN = alignment]
[VSPACE = pixels]
[HSPACE = pixels] >
[< PARAM NAME = AttributeName VALUE = AttributeValue>] [< PARAM NAME = AttributeName2 VALUE =AttributeValue>]

[HTML Displayed in the absence of Java]
</APPLET>

- CODEBASE
  CODEBASE is an optional attribute that specifies the base URL of the applet code
- The HTML document's URL directory is used as the CODEBASE if this attribute is not specified.
- CODE
  CODE is a required attribute that gives the name of the file containing your applet's compiled .class file.
- ALT
  The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets.
- NAME
  NAME is an optional attribute used to specify a name for the applet instance.
  To obtain an applet by name, getApplet( ) methos is ised, which is defined by the AppletContext interface.
- WIDTH and HEIGHT
  WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.
- ALIGN
  ALIGN is an optional attribute that specifies the alignment of the applet.with values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE,
  BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.
- VSPACE and HSPACE
  These attributes are optional.
  VSPACE specifies the space, in pixels, above and below the applet.
  HSPACE specifies the space, in pixels, on each side of the applet.
- PARAM NAME and VALUE
  The PARAM specifies applet-specific arguments in an HTML page. Applets access their attributes with the getParameter( ) method.

notes4free.in

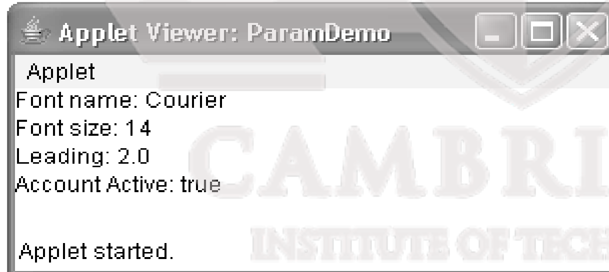
- **Passing Parameters to Applets**
- the APPLET tag in HTML allows to pass parameters to applet.
- To retrieve a parameter, getParameter( ) method is used.
- It returns the value of the specified parameter in the form of a String object.
- Thus, for numeric and boolean values,its need to convert their string representations into their internal formats.
- Here is an example that demonstrates passing parameters:

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/* <applet code="ParamDemo" width=300 height=200>
</applet> */
public class ParamDemo extends Applet
{
        String fontName;
        int fontSize;
        float leading;
        boolean active;
// Initialize the string to be displayed.
        public void start()
        {
                String param;
                fontName = getParameter("fontName");
                if(fontName == null)
                fontName = "Not Found";
                param = getParameter("fontSize");
                try
                {
                        if(param != null)
// if not found
                        fontSize = Integer.parseInt(param);
                        else
                        fontSize = 0;
                }
                catch(NumberFormatException e)
                {
                        fontSize = -1;
                }
                param = getParameter("leading");
                try
```


notes4free.in

```
{
        if(param != null)
// if not found
        leading = Float.valueOf(param).floatValue();
        else leading = 0;
}
catch(NumberFormatException e)
{
        leading = -1;
}
param = getParameter("accountEnabled");
if(param != null)
active = Boolean.valueOf(param).booleanValue();
}
public void paint(Graphics g)
{
g.drawString("Font name: " + fontName, 0, 10);
g.drawString("Font size: " + fontSize, 0, 26);
g.drawString("Leading: " + leading, 0, 42); g.drawString("Account
Active: " + active, 0, 58);
}
```

- conversions to numeric types must be attempted in a try statement that catches NumberFormatException. Uncaught exceptions should never occur within an applet.



Improving the Banner Applet

- It is possible to use a parameter to enhance the banner applet
- However, passing the message as a parameter allows the banner applet to display a different message each time it is executed.
- the APPLET tag specifies a parameter called message that is linked to a quoted string.

```
// A parameterized banner
import java.awt.*;
import java.applet.*;
/* */
 public class ParamBanner extends Applet implements Runnable
```


notes4free.in

```
{
    String msg;
    Thread t = null;
    int state;
    boolean stopFlag;
// Set colors and initialize thread.
    public void init()
    {
        setBackground(Color.cyan);
        setForeground(Color.red);
    }
// Start thread
    public void start()
    {
        msg = getParameter("message");
        if(msg == null)
        msg = "Message not found.";
        msg = " " + msg;
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }
// Entry point for the thread that runs the banner.
    public void run()
    {
        char ch;
// Display banner
        for( ; ; )
        {
        try
        {
            repaint();
            Thread.sleep(250);
            ch = msg.charAt(0);
            msg = msg.substring(1, msg.length());
            msg += ch;
            if(stopFlag) break;
        }
        catch(InterruptedException e) { }

    }
```


notes4free.in

```
}
// Pause the banner.
        public void stop()
        {
                stopFlag = true;
                t = null;
        }
// Display the banner.
        public void paint(Graphics g)
        {
                g.drawString(msg, 50, 30);
        }
}
```

- **getDocumentBase( ) and getCodeBase( )**

- Java allows applet to load data from the directory holding the HTML file that started the applet (the document base)
- and the directory from which the applet's class file was loaded (the code base).
- These directories are returned as URL objects by getDocumentBase( ) and getCodeBase( ).
- To actually load another file, will use the showDocument( ) method defined by the AppletContext interface.

```
import java.awt.*;
import java.applet.*;
import java.net.*;
/*<applet code="Bases"  width=300 height=200> </applet>*/
public class Bases extends Applet
{
        // Display code and document bases.
        public void paint(Graphics g)
        {
                String msg;
                URL url = getCodeBase();
                // get code base
                msg = "Code base: " + url.toString();
                g.drawString(msg, 10, 20);
                url = getDocumentBase();
                // get document base
                msg = "Document base: " + url.toString();
                g.drawString(msg, 10, 40);
        }
```

 notes4free.in

}

Sample output from this program is shown here:

AppletContext and showDocument( )

- One application of Java is to use active images and animation to provide a graphical means of navigating the Web that is more interesting than simple text-based links.
- To allow applet to transfer control to another URL, we use showDocument( ) method defined by the AppletContext interfac
- The context of the currently executing applet is obtained by a call to the getAppletContext( ) method defined by Applet.
- This method has no return value and throws no exception if it fails.
- There are two showDocument( ) methods.
- The method showDocument(URL) displays the document at the specified URL.
- The method showDocument(URL, String) displays the specified document at the specified location within the browser window.
- The methods defined by AppletContext are shown in

Method	**Description**
Applet getApplet(String appletName)	Returns the applet specified by appletName if it is within the current applet context. Otherwise, null is returned.
Enumeration<Applet> getApplets( )	Returns an enumeration that contains all of the applets within the current applet context.
AudioClip getAudioClip(URL url)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.
Image getImage(URL url)	Returns an Image object that encapsulates the image found at the location specified by url.
InputStream getStream(String key)	Returns the stream linked to key. Keys are linked to streams by using the setStream( ) method. A null reference is returned if no stream is linked to key.

 notes4free.in

Iterator<String> getStreamKeys( )	Returns an iterator for the keys associated with the invoking object. The keys are linked to streams. See getStream( ) and setStream( ).
void setStream(String key, InputStream strm)	Links the stream specified by strm to the key passed in key. The key is deleted from the invoking object if strm is null.
void showDocument(URL url)	Brings the document at the URL specified by url into view. This method may not be supported by applet viewers.
void showDocument(URL url, String where)	Brings the document at the URL specified by url into view. This method may not be supported by applet viewers. The placement of the document is specified by where as described in the text.
void showStatus(String str)	Displays str in the status window.

- Upon execution, it obtains the current applet context and uses that context to transfer control to a file called Test.html. This file must be in the same directory as the applet.

```
 import java.awt.*;
 import java.applet.*;
 import java.net.*;
/*<applet code=”ACDemo” width=300 height=200>
</applet> */
public class ACDemo extends Applet
        {
        public void start()
        {
                AppletContext ac = getAppletContext();
                URL url = getCodeBase();
                // get url of this applet
                try
                {
                 ac.showDocument(new URL(url+"Test.html"));
                 }
                catch(MalformedURLException e)
                {
                showStatus("URL not found");
```


notes4free.in

```
            }
        }
        }
```

The AudioClip Interface

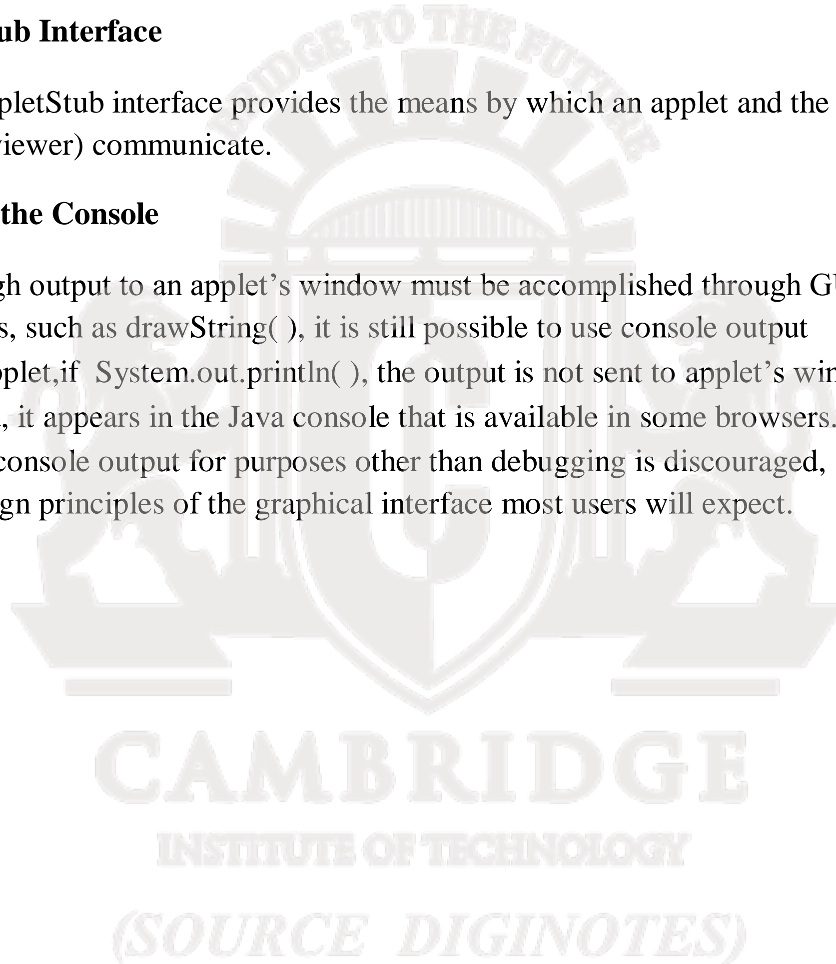
- The AudioClip interface defines these methods: play( ) (play a clip from the beginning), stop( ) (stop playing the clip), and loop( ) (play the loop continuously).
- After its loaded ,using getAudioClip( ), we can use these methods to play it.

The AppletStub Interface

- The AppletStub interface provides the means by which an applet and the browser (or applet viewer) communicate.

Outputting to the Console

- Although output to an applet's window must be accomplished through GUI-based methods, such as drawString( ), it is still possible to use console output
- In an applet,if System.out.println( ), the output is not sent to applet's window.
- Instead, it appears in the Java console that is available in some browsers.
- Use of console output for purposes other than debugging is discouraged, since it violates the design principles of the graphical interface most users will expect.



 notes4free.in

Unit ---3

Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT.

In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables.

Even familiar components such as buttons have more capabilities in Swing. For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.

Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and, therefore, are platform-independent. The term *lightweight* is used to describe such elements.

Swing are built on AWT.

Explain two key features of Swing.

1. Swing components are light weight

   They are entirely written in java they does not map to native platform specific code. More flexible and more efficient. Not in rectangular shapes.

2. Swing supports a pluggable look and feel.

   It becomes possible to change the that component is rendered with out affecting any of its other aspects. Possible to create new look and feel for any given component with out side effects. Look and feel is simply plugged in.

Briefly explain Container and Component of Swing.

1. A Swing GUI consists of two key items: Components and Container
2. A term component is an independent visual control such as push button or slider.
3. A container holds group of components. Thus container is special kind of component that holds that is designed to hold other components. Container are also called components so container can hold other container.

   Components

   1. Swing components are derived from JComponent Class. Supports pluggable look and feel. It inherits Component and Container of AWT.

notes4free.in

Swing Components : JButton, JCheckBox, JComboBox ,JTree ,JLabel,
JTable ,JPanel ……..etc

Container

1. Swing defines two types of heavy weight container.
   JFrame , JApplet
2. Others are light weight containers.
3. Top level containers should be declared first like JFrame and JApplet.
4. Light weight containers example if JPanel. This is used to manage group of related components
5. JPanel is used to create subgroups of related components that are contained with in another container.

Examples of containers

JPanel is Swing's version of the AWT class Panel and uses the same default layout,FlowLayout. JPanel is descended directly from JComponent.

JFrame is Swing's version of Frame and is descended directly from that class. The components added to the frame are referred to as its contents; these are managed by the contentPane. To add a component to a JFrame, we must use its contentPane instead.

JWindow is Swing's version of Window and is descended directly from that class. LikeWindow, it uses BorderLayout by default.

JDialog is Swing's version of Dialog and is descended directly from that class. Like Dialog, it uses BorderLayout by default. Like JFrame and JWindow,

What is swing and what is its application?

Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit (AWT).

Swing provides a native look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform.

notes4free.in

It has more powerful and flexible components than AWT. In addition to familiar components such as buttons, check boxes and labels, Swing provides several advanced components such as tabbed panel, scroll panes, trees, tables, and lists.

Explain MVC architecture of swing:

1. In general a visual component is composite of three distinct aspects
   The way that the component looks when rendered on the screen
   The way the component reacts to the user.
   The state information associated with the component
2. The architecture has proven itself to be effective is MVC
3. MVC mean Model View Controller.
4. Model corresponds to the state information associated with the component. For example in case of check Box model contained a field that indicates if check box ix checked or unchecked.
5. View determines how the component is displayed on the screen
6. Controller determines how the component react to the user after that result in the view is updated.
7. By separating model , view , and controller the specific implementation of one model can be changed with out affecting other model.
8. The MVC architecture sounds good but in swing separating view and controller is not beneficial.
9. Swing uses modified version of MVC called UI delegate. For this reason swings approach is called **Model delegate** architecture.
10. Swings pluggable look and feel is possible by its model delegate architecture.
11. Because view and controller are separate look and feel can be changed without affecting the component

notes4free.in

Class	Description
AbstractButton	Abstract super class for Swing buttons.
ButtonGroup	Encapsulates a mutually exclusive set of buttons.
Image	Icon encapsulates an icon.
JApplet	The Swing version of Applet.
JButton	The Swing push button class.
JCheckBox	The Swing check box class.
JComboBox	Encapsulates a combo box (an combination of a drop-down listand text field).
JLabel	The Swing version of a label.
JRadioButton	The Swing version of a radio button.
JScrollPane	Encapsulates a scrollable window.
JTabbedPane	Encapsulates a tabbed window.
JTable	Encapsulates a table-based control.
JTextField	The Swing version of a text field.
JTree	Encapsulates a tree-based control.

The Swing-related classes are contained in **javax.swing**

JLabel

Swing labels are instances of the **JLabel** class, which extends **JComponent**. It can display text and/or an icon. Some of its constructors are shown here:

JLabel(Icon i)

Label(String s)

JLabel(String s, Icon i, int align)

Here, *s* and *i* are the text and icon used for the label.

The *align* argument is either **LEFT**,**RIGHT**, or **CENTER**

The text associated with the label can be read and written by the following methods: String getText( ) ,void setText(String s)

Example Program

import javax.swing.*;

```
import javax.swing.*;
class SwingDemo
{
SwingDemo()
```


notes4free.in

```
{
        JFrame jfrm=new JFrame("A simple Swing Application");
        jfrm.setSize(275,100);
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JLabel jlab=new JLabel("Swing means power ful GUI");
        jfrm.add(jlab);
        jfrm.setVisible(true);
}
public static void main(String args[]) {
 ob= new SwingDemo();
}
}
```

Text Fields

The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**.

It provides functionality that is common to Swing text components.

One of its subclasses is **JTextField**, which allows you to edit one line of text. Some of its constructors are shown here:

JTextField( )

JTextField(int cols)

JTextField(String s, int cols)

JTextField(String s)

Here, *s* is the string to be presented, and *cols* is the number of columns in the text field.

The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a **JTextField** object is created and is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
```


notes4free.in

```
</applet>
*/
public class JTextFieldDemo extends JApplet {
JTextField jtf;
public void init() {
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
jtf = new JTextField(15);
contentPane.add(jtf);
}
}
```

Output from the above program

JButton

The **JButton** class provides the functionality of a push button. **JButton** allows an icon, a string, or both to be associated with the push button. Some of its constructors are shown here:

JButton(Icon i)

JButton(String s)

JButton(String s, Icon i)

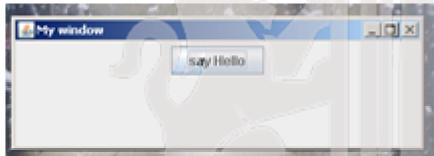
Here, *s* and *i* are the string and icon used for the button.

The following program displays a button on swing frame and displays string "SayHello"

```
import javax.swing.*;
import java.awt.*;
public class First {
	JFrame jf;
	public First()
	{
```


notes4free.in

```
        jf=new JFrame("My window");
        JButton btn= new JButton("say Hello");
        jf.add(btn);
        jf.setLayout(new FlowLayout());
        jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jf.setSize(400,400);
        jf.setVisible(true);
        }
        public static void main(String[] args) {
                    new First();
        }

}
```



Check Boxes

The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of **AbstractButton**. Some of its constructors are shown here:

JCheckBox(Icon i)

JCheckBox(Icon i, boolean state)

JCheckBox(String s)

JCheckBox(String s, boolean state)

JCheckBox(String s, Icon i)

JCheckBox(String s, Icon i, boolean state)

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the check box is initially selected. Otherwise, it is not. The state of the check box can be changed via the following method: void setSelected(boolean state) Here, *state* is **true** if the check box should be checked.

1.The following example illustrates how to create an applet that displays four check boxes and a text field. When a check box is pressed, its text is displayed in the text field.

2.) flow layout is assigned as its layout manager.

notes4free.in

3.).four check boxes are added to the content pane, and icons are assigned for the normal, rollover, and selected states. The applet is then registered to receive item events.

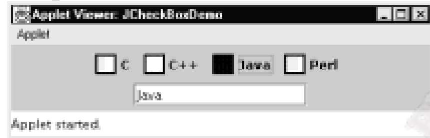
Finally, a text field is added to the JFrame. When a check box is selected or deselected, an item event is generated. This is handled by **itemStateChanged( )**. Inside **itemStateChanged( )**, the **getItem( )** method gets the **JCheckBox** object that generated the event. The **getText( )** method gets the text for that check box and uses it to set the text inside the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo extends JFrame
implements ItemListener {
JTextField jtf;
JCheckBoxDemo()
{
        setLayout(new FlowLayout());


        JCheckBox cb = new JCheckBox("C");
        cb.addItemListener(this);
        add(cb);

        cb = new JCheckBox("C++");
        cb.addItemListener(this);
        add(cb);

        cb = new JCheckBox("Java");
        addItemListener(this);
        add(cb);

        cb = new JCheckBox("Perl", normal);
        addItemListener(this);
        add(cb);

        jtf = new JTextField(15);
        add(jtf);
}

public void itemStateChanged(ItemEvent ie) {
```


notes4free.in

```
        JCheckBox cb = (JCheckBox)ie.getItem();
        jtf.setText(cb.getText());
}
Public static void main(String args[])
        {
        new JCheckBoxDemo();
        }
}
```

Output



Combo Boxes

1. Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**.

2. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can also type your selection into the text field.

3. Two of **JComboBox**'s

constructors are shown here:

JComboBox( )

JComboBox(array a)

Here, *a* is a array that initializes the combo box.

4. Items are added to the list of choices via the **addItem( )** method, whose signature is shown here: void addItem(Object obj)

Here, *obj* is the object to be added to the combo box.

T he following example contains a combo box and a label. The label displays an icon. The combo box contains entries for "France", "Germany", "Italy", and "Japan". When a country is selected, the label is updated to display the flag for that country.

```
import java.awt.*;
import java.awt.event.*;
```


notes4free.in

```
import javax.swing.*;

public class JComboBoxDemo extends JFrame
implements ItemListener {
JLabel jl;
ImageIcon france, germany, italy, japan;

public void init() {
setLayout(new FlowLayout());

JComboBox jc = new JComboBox();
jc.addItem("France");
jc.addItem("Germany");
jc.addItem("Italy");
jc.addItem("Japan");
jc.addItemListener(this);
add(jc);

jl = new JLabel();
add(jl);
}
public void itemStateChanged(ItemEvent ie) {
String s = (String)ie.getItem();
jl.setText(s);
}
}
```

- Output is shown here:

Radio Buttons

Radio buttons are supported by the **JRadioButton** class, which is a concrete implementation of **AbstractButton**. Some of its constructors are shown here:

```
JRadioButton(Icon i)
JRadioButton(Icon i, boolean state)
JRadioButton(String s)
JRadioButton(String s, boolean state)
JRadioButton(String s, Icon i)
```


notes4free.in

JRadioButton(String s, Icon i, boolean state)

Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time.

For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected.

The **ButtonGroup** class is instantiated to create a button group. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

void add(AbstractButton ab)

Here, *ab* is a reference to the button

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonDemo extends JFrame
implements ActionListener
{
JTextField tf;
JRadioButtonDemo()
setLayout(new FlowLayout());
JRadioButton b1 = new JRadioButton("A");
b1.addActionListener(this);
add(b1);
JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this);
add(b2);
JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
add(b3);

// Define a button group

ButtonGroup bg = new ButtonGroup();

bg.add(b1);
bg.add(b2);
bg.add(b3);

// Create a text field and add it
// to the frame
tf = new JTextField(5);

add(tf);
}
public void actionPerformed(ActionEvent ae) {
tf.setText(ae.getActionCommand());
}
}
```

Output from this applet is shown here:

notes4free.in

Tabbed Panes

1. A *tabbed pane* is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.
2. Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. We will use its default constructor. Tabs are defined via the following method: void addTab(String str, Component comp)

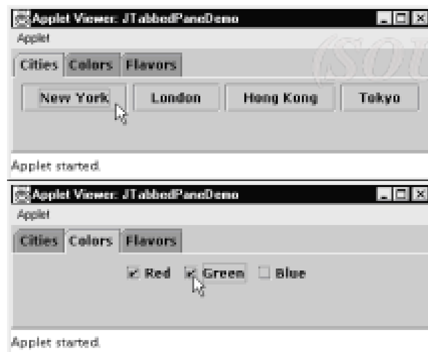
Here, *str* is the title for the tab, and *comp* is the component that should be added to the tab. Typically, a **JPanel** or a subclass of it is added. The general procedure to use a tabbed pane in an applet is outlined here:

1. Create a **JTabbedPane** object.
2. Call **addTab( )** to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet. The following example illustrates how to create a tabbed pane.

```
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/


public class JTabbedPaneDemo extends JApplet {
public void init() {
JTabbedPane jtp = new JTabbedPane();
jtp.addTab("Cities", new CitiesPanel());
jtp.addTab("Colors", new ColorsPanel());
jtp.addTab("Flavors", new FlavorsPanel());
getContentPane().add(jtp);
}
}


class CitiesPanel extends JPanel {
```


notes4free.in

```
public CitiesPanel() {
JButton b1 = new JButton("New York");
add(b1);
JButton b2 = new JButton("London");
add(b2);
JButton b3 = new JButton("Hong Kong");
add(b3);
JButton b4 = new JButton("Tokyo");
add(b4);
}
}


class ColorsPanel extends JPanel {
public ColorsPanel() {
JCheckBox cb1 = new JCheckBox("Red");
add(cb1);
JCheckBox cb2 = new JCheckBox("Green");
add(cb2);
JCheckBox cb3 = new JCheckBox("Blue");
add(cb3);
}
}

class FlavorsPanel extends JPanel {
public FlavorsPanel() {
JComboBox jcb = new JComboBox();
jcb.addItem("Vanilla");
jcb.addItem("Chocolate");
jcb.addItem("Strawberry");
add(jcb);
}
}
```



CAMBRIDGE INSTITUTE OF TECHNOLOGY (SOURCE DIGINOTES)

notes4free.in

Scroll Panes

1. A *scroll pane* is a component that presents a rectangular area in which a component may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary.
2. Scroll panes are implemented in Swing by the **JScrollPane** class, which extends **JComponent**. Some of its constructors are shown here:

   JScrollPane(Component comp)

Here are the steps that you should follow to use a scroll pane in an applet:

1. Create a **JComponent** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)
3. Add the scroll pane to the content pane of the applet.

The following example illustrates a scroll pane. First, the content pane of the **JApplet** object is obtained and a border layout is assigned as its layout manager. Next, a **JPanel** object is created and four hundred buttons are added to it, arranged into twenty columns. The panel is then added to a scroll pane, and the scroll pane is added to the content pane. This causes vertical and horizontal scroll bars to appear. You can use the scroll bars to scroll the buttons into view.

(SOURCE DIGINOTES)

```
import javax.swing.*;

import java.awt.BorderLayout;
import java.awt.GridLayout;


public class JScrollPaneDemo extends JApplet{

	public void init()
	{
		JPanel jp=new JPanel();
		jp.setLayout(new GridLayout(20,20));
```


notes4free.in

```
        add(jp);

                int b=0;
                for(int i=0;i<20;i++)
                        for(int j=0;j<20;j++)
                        {
                        jp.add(new JButton("Button"+b));
                                ++b;
                        }
                JScrollPane jsp=new JScrollPane(jp);
                add(jsp,BorderLayout.CENTER);

        }


}
```

output

Tables

1. A *table* is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Tables are implemented by the **JTable** class, which extends **JComponent**. One of its constructors is shown here:

   JTable(Object data[ ][ ], Object colHeads[ ])

   Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

2. Here are the steps for using a table in an applet:

- Create a **JTable** object.

notes4free.in

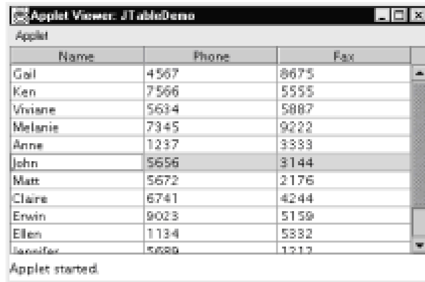
- Create a **JScrollPane** object.
- Add the table to the scroll pane.
- Add the scroll pane to the content pane of the applet.

The following example illustrates how to create and use a table. array of strings is created for the column headings. This table has three columns. A two-dimensional array of strings is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane and then the scroll pane is added to the content pane.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JTableDemo extends JFrame {

	JTable jtbl;
	final String[] colHeads = { "Name", "Phone", "Fax" };

final Object[][] data = {{ "Gail", "4567", "8675" },{ "Ken", "7566", "5555" },{ "Viviane", "5634",
"5887" },
{ "Melanie", "7345", "9222" },{ "Anne", "1237", "3333" },{ "John", "5656", "3144" },{ "Matt", "5672",
"2176" },
{ "Claire", "6741", "4244" },{ "Erwin", "9023", "5159" },{ "Ellen", "1134", "5332" },{ "Jennifer",
"5689", "1212" },{ "Ed", "9030", "1313" },{ "Helen", "6751", "1415" }};

	JTableDemo()
	{
		setTitle("MY window");

		setSize(400,400);
		setLayout(new FlowLayout());
		jtbl=new JTable(data,col);
		add(jtbl);
		setVisible(true);
		setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	}
	public static void main(String[] args) {
		// TODO Auto-generated method stub
	 new	JTableDemo();

	}

}
```


notes4free.in

Out put



University questions

1. Difference between Swing and AWT(4)
2. Explain MVC architecture of swing(6)
3. Explain different types of swing button (10) (JButton, JToggleButton, JCheckBox, JRadioButton)
4. What is swing? Explin Compnents and Containers in the swing (8)
5. Explain following component with example(12)
   i) JTextField
   ii) JButton Class
   iii) JComboBox
6. How AWT is different from swings? what are the two key features of it? Explain.(08 Marks)
7. List four types of buttons in swings with their use. write a program to create four different types of buttons on JApplet use suitable event to show action on JLabel (12 Marks
8. Discuss about swing features. List its components and container.(10)
9. Develop an applet to create a text field ,Label box and 4 check boxes with the caption “red”, “blue”,”yellow”,”green”(10)
10. Create a swing applet that has two buttons named alpha and beta when either of the buttons pressed it should display “ alpha was pressed and beta was pressed respectively.(8)
11. Write steps to create JTable. Write a program to create a table with the column headings “ FName, LName, Age” and insert atleast five tuples.(6)

notes4free.in