B M S INSTITUTE OF TECHNOLOGY & MANAGEMENT

YELAHANKA, BENGALURU – 560064.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE -1 NOTES OF

OBJECT ORIENTED CONCEPTS -18CS45

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2018 -2019)

SEMESTER – IV

Prepared by,

Mr. Muneshwara M S

Asst. Prof, Dept. of CSE

VISION AND MISSION OF THE CS&E DEPARTMENT

Vision

To develop technical professionals acquainted with recent trends and technologies of computer science to serve as valuable resource for the nation/society.

Mission:

Facilitating and exposing the students to various learning opportunities through dedicated academic teaching, guidance and monitoring.

VISION AND MISSION OF THE INSTITUTE

Vision

To emerge as one of the finest technical institutions of higher learning, to develop engineering professionals who are technically competent, ethical and environment friendly for betterment of the society.

Mission

Accomplish stimulating learning environment through high quality academic instruction, innovation and industry-institute interface

MODULE 1 INTRODUCTION TO OBJECT ORIENTED CONCEPTS

The following concepts should be learn in this Module

A Review of structures, Procedure–Oriented Programming system, Object Oriented, Programming System, Comparison of Object Oriented Language with C, Console I/O, variables and reference variables, Function Prototyping, Function Overloading. Class and Objects: Introduction, member functions and data, objects and functions.

Text book 1: Ch 1: 1.1 to 1.9 Ch 2: 2.1 to 2.3 , RBT: L1, L2

A REVIEW OF STRUCTURES

Consider a function nextday( ) that accepts the addresses of 3 integers that represent a date and changes these values to represent nextday.

Prototype of this function //for calculating the next day
Suppose

If we call
d=1; m=1;
y=2002; //1st january 2002
.

Members of wrong group may be accidentally sent to the function

```
d1=28; m1=2; y1=1999;        //28thfeb99
d2=19; m2=3; y2=1999;        //19thmrch99
nextday(&d1,&m1,&y1);        //ok
nextday(&d1,&m2,&y2); //incorrect set passed
```

There is nothing in language itself that prevents the wrong set of variables from being sent to the function.
So we need structures

1) Putting structure definition and prototypes of associated functions in a header file(date.h)

```
Struct date
{
int d, m, y;
};
Void nextday(struct date *);
```

2) Put definition and other prototypes in a source code and create a library

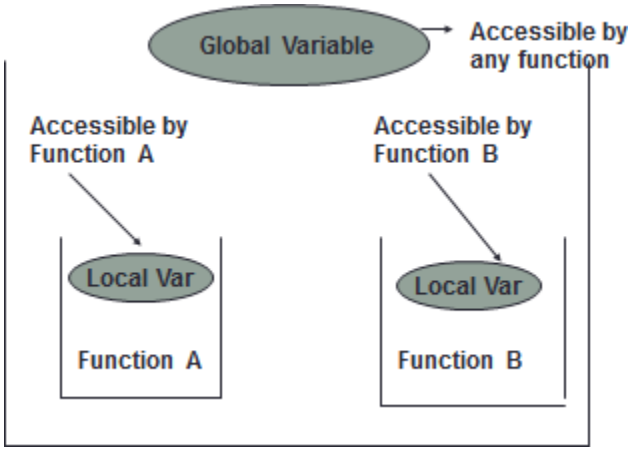
```
void main()
{
Struct date d1;
d1.d=28;
d1.m=2;        //Need for structures
d1.y=1999;
nextday(&d1);
}
```

3) Using Structures in Application Programs

- Include header file provided by programmer in the source code.
- Declare variables of new data type in the source code
- Embed calls to the associated functions by passing these variables in the source code

PROCEDURE/ STRUCTURE ORIENTED PROGRAMMING

- Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as procedure-oriented programming (POP).
- In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks.
- The primary focus is on functions.
- Dividing a program into functions and modules is one of the cornerstones of Structured Programming.
- Since many functions in a program can access global data / global variables, global data can be corrupted by that have no business to change it.
- Hence we need a way to restrict the access to the data, to hide it from all but a few critical functions. This protects the data, simplifies the maintenance and other benefits.

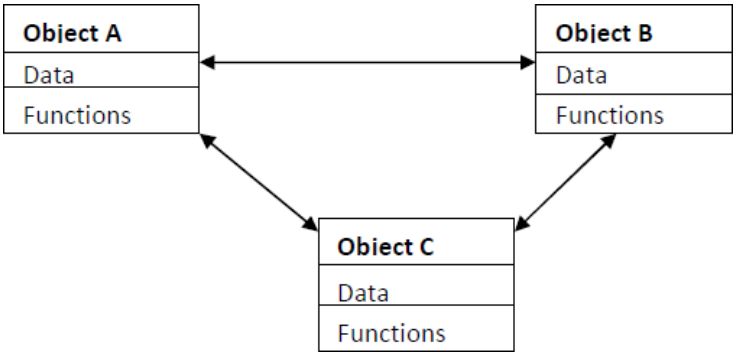


Drawback/Disadvantage

1. Data is not secure and can be manipulated by any function/procedure.
2. Associated functions that were designed by library programmer don‟t have rights to work upon the data.
3. They are not a part of structure definition itself because application program might modify the structure variables by some code inadvertently written in application program itself

OBJECT ORIENTED PROGRAMMING

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions(methods).
- An object functions are called Member functions in C++.
- Data and functions are said to be encapsulated in an object.
- Data Encapsulation & Data Hiding are the key terms in OOPs



OOPs simplify writing, debugging and maintaining the program. C++ program typically contain a number of objects interacting with each other by calling one another‟s member functions.

	Procedure Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.

Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function,it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Objects

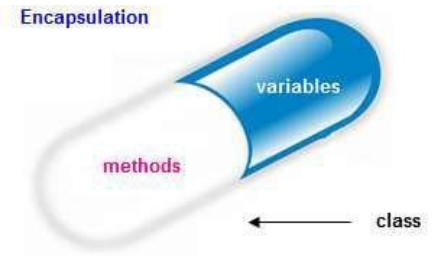
Objects are the basic runtime entities in an object oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

Class

Object contains data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class.

Data Encapsulation

- The wrapping up of data and functions into a single unit is known as encapsulation.
- The data is not accessible to the outside world, only those function which are wrapped in the class can access it.
- These functions provide the interface between the object's data and the program.
- It hides the implementation details of an object from its users.
- Encapsulation prevents unauthorized access of data or functionality.
- This insulation of the data from direct access by the program is called **data hiding** or **information hiding**.

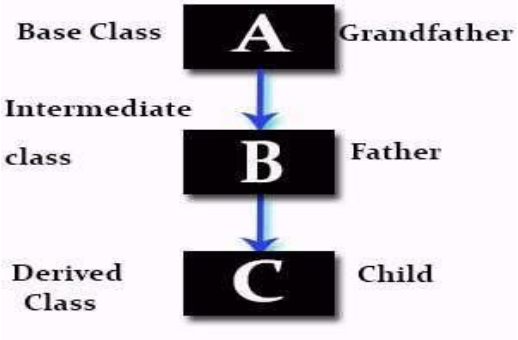


Data Abstraction

- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Since classes use the concept of data abstraction, they are known as **Abstract Data Types(ADT)**.
- It separates unnecessary details or explanations so as to reduce complexities of understanding requirements.

Inheritance

- **Inheritance** is the process by which objects of one class acquire the properties of objects of another class.
- Parent class can be given the general characteristics, while its child may be given more specific characteristics.
- Inheritance mechanism makes it possible for one object to be a specific instance of a more general class.
- Reusability – adding additional features to an existing class without modifying it. That is, deriving a new class from the existing one. The new class will have the combined features of both the classes. New class is also called as **Derived** class and existing class is called as **Base** Class.



Polymorphism

- **Polymorphism,** a Greek term means to ability to take more than one form.
- An operation may exhibits different behaviors in different instances. The behavior depends upon the type of data used in the operation.. **Ex: Function Overloading**
- For example consider the operation of addition for two numbers; the operation will generate a sum. If the operands are string then the operation would produce a third string by concatenation.
- The process of making an operator to exhibit different behavior in different instances is known **operator overloading.**

C	C++
1. C compiler cannot execute C++ programs	**1. C++ compiler can execute C programs**
2. In C, u may / may not include function prototypes	**2. In C++, you must Include function prototypes**
3. C doesn't allow for default arguments	**3. C++ lets you to specify default arguments in function prototype**
4. Declaration of the variables must be at the beginning	**4. Declaration of the variables can be anywhere before using**

5. If a C program uses a Local variable that has Same name as global variable, then C uses the value of a local variable.	**5. In C++, u can instruct program to use value of global variable with scope resolution Ex- cout << "Iamglobal var :" << ::I;**
6. Fun overloading is not there.	**6. Fun overloading exists**
7. Function inside the structure is not allowed	**7. Function inside the structure is allowed**
8. Object initialization doesn't exist	**8. Object initialization (constructor) exist**
9. Data hiding, data abstraction and data encapsulation feature doesn't exist	**9. Data hiding, data abstraction and data encapsulation exists in C++**

CONSOLE INPUT AND OUTPUT

Console output

The predefined object *cout* is an instance of ostream class. The *cout* object is said to be "connected to" the standard output device, which usually is the display screen.

```
cout << constant/variable
cout<<endl;
cout<<"\n";\\newline
```

Console input

The predefined object cin is an instance of istream class. The cin object is said to be attached to the standard input device, which usually is the keyboard.

```
cin>> variable
```

Example :

```
#include <iostream.h>
void main ()
{
 int x;
 cout << "Please enter an integer value: ";
 cin >>x;
 cout <<endl<< "Value you entered is " <<x;
 cout << " and its double is " <<x*2 << ".\n";
}
```

Variables in C++

- Can be declared anywhere in the C++ program before using them.

Reference variable

- They are used as aliases for other variables within a function.
- All operations supposedly performed on the alias (i.e., the reference) are actually

performed on the original variable.

- An alias is simply another name for the original variable.
- Must be initialized at the time of declaration.
- Reference variable can be a function argument and thus change the value of the parameter that is passed to it in the function call.

Syntax :

datatype &variable= existing variable

Example

Ex : #include <iostream.h>
void main()
{
int x=3;
int &y=x;
cout<<"x="<<x endl<<"y="<<y<<endl;
y=7;
cout<<"x="<<x<<endl<<"y="<<y<<endl;
}

Output :
x=3
y=3
x=7
Y=7

Ex: #include <iostream.h>
void main()
{
int x, y;
int x=100;
int & iRef=x;
y=iRef;
cout <<y<<endl;
y++; // x and iRef unchanged
cout <<x <<endl<<iRef<<endl<<y<<endl;
}
100
100
100
101

```
#include  <iostream>
using namespace std;
int main ()
{
int i; double d;
int& r = i;
double& s = d;
i = 5;
cout << "Value of i : " << i << endl;
cout << "Value of i reference : " << r << endl;
d = 11.7;
cout << "Value of d : " << d << endl;
cout << "Value of d reference : " << s << endl;
return 0;
}
```

Output:
Value of i : 5
Value of i reference : 5
Value of d : 11.7
Value of d reference : 11.7

Swap 2 variables using reference variables

```
#include <iostream>
using namespace std;
void swap(int &x, int &y);

int main () {
     int a = 100;
     int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;
```

```
    swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

   return 0;
}
void swap(int &x, int &y) {
   int temp;
            temp = x;
            x = y;
            y = temp;
}
```

FUNCTION PROTOTYPING

- C++ strongly supports function prototypes
- Prototype describes the function‟s interface to the compiler
- Tells the compiler the return type of function, number , type and sequence of its formal arguments

Syntax : return_type function_name(argument_list);
Eg- int add (int, int);

With prototyping, compiler ensures following .The return value of a function is handled correctly. Correct number and type of arguments are passed to a function. Since C++ compiler requires function prototyping, it will report error against function call because no function prototype is provided to resolve the function call. Prototyping guarantees protection from errors arising out of incorrect function calls. A function heading without body.

FUNCTION OVERLOADING IN C++

C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading. However to achieve this they must have different signatures.
Signature of a function means number, type and sequence of formal arguments of the function.

Ways to overload a function

- By changing number of Arguments.
- By having different types of argument.

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

void display(int);
void display(float);
void display(int, float);
```

```
int main() {

   int a = 5;
   float b = 5.5;

   display(a);
   display(b);
   display(a, b);
   return 0;
}

void display(int var) {
   cout << "Integer number: " << var << endl;
}

void display(float var) {
   cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

INLINE FUNCTIONS

Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced. All the functions defined inside class definition are by default inline, but you can also make any non-class function inline by using keyword inline with them. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called.

For an inline function, declaration and definition must be done together. For example,

Syntax:

```
inline return_type function_name(arguments)
{
//function body
}
```

Example:

```
inline double cube(double a)
{
return(a * a * a );
}
```

Why to use –

In many places we create the functions for small work/functionality which contain simple and less number of executable instruction. Imagine their calling overhead each time they are being called by callers. With inline keyword, the compiler replaces the function call statement with the function code itself (process called expansion) and then compiles the entire code. Thus, with inline functions, the compiler does not have to jump to another location to execute the function, and then jump back as the code of the called function is already available to the calling program.

CLASS AND OBJECTS

A class is a way to bind the data and its associated functions together. It allows the data and functions to be hidden, if necessary, from external use. A class declaration is similar syntactically to a structure.

General form of a class declaration is:

class class_name
{
private:
Variable declaration/data members;
Function declaration/ member functions;
protected:
Variable declaration/data members;
Function declaration/ member functions;
public:
Variable declaration/data members;
Function declaration/ member functions
};
	Private members can be accessed only from within the class.
	
	Protected members can be accessed by own class and its derived classes.
	
	Public members can be accessed from outside the class also.

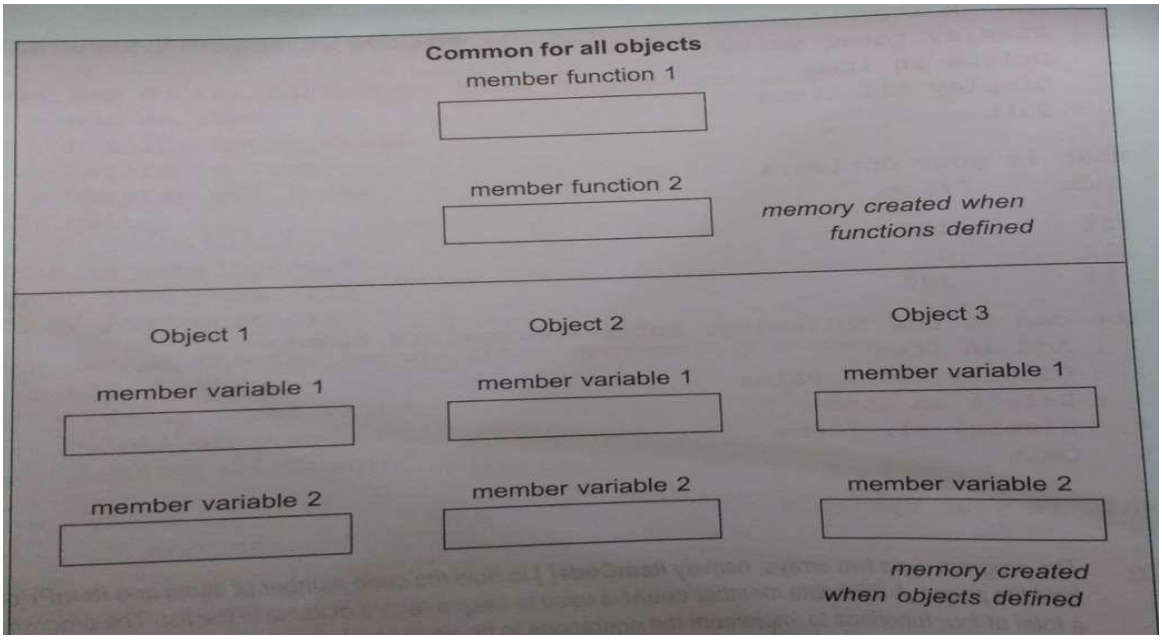
- The variables declared inside the class definition are known as **data members** and the functions declared inside a class are known as **member functions**.
- By default the data members and member function of a class are **private**.
- Private data members can be accessed by the functions that are wrapped inside the class.

How to access member of a class?

To access member of a class dot operator is used. i.e.

object-name . data-member ***and***
object-name . member-function

Memory allocation of objects:



Memory space for objects is allocated when they are declared and not when the class is specified. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created.
Only space for member variables(data members) is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects.

General steps to write a C++ program using class and object:

- Header files
- Class definition
- Member function definition
- void main function

ex:
```
#include<iostream.h>
class Add
{
int x, y, z;
public:
 void getdata()
{
cout<<"Enter two numbers";
cin>>x>>y;
}
void calculate(void);
void display(void);
};
```

```
void Add :: calculate()
{
z=x+y;
}

void Add :: display()
{
cout<<z;
}
void main()
{
    Add a;
     a.getdata();
     a.calculate();
     a.display();
}
```

Output:
Enter two numbers 5 6
11

Private & Public Members

- The private keyword makes data and functions private. Private data and functions can be accessed only from inside the same class.

```
class Student
{
 private:    // private data member
 int rollno;

 public:     // public accessor functions
 int getRollno()
 {
  return rollno;
 }

 void setRollno(int i)
 {
  rollno=i;
 }

};

int main()
{
 Student A;
 A.rollono=1;                    //Compile time error
 cout<< A.rollno;                //Compile time error

 A.setRollno(1);                   //Rollno initialized to 1
 cout<< A.getRollno();             //Output will be 1
}
```

- The public keyword makes data and functions public. Public data and functions can be accessed out of the class.

```
class Student
{
 public:
        int rollno;
 string name;
};

int main()
{
 Student A;
 Student B;
 A.rollno=1;
 A.name="Adam";

 B.rollno=2;
 B.name="Bella";

 cout <<"Name and Roll no of A is :"<< A.name << A.rollno;
 cout <<"Name and Roll no of B is :"<< B.name << B.rollno;
}
```

Scope Resolution Operator (::)

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

The definition of member function outside the class differs from normal function definition, as the function name in the function header is preceded by the class name and the scope resolution operator (: :). The scope resolution operator informs the compiler what class the member belongs to. The syntax for defining a member function outside the class is

```
Return_type class_name :: function_name (parameter_list)
{
// body of the member function
}
```

In C++, scope resolution operator is **::**. It is used for following purposes.

1) **To access a global variable when there is a local variable with same name**
2) **To define a function outside a class.**

3) **To access a class's static variables. [refer static data members]**
4) **In case of multiple Inheritance**

Global variable access:

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

int x; // Global x

int main()
{
  int x = 10; // Local x
  cout << "Value of global x is " << ::x;
  cout << "\nValue of local x is " << x;
  return 0;
}
```

Define function outside the class

```
class Box {
   public:
      double length;        // Length of a box
      double breadth;       // Breadth of a box
      double height;        // Height of a box
      double getVolume(void);
       void setdata( double l, double b, double h );
};
          double Box:: getVolume(void)
           {
                return length * breadth * height;
                    }

void Box::setLength(double l, double b, double h)
 {
   length = l;
breadth = b;
height = h;
      }

int main( )
 {
   Box Box1;
double volume = 0.0;
Box1.setdata(6.0,5.0,8.3);

volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;
  }
```

This pointer

The „this" pointer-The facility to create and call member functions of class objects is provided by the compiler. Compiler does this by using a unique pointer -> this.
this pointer - always a constant pointer. It points at the object with respect to which the function was called.
The „this" pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. „this" pointer is a constant pointer that holds the memory address of the current object. „this" pointer is not available in static member functions as static member functions can be called without any object (with class name).

- It puts a declaration of the this pointer as a leading formal argument in the prototypes of all member functions as follows

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

It passes the address of invoking object as a leading parameter to each call to the member functions.
Ex : setFeet(**&d1,** 1);

```
class Test
{
private:
  int x;
public:
  void setX (int x)
  {
     this->x = x;
  }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj;
  int x = 20;
  obj.setX(x);
  obj.print();
  return 0;
}
```

MEMBER FUNCTIONS AND MEMBER DATA

1) Default values for formal arguments of Member functions

Default values can be assigned to arguments of non-member functions and member functions. Member functions should be overloaded with care, if default values are specified for some or all of its arguments otherwise the compiler will report ambiguity error.

Ex

```
Class A
  {
    int sum(int x, int y, int z=0)
    {
        return (x + y + z );
    }
};
    int main()
    {
      A A1;
      A1. sum(10, 15) << endl;
      A1. sum(10, 15, 25) << endl;
      return 0;
      }
```

Output: 25
50

Points to remember

- Default values are specified from right to left.
- Default values must be specified in the function prototypes and not in function definitions.
  - **int mul(int i, int j=5, int k=10); // Valid**
  - **int mul(int i=5, int j); //InValid**
  - **int mul(int i=0, int j, int k=10); // InValid**
  - **int mul(int i=2, int j=5, int k=10); // Valid**

2) Constant Member functions:

A function becomes const when const keyword is used in function"s declaration. The idea of const functions is not allow them to modify the object on which they are called.

The programmer desires that one of the member functions of a class should not be able to change the value of data members. This function should be able to merely read the values contained in the data members, but not change them.

```
Class Distance
{
   int iFeet;
   float fInches;
        public:
                void setdata(int,float);
                void getdata() const;              //constant function
        };

         void Distance::setdata(int x, float y)
         {
             iFeet=x;
            fInches=y;
          }
         void Distance::getdata()       const          //const function
          {
               iFeet++;                  //ERROR!!
              fInches=fInches+ 1;        //ERROR!!
              Cout<<iFeet;
           }
```

3) Mutable Data Members

Mutable data member is never constant. It can be modified inside constant functions also. Prefixing the declaration of a data member with the key word mutable makes it mutable.

```
Class A
{
                int  x;
                mutable int y;
        public:
                void abc() const              //a constant member function
                {
                        x++;          // error: cant modify a non-constant
```

```
                y++;        //ok can modify a mutable data member in
        }
        void def()      //can modify
        {
                x++;
                y++;
        }
};
```

4) Friend Class & Friend Functions:

Friend Function

- Scope of a friend function is not inside the class in which it is declared. It is prefixed with the keyword **friend.**
- Since its scope is not inside the class, it cannot be called using the object of that class. It can be called like a normal function without using any object.
- It cannot directly access the data members like other member function and it can access the data members by using object through dot operator. 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
- It can be declared either in private or public part of the class definition. Usually it has the objects as arguments.

```
class abc
{
   ....
   ....
public:
   ....
   ....
   friend void xyz(void);  // declaration
};
```

```
class Add
{
int x, y, z;
public:
Add(int, int);
friend int calculate(Add p);
};

Add :: Add(int a, int b)
{
x=a;
y=b;
}
```

```
int calculate(Add p)
{
return(p.x+p.y);
}

void main()
{
Add a(5, 6);
cout<<calculate(a);
}
```

Output:
11

Friend Class: A class can be friend of another class. Member Functions of a friend class can access private data members of objects of class of which it is a friend**.** When this is the case, the friend class and all of its member functions have access to the private members defined within the other class.

```
class B;              //Forward declaration.
 class A
   {
     friend class B;
   //rest of the class A
    };
```

Friendship is not transitive. That is, If class A is friend with class B, and class B is friend with class C. This doesn't mean that class A is friend with class C.

```
#include <iostream>
using namespace std;
class Rectangle;
class Square
{
        friend class Rectangle;     // declaring Rectangle as friend class
        int side;
        public:
                Square ( int s )
                {
                        side = s;
                }
};
```

```
class Rectangle
{
        int length;
        int breadth;
        public:
        int getArea()
        {
                return length * breadth;
        }
        void shape( Square a )
        {
                length = a.side;
                breadth = a.side;
        }
};

int main()
{
        Square square(5);
        Rectangle rectangle;
        rectangle.shape(square);
        cout << rectangle.getArea() << endl;
        return 0;
}
```

Output:
25

5) Static Data Members

- The data member of a class preceded by the keyword **static** is known as static member.
- When we precede a member variable's declaration with static, we are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Hence static variables are called class variables.
- Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use that same variable.
- All static variables are initialized to zero before the first object is created.

```
class A
{
    int  p;
    static int q;
public:
```

```
        A();
        void incr(void);
        void display(void);
};

   A :: A()
      {
        p=5;
      }
int A:: q=10;          // initialize static data member

void A:: incr()
     {
         p++;
          q++;
      }

void A:: display()
      {
        cout<<p<<"\t"<<q<<endl;
       }
void main()
{
A a1, a2, a3;
a1.incr();
a1.display();
a2.incr();
a2.display();
a3.incr();
a3.display();
}
```

Output:

6 11
6 12
6 13

Static Member function/method

- A static function can have access to only other static members (functions or variables) declared in the same class. (Of course, global functions and data may be accessed by static member Function.). It is accessed by class name and not by object‟s name .

  i.e. **class-name::function-name**

- The function name is preceded by the keyword **static**. A static member function does not have this pointer.

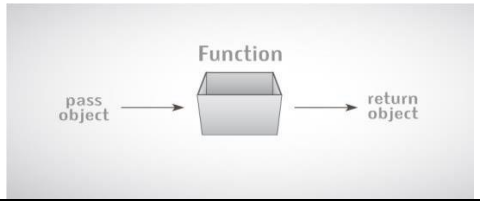
```
#include <iostream>
using namespace std;

class Counter
{
        private:
            static int count;               //static data member as count

        public:
                Counter()                   //default constructor
                {
                    count++;
                  }
        static void Print()                 //static member function
                {
                        cout<<"\nTotal objects are: "<<count;
                }
};

int Counter :: count = 0;               //count initialization with 0

int main()
{
        Counter OB1;
        OB1.Print();

        Counter OB2;
        OB2.Print();

        Counter OB3;
        OB3.Print();

        return 0;
}
```

Output:

```
Total objects are: 1
Total objects are: 2
Total objects are: 3
```

OBJECTS & FUNCTIONS:

Objects appear as local variables. They can also be passed by reference to Functions. Finally they can be returned by value or by reference from the functions.




```
#include  <iostream>
using namespace std;
class Complex
{
   private:
            int real;
            int imag;
   public:
            void readData()
            {
                    cout << "Enter real and imaginary number
respectively:"<<endl;
                     cin >> real >> imag;
             }

             void addComplexNumbers(Complex comp1, Complex comp2)
            {
                     real=comp1.real+comp2.real;
                     imag=comp1.imag+comp2.imag;
             }
      void displaySum()
      {
         cout << "Sum = " << real<< "+" << imag << "i";
      }
  };

int main()
{
   Complex c1,c2,c3;
   c1.readData();
   c2.readData();
   c3.addComplexNumbers(c1, c2);
   c3.displaySum();
   return 0;
}
```

Output:
1
2
1
2
Sum= 2+4i

```
                y++;        //ok can modify a mutable data member in
        }
        void def()      //can modify
        {
                x++;
                y++;
        }
};
```

6) Friend Class & Friend Functions:

Friend Function

- Scope of a friend function is not inside the class in which it is declared. It is prefixed with the keyword **friend.**
- Since its scope is not inside the class, it cannot be called using the object of that class. It can be called like a normal function without using any object.
- It cannot directly access the data members like other member function and it can access the data members by using object through dot operator. 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
- It can be declared either in private or public part of the class definition. Usually it has the objects as arguments.
-

```
class abc
{
   ....
   ....
public:
   ....
   ....
   friend void xyz(void);  // declaration
};
```

```
class Add
{
int x, y, z;
public:
Add(int, int);
friend int calculate(Add p);
};

Add :: Add(int a, int b)
{
x=a;
y=b;
}
```

```
int calculate(Add p)
{
return(p.x+p.y);
}

void main()
{
Add a(5, 6);
cout<<calculate(a);
}
```

Output:
11

Friend Class: A class can be friend of another class. Member Functions of a friend class can access private data members of objects of class of which it is a friend**.** When this is the case, the friend class and all of its member functions have access to the private members defined within the other class.

```
class B;              //Forward declaration.
 class A
   {
     friend class B;
   //rest of the class A
    };
```

Friendship is not transitive. That is, If class A is friend with class B, and class B is friend with class C. This doesn't mean that class A is friend with class C.

```
#include <iostream>
using namespace std;
class Rectangle;
class Square
{
        friend class Rectangle;     // declaring Rectangle as friend class
        int side;
        public:
                Square ( int s )
                {
                        side = s;
                }
};
```

```
class Rectangle
{
        int length;
        int breadth;
        public:
        int getArea()
        {
                return length * breadth;
        }
        void shape( Square a )
        {
                length = a.side;
                breadth = a.side;
        }
};

int main()
{
        Square square(5);
        Rectangle rectangle;
        rectangle.shape(square);
        cout << rectangle.getArea() << endl;
        return 0;
}
```

Output:
25

7) Static Data Members

- The data member of a class preceded by the keyword **static** is known as static member.
- When we precede a member variable's declaration with static, we are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Hence static variables are called class variables.
- Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use that same variable.
- All static variables are initialized to zero before the first object is created.

```
class A
{
    int  p;
    static int q;
public:
```

```
        A();
        void incr(void);
        void display(void);
};

   A :: A()
      {
        p=5;
      }
int A:: q=10;          // initialize static data member

void A:: incr()
     {
         p++;
          q++;
      }

void A:: display()
      {
        cout<<p<<"\t"<<q<<endl;
       }
void main()
{
A a1, a2, a3;
a1.incr();
a1.display();
a2.incr();
a2.display();
a3.incr();
a3.display();
}
```

Output:

```
6 11
6 12
6 13
```

Static Member function/method

- A static function can have access to only other static members (functions or variables) declared in the same class. (Of course, global functions and data may be accessed by static member Function.). It is accessed by class name and not by object‟s name .

  i.e. **class-name::function-name**

- The function name is preceded by the keyword **static**. A static member function does not have this pointer.

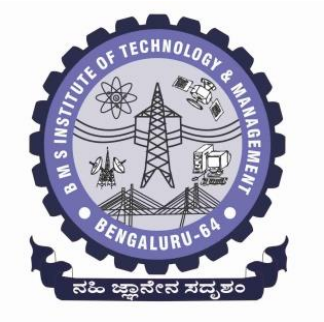
```
#include <iostream>
using namespace std;

class Counter
{
        private:
            static int count;               //static data member as count

        public:
                Counter()                   //default constructor
                {
                    count++;
                  }
        static void Print()                 //static member function
                {
                        cout<<"\nTotal objects are: "<<count;
                }
};

int Counter :: count = 0;               //count initialization with 0

int main()
{
        Counter OB1;
        OB1.Print();

        Counter OB2;
        OB2.Print();

        Counter OB3;
        OB3.Print();

        return 0;
}
```

Output:

```
Total objects are: 1
Total objects are: 2
Total objects are: 3
```

B M S INSTITUTE OF TECHNOLOGY & MANAGEMENT

YELAHANKA, BENGALURU – 560064.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE -2 NOTES OF

OBJECT ORIENTED CONCEPTS -18CS45

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2018 -2019)

SEMESTER – IV

Prepared by,

Mr. Muneshwara M S

Asst. Prof, Dept. of CSE

VISION AND MISSION OF THE CS&E DEPARTMENT

Vision

To develop technical professionals acquainted with recent trends and technologies of computer science to serve as valuable resource for the nation/society.

Mission:

Facilitating and exposing the students to various learning opportunities through dedicated academic teaching, guidance and monitoring.

VISION AND MISSION OF THE INSTITUTE

Vision

To emerge as one of the finest technical institutions of higher learning, to develop engineering professionals who are technically competent, ethical and environment friendly for betterment of the society.

Mission

Accomplish stimulating learning environment through high quality academic instruction, innovation and industry-institute interface

MODULE 2 CLASS AND OBJECTS (CONTD)

The following concepts should be learn in this Module

Objects and arrays, Namespaces, Nested classes, Constructors, Destructors. **Introduction to Java**: Java's magic: the Byte code; Java Development Kit (JDK); the Java Buzzwords, Object-oriented programming; Simple Java programs. Data types, variables and arrays, Operators, Control Statements.

Text book 1:Ch 2: 2.4 to 2.6Ch 4: 4.1 to 4.2

Text book 2: Ch:1 Ch: 2 Ch:3 Ch:4 Ch:5 , RBT: L1, L2

OBJECTS AND ARRAYS:

Like array of other user-defined data types, an array of type class can also be created. The array of type class contains the objects of the class as its individual elements. Thus, an array of a class type is also known as an array of objects. An array of objects is declared in the same way as an array of any built-in data type.

The syntax for declaring an array of objects is

class_name array_name [size] ;

```
class Employee
{
char name[30];
int age;
public:
void getdata(void);
void putdata(void);
};
void Employee:: getdata(void)
{
cout<<"Enter Name and Age:";
cin>>name>>age;
}
void Employee:: putdata(void)
{
cout<<name<<"\t"<<age<<endl;
}
void main()
```

```
{
Employee e[5];
int i;
for(i=0; i<5; i++)
{
e[i].getdata();
}
for(i=0; i<5; i++)
{
e[i].putdata();
}
}
```

NAMESPACE:

Namespaces – enable C++ programmer to prevent pollution of global namespace that lead to name clashes.

Global namespace refer to the entire source code. It includes all the directly and indirectly included header files. By default, name of each class is visible in the source code i.e. in the global space. This can lead to problems.

Namespace is used to define a scope where identifiers like variables, functions, classes, etc are declared. The main purpose of using a namespace is to prevent ambiguity that may occur when two identifiers have same name.

Consider

A1.h A2.h

class A

{

//body of A

}; class A

{

//body of A

};

Main_prog.cpp

#include “A1.h” #include “A2.h” main()

{

A Aobj; //ERROR: Ambiguity error due to multiple definitions of A

}

Syntax of Namespace Definition:

The member can be accessed in the program as,

using namespace namepace_name namespace_name::member_name;

/*A1.h*/

namespace A1

{

class A

{

};

} /*end of namespace A1.h*/ /*A2.h*/

namespace A2

{

class A

{

};

} **/*end of namespace A2.h*/**

The using directive enable us to make class definition inside A namespace visible so that qualifying the name of referred Class by name of namespace is no longer required. Code below tells how this is done.

#include “A1.h” #include “A2.h” void main()

```
{
using namespace A1;
A1::A Aobj1;        //ok: Aobj1 is an object of class defined in A1.h
A2::A Aobj2;        //ok: Aobj2 is an object of class defined in A1.h
}
```

Rules for namespace:

- Namespace declarations appear only at global scope.
- Namespace declarations can be nested within another namespace.
- Namespace declarations don‟t have access specifiers. (Public or private)
- No need to give semicolon after the closing brace of definition of namespace.

using namespace std;

The using namespace statement specifies that the members defined in std namespace will be used frequently throughout the program.

CONSTRUCTORS:

A class constructor is a special member function of a class that is executed whenever we create new objects of that class. Constructors can be very useful for setting initial values for certain member variables.

Constructors are special class functions which performs initialization of every object. The Compiler calls the Constructor whenever an object is created.

Characteristics of a constructor

- They should be declared in the public section.
- They are invoked directly when an object is created.
- They don‟t have return type, not even void and hence can‟t return any values.
- They can‟t be inherited; through a derived class, can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors can‟t be virtual.
- Constructors can be inside the class definition or outside the class definition.
- Constructor can‟t be friend function.

- They make implicit calls to the operators new and delete when memory allocation is required.
- When a constructor is declared for a class, initialization of the class objects becomes necessary after declaring the constructor.

Different Types of Constructor

Default Constructor:-

Default Constructor is also called as Empty Constructor which has no arguments and It is Automatically called when we creates the object of class but Remember name of Constructor is same as name of class and Constructor never declared with the help of Return Type.

```
class Add
{
int x, y, z;
public:
Add(); // Default Constructor
void display(void);
};
Add::Add()
{
x=6;
y=5;
}
void Add :: display()
{
cout<< x+y;
}
void main()
{
Add a;
a.display();
}
```

Output:

11

Parameterized Constructor:-

This is Another type Constructor which has some Arguments and same name as class name .We have to create object of Class by passing some Arguments at the time of creating object with the name of class.

```
 class Add
{
Int x, y, z;
public:
Add(int, int);
void display(void);
};
Add :: Add(int a, int b)
{
x=a;
y=b;
}
void Add :: display()
{
cout<<x+y;
}
void main()
{
Add a(5, 6);
a.display();
}
```

Output:

11

A parameterized constructor can be called:

(i) Implicitly: Add a(5, 6);

(ii) Explicitly : Add a=Add(5, 6);

Copy Constructor:-

This is also Another type of Constructor. In this Constructor we pass the object of class into the Another Object of Same Class. As name Suggests you Copy, means Copy the values of one Object into the another Object of Class .

When we are using or passing an Object to a Constructor then we must have to use the & Ampersand or Address Operator.

```
class Add
{
int x, y, z;
public:
Add(int a, int b)
{
x=a;
y=b;
}


Add(Add &);
void display(void);
};
Add :: Add(Add &p)
{
x=p.x;
y=p.y;
cout<<"Value of x and y for new object: "<<x<<" and
"<<y<<endl;
}
void Add :: display()
{
cout<<x+y;
}
void main()
{
Add a(5, 6);
Add b(a);
b.display();
}
```

Output:

Value of x and y for new object are 5 and 6 11

DESTRUCTORS

- It is a special member function which is executed automatically when an object is destroyed.
- Its name is same as class name but it should be preceded by the symbol ~.
- It cannot be overloaded as it takes no argument.
- It is used to delete the memory space occupied by an object.
- It has no return type.
- It should be declared in the public section of the class.

```
 class A
{
A()
{
cout << "Constructor called";
}
~A()
{
cout << "Destructor called";
}

};
int main()
{
A obj1; // Constructor Called
int x=1
if(x)
{
A obj2; // Constructor Called
} // Destructor Called for obj2
} // Destructor called for obj1
```

NESTED CLASSES:

A class can be defined inside another class. Such a class is known as Nested Class. The class that contains the nested class is known as enclosing class. Nested classes can be defined in the private, protected, or public sections of the enclosing class.

- A nested class is declared inside another class.
- The scope of inner class is restricted by the outer class.
- While declaring an object of inner class, the name of the inner class must be preceded by the outer class name and scope resolution operator.

A nested class is created if it does not have any relevance outside its enclosing class. By defining class as a nested class, name collision can be avoided. That is, if there is class B defined as global class, its name will not clash with the nested class B. The size of the object of an enclosing class is not affected by the presence of nested classes.

How are the member functions of a nested class defined?

Member functions of a nested class can be defined outside the definition of enclosing class.

Syntax:

```
class A
{
public:
class B
{
public:
void BTest();
};
};      // Defining function outside the class


void A :: B :: BTest()
{
//function body
}


#include <iostream.h>
 class Nest
{       public:
class Display
{       private:
int s;
public:
```

```
void sum( int a, int b)
{       s =a+b;
}
void show( )
{       cout << "\nSum of a and b is:: " << s;
}
};
};
void main()
{       Nest::Display x;
x.sum(12, 10);
x.show();
}
```

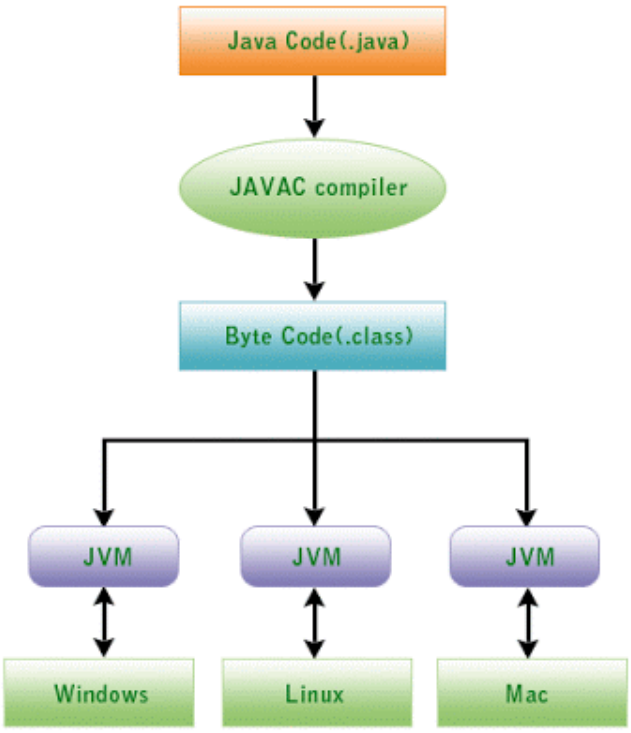
Sum of a and b is::22

INTRODUCTION TO JAVA

Java Byte Code:

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Byte code 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*).In essence, the original JVM was designed as an interpreter for bytecode.

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.

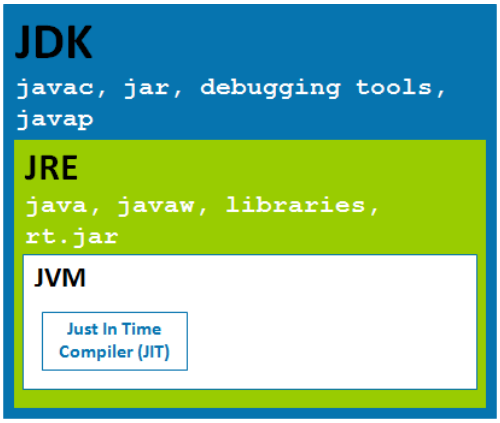


Bytecode is the intermediate representation of Java programs just as assembler is the intermediate representation of C or C++ programs. Programming code, once compiled, is run through a virtual machine instead of the computer's processor. By using this approach, source code can be run on any platform once it has been compiled and run through the virtual machine.

Bytecode is the compiled format for Java programs. Once a Java program has been converted to bytecode, it can be transferred across a network and executed

by Java Virtual Machine (JVM). Bytecode files generally have a .class extension.

A method's bytecode stream is a sequence of instructions for the Java virtual machine. Each instruction consists of a one-byte opcode followed by zero or more operands. Rather than being interpreted one instruction at a time, Java bytecode can be recompiled at each particular system platform by a just-in-time compiler. Usually, this will enable the Java program to run faster.



Java Development Kit (JDK):

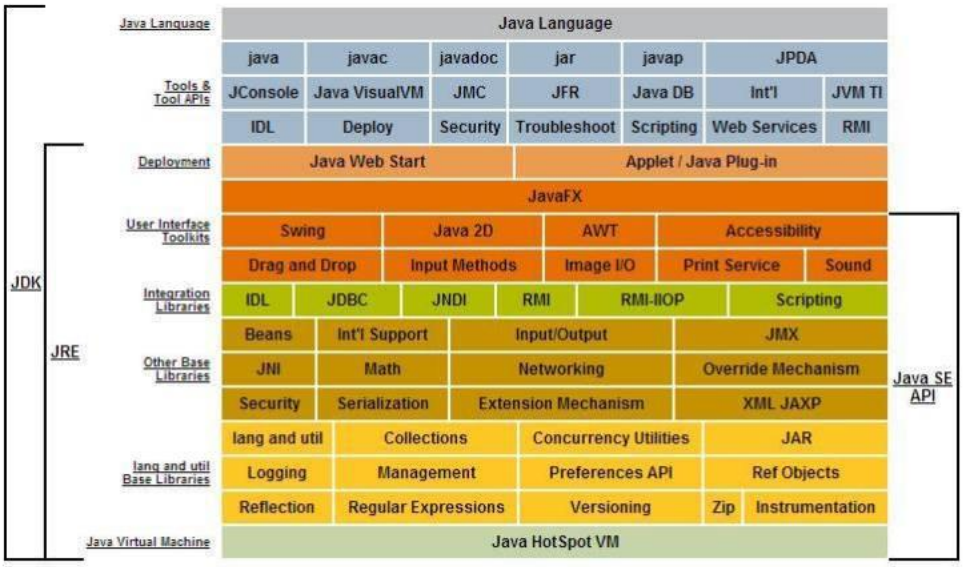
Java Development Kit contains two parts. One part contains the utilities like **javac**, debugger, **jar** which helps in compiling the source code (**.java** files) into byte code (**.class** files) and debug the programs. The other part is the **JRE**, which contains the utilities like **java** which help in running/executing the byte code. If we want to write programs and run them, then we need the JDK installed.

Java Run-time Environment (JRE):

Java Run-time Environment helps in running the programs. **JRE** contains the **JVM**, the java classes/packages and the run-time libraries. If we do not want to write programs, but only execute the programs written by others, then **JRE** alone will be sufficient.

Java Virtual Machine (JVM):

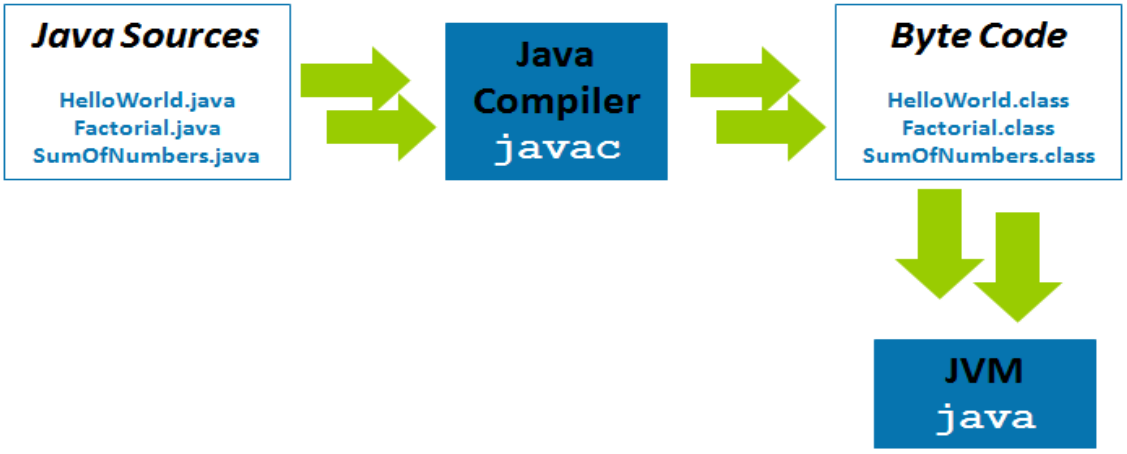
Java Virtual Machine is important part of the **JRE**, which actually runs the programs (**.class** files), it uses the java class libraries and the run-time libraries to execute those programs. Every operating system(OS) or **platform** will have a different **JVM**.



Just In Time Compiler (JIT):

JIT is a module inside the **JVM** which helps in **compiling** certain parts of byte code into the **machine code** for higher performance. Note that only certain parts of byte code will be compiled to the machine code, the other parts are usually **interpreted** and executed.

Java is distributed in two packages - **JDK** and **JRE**. When **JDK** is installed it also contains the **JRE**, **JVM** and **JIT** apart from the compiler, debugging tools. When **JRE** is installed it contains the **JVM** and **JIT** and the class libraries. **javac** helps in compiling the program and **java** helps in running the program. When the words **Java Compiler**, **Compiler** or **javac** is used it refers to **javac**, when the words **JRE**, **Run-time Enviroment**, **JVM**, **Virtual Machine** are used, it refers to **java**.



Write (Compile) Once and Run Anywhere (WORA)

This terminology was given by Sun Microsystem for their programming language - Java. According to this concept, the same code must run on any machine and hence the source code needs to be portable. So Java allows run Java bytecode on any machine irrespective of the machine or the hardware, using JVM (Java Virtual Machine). The bytecode generated by the compiler is not platform-specific and hence takes help of JVM to run on a wide range of machines. So we can call Java programs as a write once and run on any machine residing anywhere.

Java Buzz Words

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. If you already understand the basic concepts of object-oriented programming like C++, learning Java will be even easier.

Object Oriented:

In Java, everything is an Object. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

Platform Independent:

Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.

Secure: With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Architecture-neutral: Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

Portable: Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.

Robust: Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and de-allocation for you. Java helps in this area by providing object-oriented exception handling.

Multithreaded: With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.

Interpreted: Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.

High Performance: With the use of Just-In-Time compilers, Java enables high performance.

Distributed: Java is designed for the distributed environment of the internet. Java also supports Remote Method Invocation (RMI). It handles TCP/IP protocols.

Dynamic: Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java

James Gosling initiated Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called 'Oak' after an oak tree that stood outside Gosling's office, also went by the name 'Green' and ended up later being renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.
On 13 November, 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).
On 8 May, 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Object Oriented Paradigm

Abstraction

An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction.
From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units.
For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions.

Three OO Concepts:

- Encapsulation
- Inheritance
- Polymorphism

Encapsulation

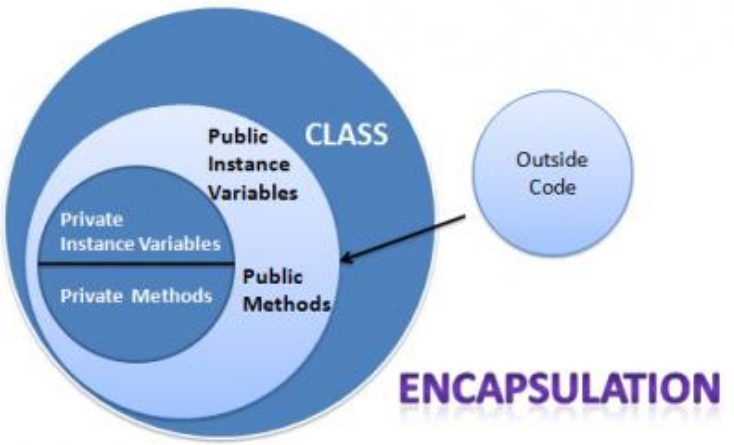
Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. It acts as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.

A class defines the structure and behaviour (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behaviour defined by the class. For this reason, objects are sometimes referred to as instances of a class.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called members of the class. Specifically, the data defined by the class are referred to as **member variables or instance variables**. The code that operates on that data is referred to as **member methods or just methods.**

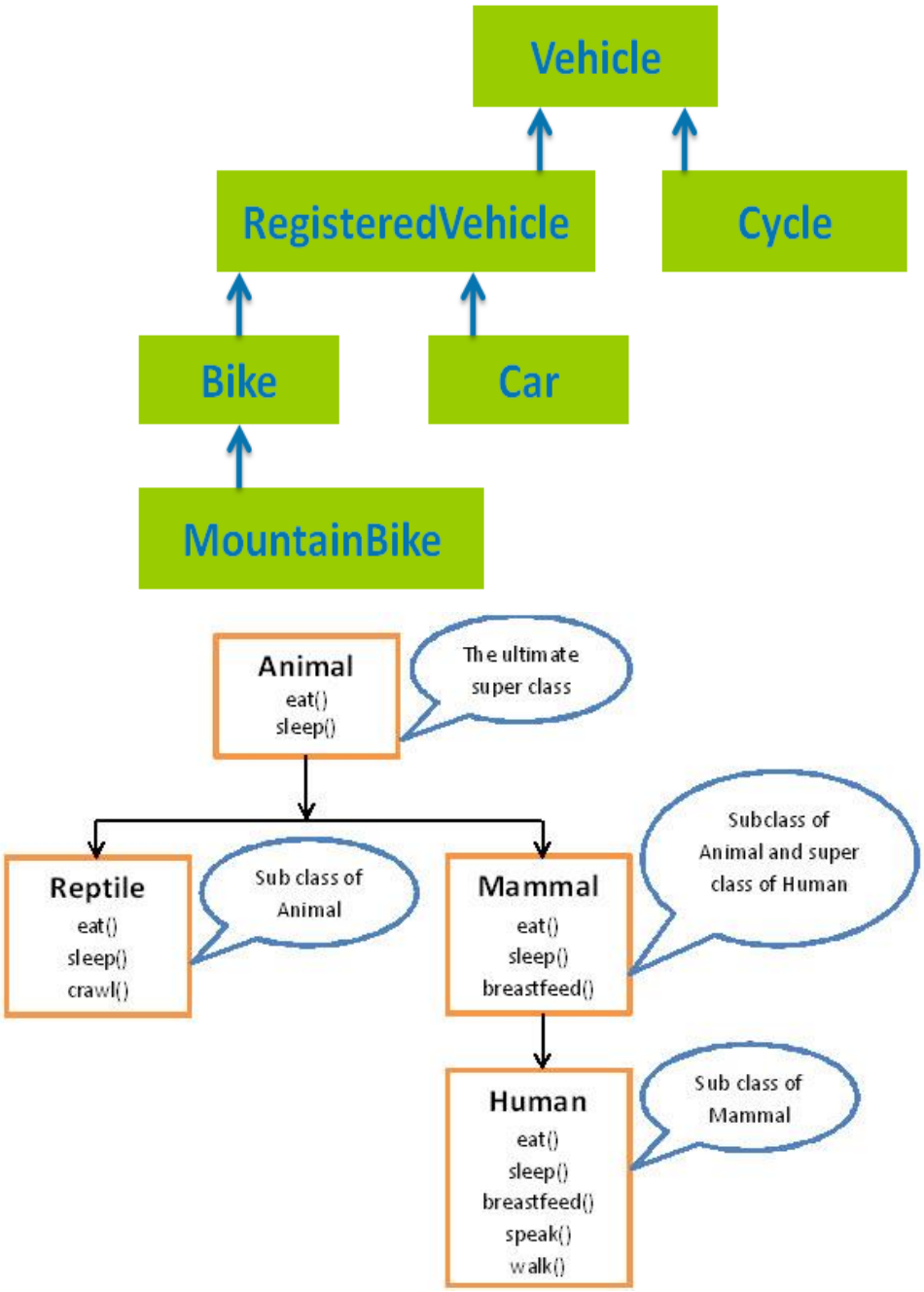
Each method or variable in a class may be marked private or public.

- The public interface of a class represents everything that external users of the class need to know, or may know.
- The private methods and data can only be accessed by code that is a member of the class.



Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification.
The class which inherits the properties of other is known as **subclass** (derived class, child class) and the class whose properties are inherited is known as **superclass** (base class, parent class).



Since mammals are simply more precisely specified animals, they inherit all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the class hierarchy.

Polymorphism

Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. **Polymorphism in java** is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word

"poly" means many and "morphs" means forms. So polymorphism means many forms.

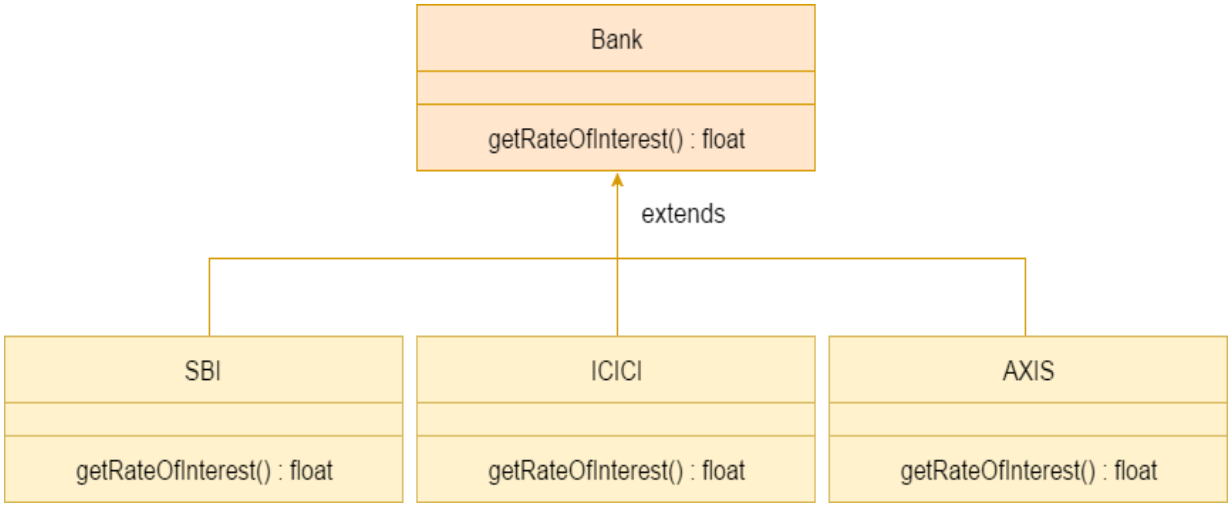
There are two types of polymorphism in java:

- compile time polymorphism and
- Runtime polymorphism.

We can perform polymorphism in java by **method overloading and method overriding.**

Example: Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs.

```
public class Dog extends Animal {

    public void makeSound() {
        System.out.println("Woof!");
    }

    public void makeSound(boolean injured) {
        System.out.println("Whimper");
    }
}
```



Polymorphism, Encapsulation, and Inheritance Work Together

When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scalable programs than does the process-oriented model. A well-designed hierarchy of classes is the basis for reusing the code in which you

have invested time and effort developing and testing. Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of your classes. Polymorphism allows you to create clean, sensible, readable, and resilient code.

Java Basics

When we consider a Java program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods, and instance variables mean.

- **Object -** Objects have states and behaviours. Example: A dog has states - colour, name, breed as well as behaviour such as wagging their tail, barking, eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/blueprint that describes the behaviour/state that the object of its type supports.
- **Methods -** A method is basically behaviour. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables -** Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

First Java Program

Let us look at a simple code that will print the words ***Hello World***.

```
public class MyFirstJavaProgram {
/* This is my first java program.
This will print 'Hello World' as the output */


  public static void main(String []args) {
     System.out.println("Hello World"); // prints Hello World
}
}
```

```
C:\> javac MyFirstJavaProgram.java
C:\> java MyFirstJavaProgram
Hello World
```

Java supports three styles of comments.

1) The one shown at the top of the program is called a multiline comment. This type of comment must begin with /* and end with */.
2) Single line comment . //
3) Documentation Comment

public static void main(String args[]) {

All Java applications begin execution by calling main( ).

When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared.

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.

System.out.println("This is a simple Java program.");

This line outputs the string "This is a simple Java program." followed by a new line on the

screen. In this case, 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.

DATATYPES

There are eight primitive datatypes supported by Java.

Data Type	**Range**	**Default size**
Boolean	False/ true	1 bit
char	0-65536	2 byte
byte	–128 to 127	1 byte
short	–32,768 to 32,767	2 byte
int	–2,147,483,648 to 2,147,483,647	4 byte
long	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8 byte
float	1.4e–045 to 3.4e+038	4 byte
double	4.9e–324 to 1.8e+308	8 byte

Integers: Java does not support unsigned, positive-only integers.

Char: Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits.

```
public class JavaCharExample {

    public static void main(String[] args) {
        char ch1 = 'a';
        char ch2 = 65; /* ASCII code of 'A'*/

        System.out.println("Value of char variable ch1 is :" +
ch1);
        System.out.println("Value of char variable ch2 is :" +
ch2);
    }
}
```

Output would be
Value of char variable ch1 is :a
Value of char variable ch2 is :A

- **Byte** are useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.

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

byte b1 = 100;
byte b2 = 20;

System.out.println("Value of byte variable b1 is :" + b1);
System.out.println("Value of byte variable b1 is :" + b2);
}
}
```

Output would be
Value of byte variable b1 is :100
Value of byte variable b1 is :20

- **Boolean takes either True or false**

```
public class JavaBooleanExample {

public static void main(String[] args) {

   boolean b1 = true;
   boolean b2 = false;
   boolean b3 = (10 > 2)? true:false;

System.out.println("Value of boolean variable b1 is :" + b1);
System.out.println("Value of boolean variable b2 is :" + b2);
System.out.println("Value of boolean variable b3 is :" + b3);
}
}
```

Output would be
Value of boolean variable b1 is :true
Value of boolean variable b2 is :false
Value of boolean variable b3 is :true

- **Floating**-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision.

```
import java.util.*;

public class JavaFloatExample {

      public static void main(String[] args) {
            float f = 10.4f;
            System.out.println("Value of float variable f is :" + f);
      }
}
```

Output would be
Value of float variable f is :10.4

- **Double** precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values.

```
public class JavaDoubleExample {

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

```
                double d = 1232.44;
                System.out.println("Value of double variable d is :" +
        d);
            }
        }
```

Output would be
Value of double variable f is :1232.44

Java Literals

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation. Literals can be assigned to any primitive type variable.

For example:

```
byte a = 68;
char a = 'A'
```

byte, int, long, and short can be expressed in decimal(base 10), hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicate octal, and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal = 100;
int octal = 0144;
int hexa = 0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001';
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

Notation	Character represented
\n	Newline (0x0a)
\r	Carriage return (0x0d)
\f	Form feed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

VARIABLES

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory;

Declaring a Variable

data type variable [= value][, variable [= value] ...] ;

Following are valid examples of variable declaration and initialization in Java:

```
int a, b, c;                    // Declares three ints, a, b, and c.
int a = 10, b = 10;  // Example of initialization
byte B = 22;                    // initializes a byte type variable B.
double pi = 3.14159;        // declares and assigns a value of PI.
char a = 'a';  // the char variable a is initialized with value 'a'
```

There are three kinds of variables in Java:

- Local variables
- Instance variables
- Class/Static variables

Local Variables

- Local variables are declared in methods, constructors, or blocks.
- Local variables are implemented at stack level internally.

- Access modifiers cannot be used for local variables.

Instance Variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Access modifiers can be given for instance variables.

Class/static Variables

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.

Dynamic Initialization

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.
Ex:

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

Here, three local variables—a, b, and c—are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse

The Scope and Lifetime of Variables

Java allows variables to be declared within any block. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your 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.

As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification.

To understand the effect of nested scopes, consider the following program:

```
class Scope {
public static void main(String args[]) {
int x; // known to all code within main
x = 10;
if(x == 10) { // start new scope
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);
}
}
```

Output

```
x and y: 10 20
x is 22
```

Type Conversion and Casting

- **Widening Casting(Implicit**)

- **Narrowing Casting(Explicitly done)**

1) **Java's Automatic Conversions (Implicit Conversion)**

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. Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Ex:

```
//64 bit long integer

long l;

//32 bit long integer

int i;

l=i;
```

2) **Casting Incompatible Types**

Although the automatic type conversions are helpful, they will not fulfill all needs. To create a ***narrowing*** conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion.

It has this general form: **(target-type) value**

```
public class Test
{
   public static void main(String[] args)
   {
     double d = 100.04;
```

```
    long l = (long)d;  //explicit type casting required
    int i = (int)l;     //explicit type casting required

    System.out.println("Double value "+d);
    System.out.println("Long value "+l);
    System.out.println("Int value "+i);

  }

}
```

Output :
Double value 100.04
Long value 100
Int value 100

Automatic Type Promotion in Expressions

```
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 sub expression a*b is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, 50 * 40, is legal even though a and b are both specified as type byte.

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

```
byte b = 50;
b = (byte)(b * 2);  //which yields the correct value of 100.
```

```
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);

}
}
```

Output: will be of double data type

Type Promotion Rules

1.All byte, short and char values are promoted to int.
2.If one operand is a long, the whole expression is promoted to long.
3.If one operand is a float, the entire expression is promoted to float.
4.If any of the operands is double, the result is double.

ARRAYS

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index.

One-Dimensional Arrays

A one-dimensional array is essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one-dimensional
array declaration is:

datatype *identifier* [ ];

Or

datatype[ ] *identifier*;

Ex: int month_days[];

It declares an array variable but do not allocate any memory. New is a special operator that allocates memory.

array-var = new type[size]; // (new will automatically be initialized to zero)

OR

dataType[] arrayVar = new dataType[arraySize];

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

Arrays can be initialized when they are declared. There is no need to use new.

```
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

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays.

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

Ex:

```
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++) {
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
```

Ex:

```
String[][] sampleData = {    {"a", "b", "c", "d"},  {"e", "f", "g", "h"}, {"i", "j", "k", "l"},
      {"m", "n", "o", "p"} };
```

OPERATORS IN JAVA

Java provides a rich operator environment. Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

Arithmetic Operators
Relational Operators
Bitwise Operators
Logical Operators
Ternary Operator and
Assignment Operator.

Operators	Precedence
Postfix	*expr*++ *expr*--
Unary	++*expr* --*expr* +*expr* -*expr* ~ !
Multiplicative	* / %
Additive	+ -
Shift	<< >> >>>
Relational	< > <= >= instance of
Equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
Ternary	? :
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Arithmetic Operators

The basic arithmetic operations—addition, subtraction, multiplication, and division—all behave as you would expect for all numeric types.

- The unary minus operator negates its single operand.
- The unary plus operator simply returns the value of its operand.

```
class OperatorExample{
public static void main(String args[]){      int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

Output

```
15
5
50
2
0
```

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 %:

```
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);
}
}
```

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

Note: **RESULT OF ARITHMETIC EXPRESSIONIS ALWAYS**
MAX(int, type of a and type of b)

Ex: byte a=10;
byte b= 20;
c= a+b; // the result will be integer type

Arithmetic Compound Assignment Operators [Shorthand assignment]

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

***var op= expression*;**

```
class OperatorExample{
  public static void main(String args[]){
  int a=10;
  int b=20;
  a+=4;//a=a+4 (a=10+4)
  b-=4;//b=b-4 (b=20-4)
  System.out.println(a);
  System.out.println(b);
  }}
```

The Relational Operators

The *relational operators* determine the relationship that one operand has to the other. The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

operator	Description
==	Check if two operands are equal
!=	Check if two operands are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

Boolean Logical Operators

The Boolean logical operators shown here operate only on **boolean** operands and relational expressions. All of the binary logical operators combine two **boolean** values to form a resultant **boolean** value. The logical Boolean operators, **&**, |, and ^, operate on **boolean** values in the same way that they operate on the bits of an integer.

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

Short-Circuit Logical Operators

Java provides two interesting Boolean operators not found in some other computer languages. 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.
Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly.

For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if ( denom != 0 && num / denom >10)
```

Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when denom is zero. If this line of code was written using the single "&" version of AND, both sides would have to be evaluated, causing a run-time exception when denom is zero.

```
class ShortCircuitAnd
{
   public static void main(String arg[])
   {
      int c = 0, d = 100, e = 50; // LINE A
      if( c == 1 && e++ < 100 )
      {
         d = 150;
      }
      System.out.println("e = " + e );
      }
}
```

OUTPUT
e = 50

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types: **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

operator	description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift
>>>	Right Shift zero fill
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise X-OR assignment
>>>=	Shift right zero fill assignment
>>=	Shift right assignment
<<=	Shift left assignment

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

Java AND Operator: Logical && vs Bitwise &

```
class OperatorExample{
   public static void main(String args[]){
   int a=10;
   int b=5;
   int c=20;
```

```
   System.out.println(a<b && a++<c);       //false && true = false
   System.out.println(a);                  //10 because second condition
is not checked
   System.out.println(a<b & a++<c);        //false && true = false
   System.out.println(a);                  //11 because second condition is
checked
   }}
```

Output:

```
false
10
false
11
```

Java OR Operator: Logical || and Bitwise |

The logical || operator doesn't check second condition if first condition is true. It checks second condition only if first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
class OperatorExample{
   public static void main(String args[]){
   int a=10;
   int b=5;
   int c=20;
   System.out.println(a>b||a<c);           //true || true = true

   System.out.println(a>b|a<c);            //true | true = true

   System.out.println(a>b||a++<c);         //true || true = true
   System.out.println(a);                  //10 because second condition is not
checked
   System.out.println(a>b|a++<c);          //true | true = true
   System.out.println(a);                  //11 because second condition is
checked
   }}
```

Output:

```
true
true
true
10
true
11
```

Left Shift & Right Shift

```
   class OperatorExample
{
   public static void main(String args[]){
   System.out.println(10<<2);       //10*2^2=10*4=40
   System.out.println(10<<3);       //10*2^3=10*8=80
   System.out.println(10>>2);      //10/2^2=10/4=2
   System.out.println(20>>2);     //20/2^2=20/4=5
   System.out.println(20>>3);    //20/2^3=20/8=2
   }
}
```

Java Shift Operator Example: >> vs >>>

>>> is also known as Unsigned Right Shift. For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an *unsigned shift*.

int x = 13 >>> 1;
Out put : 6

// 13 = 00000000000000000000000000001101
// 6 = 00000000000000000000000000000110

y = -8 >>> 2;
Output : 1073741822 // -8 = 11111111111111111111111111111000

// 1073741822 =
00111111111111111111111111111110

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 to further illustrate what is happening:
11111111 11111111 11111111 11111111 –1 in binary as an int
>>>24
00000000 00000000 00000000 11111111 255 in binary as an int

```
class OperatorExample{
public static void main(String args[]){
   //For positive number, >> and >>> works same
   System.out.println(20>>2);
   System.out.println(20>>>2);
   //For nagative number, >>> changes parity bit (MSB) to 0
   System.out.println(-20>>2);
   System.out.println(-20>>>2);
}
}
```

Output

```
5
5
-5
1073741819
```

Java Ternary Operator

Java includes a special ternary (three-way) operator that can replace certain types of if-then else statements. This operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered.

The ? has this general form:

expression1 ? expression2 : expression3

Here, expression1 can be any expression that 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 OperatorExample{
  public static void main(String args[]){
  int a=2;
  int b=5;
  int min=(a<b)?a:b;
  System.out.println(min);
  }}
```

Output:

2

Increment and Decrement

The ++ and the – – are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one.

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

Output

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

CONTROL STATEMENTS

If- else:

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the 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.

```
public class IfExample {
public static void main(String[] args) {
   int age=20;
   if(age>18)
     {
      System.out.print("Eligible to vote");
     }
}
}
```

Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that 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.

Syntax :

```
if (condition)
{
        if (condition){
                //Do something
        }
//Do something
}
```

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

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-elseif
ladder. It looks like this:

Syntax: if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
.
.
.
else
statement;

```
public class ControlFlowDemo
{
   public static void main(String[] args)
   {
      char ch = 'o';

      if (ch == 'a' || ch == 'A')
         System.out.println(ch + " is vowel.");
      else if (ch == 'e' || ch == 'E')
         System.out.println(ch + " is vowel.");
      else if (ch == 'i' || ch == 'I')
         System.out.println(ch + " is vowel.");
      else if (ch == 'o' || ch == 'O')
         System.out.println(ch + " is vowel.");
      else if (ch == 'u' || ch == 'U')
         System.out.println(ch + " is vowel.");
      else
         System.out.println(ch + " is a consonant.");
          }
}
```

Switch

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

Syntax: switch (*expression*)

```
{
        case value1:
        // statement sequence
        break;
        case value2:
        // statement sequence
        break;
        .
        .
        .
        case valueN :
        // statement sequence
        break;
        default:
        // default statement sequence
}
```

expression must be of type byte, short, int, char, or enumerated data type(String).

```
class StringSwitch {
public static void main(String args[]) {
String str = "two";
 switch(str)
{
 case "one":
                System.out.println("one");
                break;
case "two":
                System.out.println("two");
                break;
case "three":
                 System.out.println("three");
                 break;
default:
               System.out.println("no match");
               break;
}}}
```

Output : two

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

```
    int number=20;
    switch(number){
    case 10: System.out.println("10");break;
    case 20: System.out.println("20");break;
    case 30: System.out.println("30");break;
    default:System.out.println("Not in 10, 20 or 30");
    }
  }
  }
```

Output : 20

Nested switch Statements

You can use a switch as part of the statement sequence 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.

For example, the following fragment is perfectly valid:

```
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: // .....so on.
```

Iteration Statements

Java's iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

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.

Example:

```
class WhileLoopExample{
        public static void main(String[] args){
                int num=0;
                while(num<=5){
                        System.out.println(""+num);
                        num++;
                }
        }
}
```

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. As with all of Java's loops, condition must be a Boolean expression.

The do-while loop is especially useful when you process a menu selection, because you will
usually want the body of a menu loop to execute at least once.

```
class Menu {
public static void main(String args[])
{
char choice;
```

```
do
{
   System.out.println("Help on: ");
    System.out.println(" 1. if");
    System.out.println(" 2. switch");
    System.out.println(" 3. while");
     System.out.println(" 4. do-while");
    System.out.println(" 5. for\n");
    System.out.println("Choose one:");
choice = (char) System.in.read();
} while( choice < '1' || choice > '5');

System.out.println("\n");

switch(choice) {
case '1':
System.out.println("The if:\n");
System.out.println("if(condition) statement;");
System.out.println("else statement;");
break;
case '2':
System.out.println("The switch:\n");
System.out.println("switch(expression) {");
System.out.println(" case constant:");
System.out.println(" statement sequence");
System.out.println(" break;");
System.out.println(" //...");
System.out.println("}");
break;
case '3':
System.out.println("The while:\n");
System.out.println("while(condition) statement;");
break;
case '4':
System.out.println("The do-while:\n");
System.out.println("do {");
System.out.println(" statement;");
System.out.println("} while (condition);");
break;
case '5':
System.out.println("The for:\n");
System.out.print("for(init; condition; iteration)");
System.out.println(" statement;");
break;
```

```
}
}
}
```

```
public class DoWhileExample {
public static void main(String[] args) {
   int i=1;
   do{
      System.out.println(i);
   i++;
   }while(i<=10);
}
}
```

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 newer "for-each" form.

1) Here is the general form of the traditional for statement:

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

Ex :

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++)
        sum += nums[i];
```

2) **For-Each Version of the for Loop:**

The general form of the for-each version of the for is shown here:

for(type itr-var : collection) statement-block

Here, type specifies the type and itr-var specifies the name of an iteration variable that will receive the elements from a collection(array), one at a time, from beginning to end. The collection being cycled through is specified by collection.

Ex :

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums)
sum += x;
```

```
public class Test {
  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ) {
      System.out.print( x );
      System.out.print(",");
    }
    System.out.print("\n");
    String [] names = {"James", "Larry", "Tom", "Lacy"};

    for( String name : names ) {
      System.out.print( name );
      System.out.print(",");
    }
  }
}
```

```
10, 20, 30, 40, 50,
James, Larry, Tom, Lacy,
```

Iterating Over Multidimensional Arrays

```
class sample {
public static void main(String args[]) {
int sum = 0;
int nums[][] = new int[3][5];
for(int i = 0; i < 3; i++)
for(int j = 0; j < 5; j++)
      nums[i][j] = (i+1)*(j+1);

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

Output:

Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5

```
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
```

Jump Statements

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.

1) Using break to Exit a Loop
2) Using break as a Form of Goto:
   Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain.

```
public class BreakDemo
{
  public static void main(String[] args)
  {
    for (int i = 1; i <= 10; i++)
    {
      if (i == 5)
      {
        break;          // terminate loop if i is 5
      }
      System.out.print(i + " ");
    }
    System.out.println("Thank you.");
  }
}
```

Output 1 2 3 4 Thank you

Using continue

In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to

the conditional expression. For all three loops, any intermediate code is bypassed.

```
public class ContinueDemo
{
  public static void main(String[] args)
  {
    for (int i = 1; i <= 10; i++)
    {
      if (i % 2 == 0)
      {
        continue;    // skip next statement if i is even
      }
      System.out.println(i + " ");
    }
  }
}
```

1 3 5 7 9

Break	**Continue**
The break statement results in the termination of the loop, it will come out of the loop and stops further iterations.	The continue statement stops the current execution of the iteration and proceeds to the next iteration
The break statement has two forms: labelled and unlabelled. An unlabelled break statement terminates the innermost switch, for, while, or do-while statement, but a labelled break terminates an outer statement.	The continue statement skips the current iteration of a for, while , or do-while loop. The unlabelled form skips to the end of the innermost loop's body and evaluates the Boolean expression that controls the loop. A labelled continue statement skips the current iteration of an outer loop marked with the given label.
The general form of the labelled break statement is shown here: **break label;**	The general form of the labelled continue statement is shown here: **continue label;**

class Break {
public static void main(String args[]) {
boolean t = true;
first: { **second:** {
third: {
System.out.println("Before the break.");
	class ContinueLabel {
	public static void main(String args[]) {
	outer: for (int i=0; i<4; i++)
	{
	for(int j=0; j<4; j++)
	{
	if(j > i)

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.");}}}
	{
	System.out.println();
	continue outer;
	}
	System.out.print(" " + (i * j));
	}
	}
	System.out.println();
	}
	}

Output: Before the break. This is after second block.	0 0 1 0 2 4 0 3 6 9

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. As such, it is categorized as a jump statement.

At any time in a method, the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.

```
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.");
}
}
```

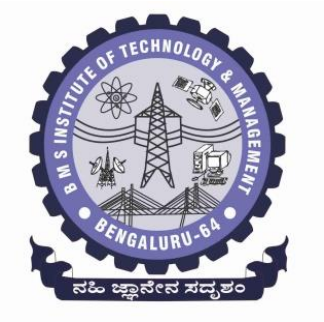
The output from this program is shown here:

Before the return.

Here, return causes execution to return to the Java run-time system, since it is the run-time system that call main( ):

B M S INSTITUTE OF TECHNOLOGY & MANAGEMENT

YELAHANKA, BENGALURU – 560064.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE -3 NOTES OF

OBJECT ORIENTED CONCEPTS -18CS45

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2018 -2019)

SEMESTER – IV

Prepared by,

Mr. Muneshwara M S

Asst. Prof, Dept. of CSE

VISION AND MISSION OF THE CS&E DEPARTMENT

Vision

To develop technical professionals acquainted with recent trends and technologies of computer science to serve as valuable resource for the nation/society.

Mission:

Facilitating and exposing the students to various learning opportunities through dedicated academic teaching, guidance and monitoring.

VISION AND MISSION OF THE INSTITUTE

Vision

To emerge as one of the finest technical institutions of higher learning, to develop engineering professionals who are technically competent, ethical and environment friendly for betterment of the society.

Mission

Accomplish stimulating learning environment through high quality academic instruction, innovation and industry-institute interface

MODULE 3. CLASSES, INHERITANCE,EXCEPTION HANDLING

The following concepts should be learn in this Module

Classes: Classes fundamentals; Declaring objects; Constructors, this keyword, garbage collection. **Inheritance:** inheritance basics,using super, creating multi level hierarchy, method overriding. **Exception handling:**Exception handling in Java.

Text book 2: Ch:6 Ch: 8 Ch:10 , RBT: L1, L2, L3

CLASS FUNDAMENTALS:

A class is declared by using class keyword. class is a template for an object, and an object is an instance of a class.

Syntax :

```
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}}
```

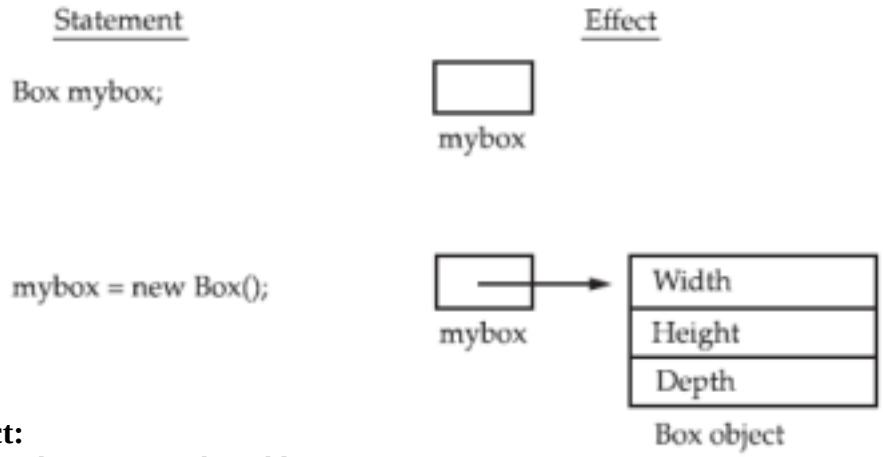
The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.

Declaring Objects/ Instantiating a Class:

Creating objects of a class is a two-step process.

- First, you must declare a variable of the class type which is simply a variable that can refer to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable using the new operator. The new operator dynamically allocates memory for an object and returns a reference to it where the address is stored.

```
Box mybox = new Box();

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



Object:

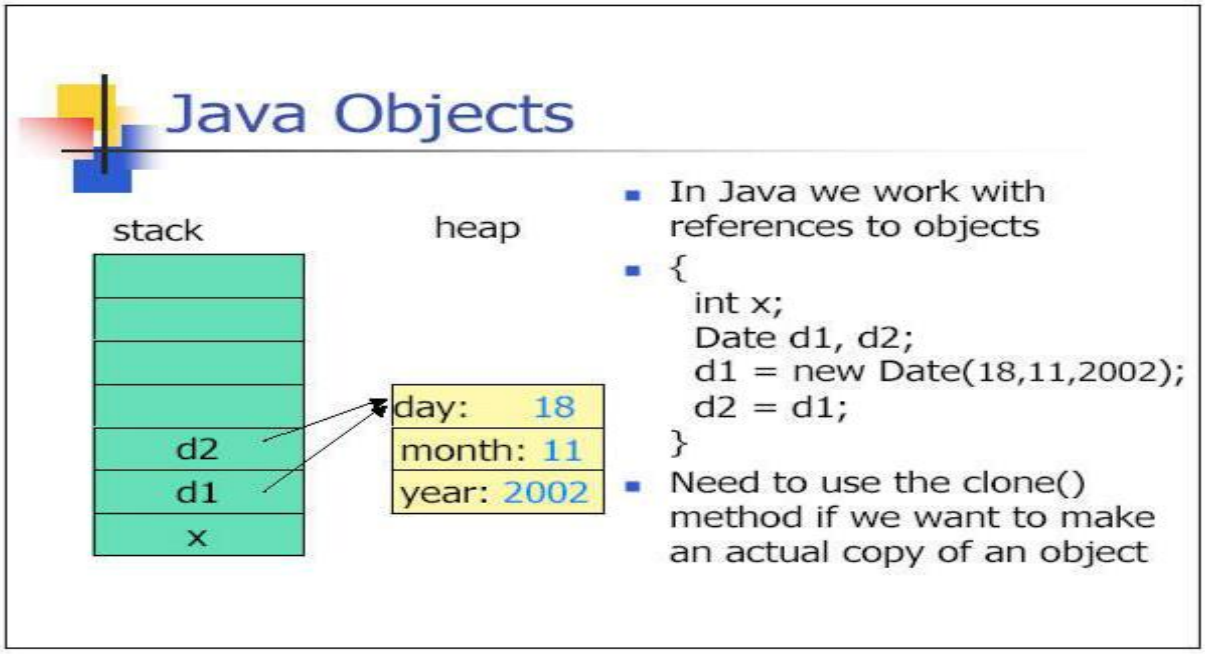
- Object is *a real world entity*.
- Object is *a run time entity*.
- Object is *an entity which has state and behavior*.
- Object is *an instance of a class*.

Java Heap Space

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it's always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that doesn't have any reference.

Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order



A Simple Class

```
class Box {
double width;
double height;
double depth;
}
                    // This class declares an object of type Box.
Class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;

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

vol = mybox.width * mybox.height *
mybox.depth; System.out.println("Volume is " +
vol); }
}
```

Introducing Methods

This is the general form of a method:

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

return value;
}

Methods define the interface to most classes. This allows the class implementer to hide the specific layout of internal data structures behind cleaner method abstractions. Defining methods provide access to data, you can also define methods that are used internally by the class itself.

```
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 Demo {
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);

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

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

CONSTRUCTORS:

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.A constructor doesn't have a return type.The name of the constructor must be the same as the name of the class.Unlike methods, constructors are not considered members of a class.

A constructor is called automatically when a new instance of an object is created.

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Default Constructor: It is a constructor which do not take any arguments.If you do not define any constructor in your class, java generates one for you by default.

```
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 demo
{
public static void main(String args[]) {
Box mybox1 = new Box();

double vol;

vol = mybox1.volume();
System.out.println("Volume is " + vol);
}
}
```

Parameterized constructor

A constructor that have parameters is known as parameterized constructor.

```
Class Student
{
   int id;
   String name;

   Student(int I,String n)
  {
   id = I;
   name = n;
   }

   void display()
        {
         System.out.println(id+" "+name);
        }
}

Class test{
   public static void main(String args[])
   {
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   s1.display();
   s2.display();
  }
}
```

Objects as Parameters

Using Objects as Parameters: So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking
object boolean equalTo(Test o) {
if(o.a == a && o.b == b) return
true; else return false;
}
}
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22); Test ob3 = new Test(-1, -
1); System.out.println("ob1 == ob2: " +
ob1.equalTo(ob2)); System.out.println("ob1 == ob3: " +
ob1.equalTo(ob3));
}
}
```

Exercise 1:
Write a java program to create 2 objects of complex numbers and pass these objects as parameters to the methods. Perform addition of 2 complex numbers and return the sum as an object.

The this Keyword

Java defines the **this** keyword. **It** can be used inside any method to refer to the *current o*bject.

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

- **this** keyword is used to refer to current object.
- **this** is always a reference to the object on which method was invoked.
- **this** can be used to invoke current class constructor.
- **this** can be passed as an argument to another method.

Instance Variable Hiding:

Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.

```
class Student
{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee)
{
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display()
{
System.out.println(rollno+" "+name+" "+fee);
}
}

class Test
{
public static void main(String args[])
{
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}
}
```

Overloaded Constructors

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

```
class Student{
   int id;
   String name;
   int age;
   Student (int i,String n)
  {
        id = i;
       name = n;
   }
   Student (int i,String n,int a)
```

```
   {
       id = i;
       name = n;
       age=a;
    }
   void display()
{
      System.out.println(id+" "+name+" "+age);
}

   public static void main(String args[])
{
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
  }
}
```

Role of this () in constructor overloading

/*this() is used for calling the default constructor from parameterized constructor. It should always be the first statement in constructor body. */

```
public class student
{
  private int rollNum;
  student()
  {
    rollNum =100;
  }
  student(int rnum)
  {
    this();
   rollNum = rollNum+ rnum;
  }
  public int getRollNum() {
          return rollNum;
  }
  public void setRollNum(int rollNum) {
          this.rollNum = rollNum;
  }
}
class TestDemo{
  public static void main(String args[])
  {
     student obj = new student(12);
     System.out.println(obj.getRollNum());
   }
}
```

GARBAGE COLLECTION:

In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: 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.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

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

gc() method:

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){ TestGarbage1 s1=new
 TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

```
object is garbage collected
      object is garbage collected
```

A Stack Class:

```
class Stack
{
int stck[] = new int[10];
int top;
// Initialize top-of-
stack Stack()
 {
top = -1;
```

```
}

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

int pop()
{
if(top < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[top--];
}
}

class TestStack
{
public static void main(String args[])
 {
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();

// push some numbers onto the
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());
}
}
```

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
```

Overloading Methods

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. **Method overloading** is also known as **Static Polymorphism**.

Argument lists could differ in –

1. Number of parameters.
2. Data type of parameters.
3. Sequence of Data type of parameters.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

Advantage of method overloading

1) Method overloading *increases the readability of the program*.

```
class Calculate
{
 void sum (int a, int b)
 {
  System.out.println("sum is"+(a+b)) ;
 }
 void sum (float a, float b)
 {
  System.out.println("sum is"+(a+b));
 }
 Public static void main (String[] args)
 {
  Calculate cal = new Calculate();
```

```
  cal.sum (8,5);      //sum(int a, int b) is method is called.
```

```
  cal.sum (4.6f, 3.8f); //sum(float a, float b) is called.
 }
}
```

```
Sum is 13
Sum is 8.4
```

```
class Overloading3
{
   public void disp(char c, int num)
   {
      System.out.println("c ");
      System.out.println("num ");

   }
   public void disp(int num, char c)
   {
      System.out.println("c ");
      System.out.println("num ");
   }
}
class Sample3
{
   public static void main(String args[])
   {
      Overloading3 obj = new Overloading3();
      obj.disp('x', 51 );
      obj.disp(52, 'y');
   }
}
```

Recursion:

Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

class Factorial {

```
int fact(int n) {
int result;
if(n==1)
 return 1;
result = fact(n-1) * n;
return result;
}
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
```

```
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

Advantages of Recursion

1. Reduces time complexity.
2. Performs better in solving problems based on tree structures.

Access Control

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe. It has the widest scope among all other modifiers.

```
//save by A.java

package pack;
public class A
{
public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B
{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

```
class A
{
private int data=40;
private void msg()
{
System.out.println("Hello java");}
}

public class Simple
. {
. public static void main(String args[])
{
  A obj=new A();
   System.out.println(obj.data);                    //Compile  Time Error
     obj.msg();                    //Compile Time Error
```

```
}
}
```

protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class. The protected access modifier cannot be applied to class and interfaces.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");} }
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Default:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc. A variable or method declared without any access control modifier is available to any other class in the same package.

The fields in an interface are implicitly public static final and the methods in an interface are by default public.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
```

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Static:

It is a keyword which is used to define the class members that will be used independent of any object of that class. Static members are initialized for the first time when class is loaded. The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.[i.e., without instantiating the class]

Static Methods
Methods declared as **static** have several restrictions:
• They can only directly call other **static** methods.
• They can only directly access **static** data.
• They cannot refer to **this** or **super** in any way.

Static Blocks:
Static blocks are also called *Static initialization blocks* . A static initialization block is a normal block of code enclosed in braces, { }, and preceded by the static keyword.

```
static {
   // whatever code is needed for initialization goes here
}


class UseStatic
 {
static int a = 3;
static int b;

static void display (int x)
{
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}

static
{
System.out.println("Static block initialized.");
b = a * 4;
}
```

```
public static void main(String args[])
 {
display (42);
}
}
Static block initialized.
x = 42
a = 3
b = 12


class StaticDemo
 {
       static int a = 42;
       static int b = 99;
 static void callme()
 {
   System.out.println("a = " + a);
  }
}
class StaticByName
{
public static void main(String args[]) {
```

```
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

```
a = 42
b = 99
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator.

Inheritance:

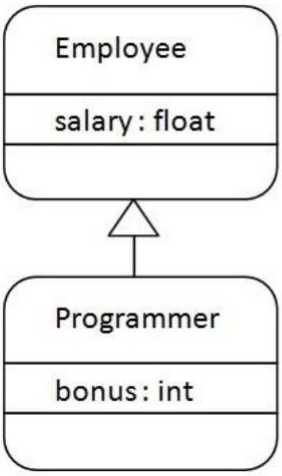
Inheritance in java is a mechanism in which one object acquires all the properties and behaviours of parent object. The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship. The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class). **extends** is the keyword used to inherit the properties of a class.

Use of inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved). ○
  For Code Reusability.

Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

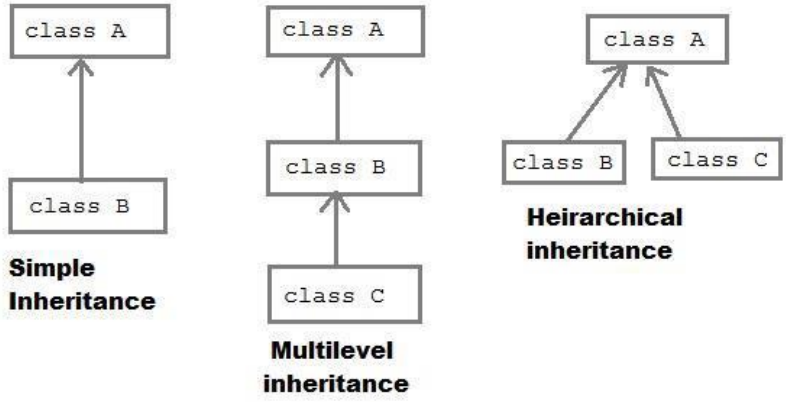



```
class Employee
{
 float salary=40000;
}

class Programmer extends Employee
{
 int bonus=10000;
 public static void main(String args[])
{
   Programmer p=new Programmer();
  System.out.println("Programmer salary is:"+p.salary);
  System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

```
Programmer salary is:40000.0
 Bonus of programmer is:10000
```

Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance



Single Level Inheritance :

One class extends one class only

```
class Animal
{
void eat()
{
   System.out.println("eating...");
}
}
class Dog extends Animal
{
. void bark()
   {
      System.out.println("barking...");
  }
}
class TestInheritance
{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}
}
```

```
barking...
eating...
```

Multilevel Inheritance:

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class.

```
class Animal
{
void eat(){
System.out.println("eating...");
}
}
class Dog extends Animal
{
void bark(){
System.out.println("barking...");
}
}
class BabyDog extends Dog{
void weep()
{
System.out.println("weeping...");
}
}
class TestInheritance2
{
public static void main(String args[])
{
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}
}
```

```
weeping...
barking...
 eating..
```

Hierarchical Inheritance :

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.** In **Hierarchical inheritance** one parent class will be inherited by **many** sub classes.

```
class Animal{
void eat(){System.out.println("eating...");}
}

class Dog extends Animal{
void bark(){System.out.println("barking...");}
}

class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}

class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//Compile Time.Error
}}
```

```
meowing...
eating...
```

Polymorphism:

Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms. There are two types of polymorphism in java:

- **compile time polymorphism and**
- **runtime polymorphism.**

Compile Time polymorphism can be achieved using overloading methods Run Time Polymorphism can be achieved using overriding methods

Runtime Polymorphism

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

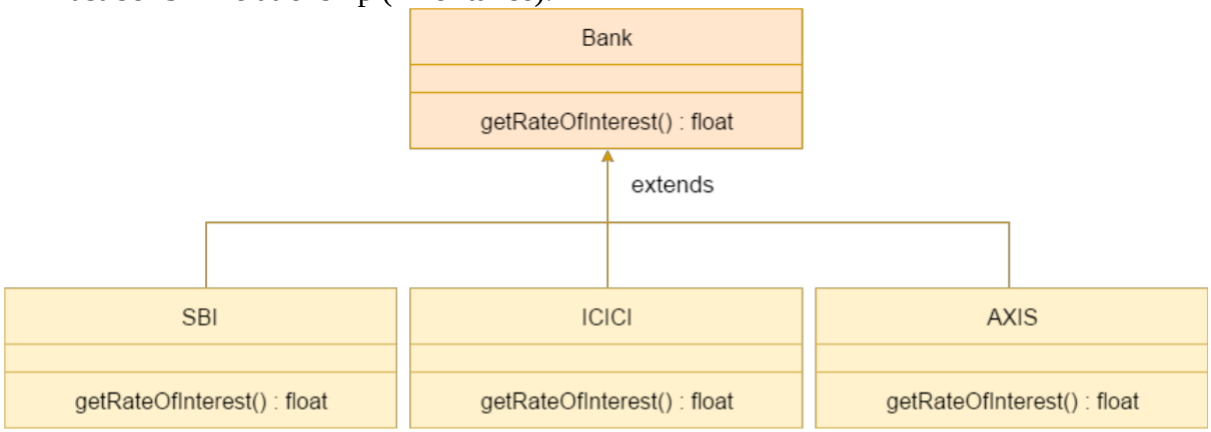
Child class has the same method as of base class. In such cases child class overrides the parent class method without even touching the source code of the base class.

Advantage of Java Method Overriding

- Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method Overriding is used for Runtime Polymorphism

Rules for Method Overriding

- method must have same name as in the parent class.
- method must have same parameter as in the parent class.
- must be IS-A relationship (inheritance).




```
class Bank{
float getRateOfInterest(){
return 0;
}
}
class SBI extends Bank{
float getRateOfInterest(){
return 8.4f;
}
}
class ICICI extends Bank{
float getRateOfInterest(){
return 7.3f;
}
}
class AXIS extends Bank{
float getRateOfInterest(){
return 9.7f;
}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
```

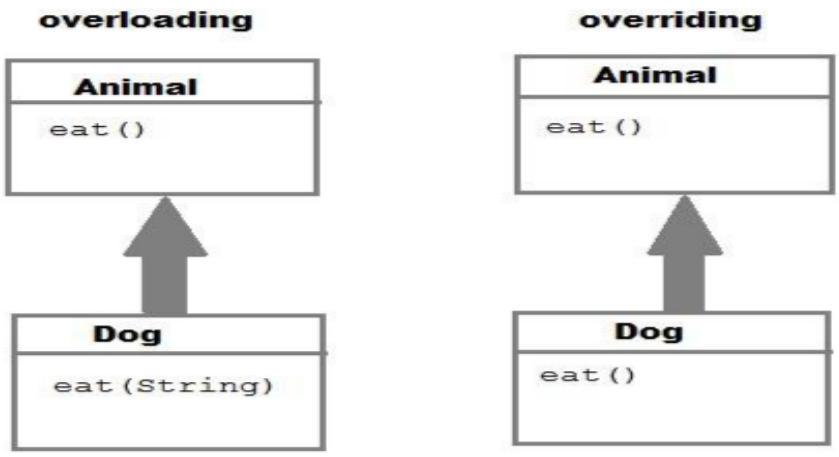
```
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}
}
```

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
```

AXIS Rate of Interest: 9.7

Difference between Overloading and Overriding

	Overloading	Overriding
1	Whenever same method or Constructor is existing multiple times within a class either with different number of parameter or with different type of parameter or with different order of parameter is known as Overloading.	Whenever same method name is existing multiple time in both base and derived class with same number of parameter or same type of parameter or same order of parameters is known as Overriding.
2	Arguments of method must be different at least arguments.	Argument of method must be same including order.
3	Method signature must be different.	Method signature must be same.
4	Private, static and final methods can be overloaded.	Private, static and final methods can not be override.
5	Access modifiers point of view no restriction.	Access modifiers point of view not reduced scope of Access modifiers but increased.
6	Also known as compile time polymorphism or static polymorphism or early binding.	Also known as run time polymorphism or dynamic polymorphism or late binding.
7	Overloading can be exhibited both are method and constructor level.	Overriding can be exhibited only at method label.
8	The scope of overloading is within the class.	The scope of Overriding is base class and derived class.
9	Overloading can be done at both static and non-static methods.	Overriding can be done only at non-static method.
10	For overloading methods return type may or may not be same.	For overriding method return type should be same.



NOTE : Static methods cannot be overridden because, a static method is bounded with class where as instance method is bounded with object.

Super Keyword:

The **super** keyword in java is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

This scenario occurs when a derived class and base class has same data members. Hence we use super keyword to refer a member of immediate parent class.

```
class Vehicle
{
   int maxSpeed = 120;
}


class Car extends Vehicle
{
   int maxSpeed = 180;

   void display()
   {
       System.out.println("Derived class Speed: " + maxSpeed);
      System.out.println("Base Speed: " + super.maxSpeed);
   }
}
```

```
/* Driver program to test */
class Test
{
   public static void main(String[] args)
   {
      Car small = new Car();
      small.display();
   }
}
```

```
Derived class Speed: 180
Base class Speed: 120
```

2) super can be used to invoke parent class method

The **super keyword** can also be used to invoke or call parent class method. It should be used in case of method overriding. In other word **super keyword** use when base class method name and derived class method name have same name.

```
class Student
{
void message()
{
System.out.println("Good Morning Sir");
}
}

class Faculty extends Student
{
void message()
{
System.out.println("Good Morning Students");
}

void display()
{
message();                    //will invoke or call current class message() method
super.message();              //will invoke or call parent class message() method
}

public static void main(String args[])
{
Student s=new Student();
s.display();
}
}
```

```
Good Morning Students
Good Morning Sir
```

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke or call the parent class constructor. Constructor are calling from bottom to top and executing from top to bottom.
To establish the connection between base class constructor and derived class constructors JVM provides two implicit methods they are: If super is not used explicitly compiler will automatically add super as the first statement.

- Super()
- Super(...)

Super():

Super() It is used for calling super class default constructor from the context of derived class constructors.

```
class Employee
{
Employee()
{
System.out.println("Employee class Constructor");
}
}
class HR extends Employee
{
HR()
{
super(); //will invoke or call parent class constructor
System.out.println("HR class Constructor"); }

}
class Supercons
{
public static void main(String[] args)
{
HR obj=new HR();
}
}
```

```
Employee class Constructor
HR class Constructor
```

Super(...)

Super(...) It is used for calling super class parameterize constructor from the context of derived class constructor.

```
class Person
{
int id;
String name;
Person(int id,String name)
{
```

```
this.id=id;
this.name=name;
}
}
class Emp extends Person
{
float salary;
Emp(int id,String name,float salary)
{
        super(id,name);                 //reusing parent constructor
        this.salary=salary;
}
void display()
{
System.out.println(id+" "+name+" "+salary);}
}

class TestSuper5
{
public static void main(String[] args)
{
Emp e1=new Emp(1,"abc",5000f);
e1.display();
}
}
1 abc 5000
```

Important rules

Rule for default constructor

Whenever the derived class constructor want to call default constructor of base class, in the context of derived class constructors we write super(). It is optional to write because every base class constructor contains single form of default constructor

Rule for Parameterized constructor

Whenever the derived class constructor wants to call parameterized constructor of base class in the context of derived class constructor we must write super(...). which is mandatory to write because a base class may contain multiple forms of parameterized constructors.

Abstract Classes

An *abstract class* is a class that is declared abstract—It can have abstract and non-abstract methods (method with body).Abstract classes cannot be instantiated, but they can be subclassed.

That is we cannot create an object for abstract classes

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Abstract method

Method that is declared without any body within an abstract class is called abstract method. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Syntax :

abstract return_type function_name ();

```
abstract class Shape
{
abstract void draw();
}

class Rectangle extends Shape{
void draw(){
System.out.println("drawing rectangle");}
}

class Circle extends Shape{
void draw(){
System.out.println("drawing circle");}
}

class TestAbstraction
{
public static void main(String args[])
{
Rectangle r = new Rectangle();
r.draw();
```

```
Shape s=new Circle();
s.draw();
}
}
```

```
drawing rectangle
drawing circle
```

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}
}
```

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract classes with constructors

```
abstract class Bike
{
   Bike(){
    System.out.println("bike is created");
}
   abstract void run();

   void changeGear(){
System.out.println("gear changed");}
 }

 class Honda extends Bike{
 void run(){
System.out.println("running safely..");
}
 }
 class TestAbstraction2{
 public static void main(String args[]){
```

```
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

```
bike is created
    running safely..
    gear changed
```

When to use Abstract Methods & Abstract Class?

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Abstract classes are used to define generic types of behaviours at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

EXCEPTION HANDLING IN JAVA

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Difference between error and exception

Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc.
Exceptions are conditions within the code. A developer can handle such conditions and take necessary corrective actions.

Few examples –

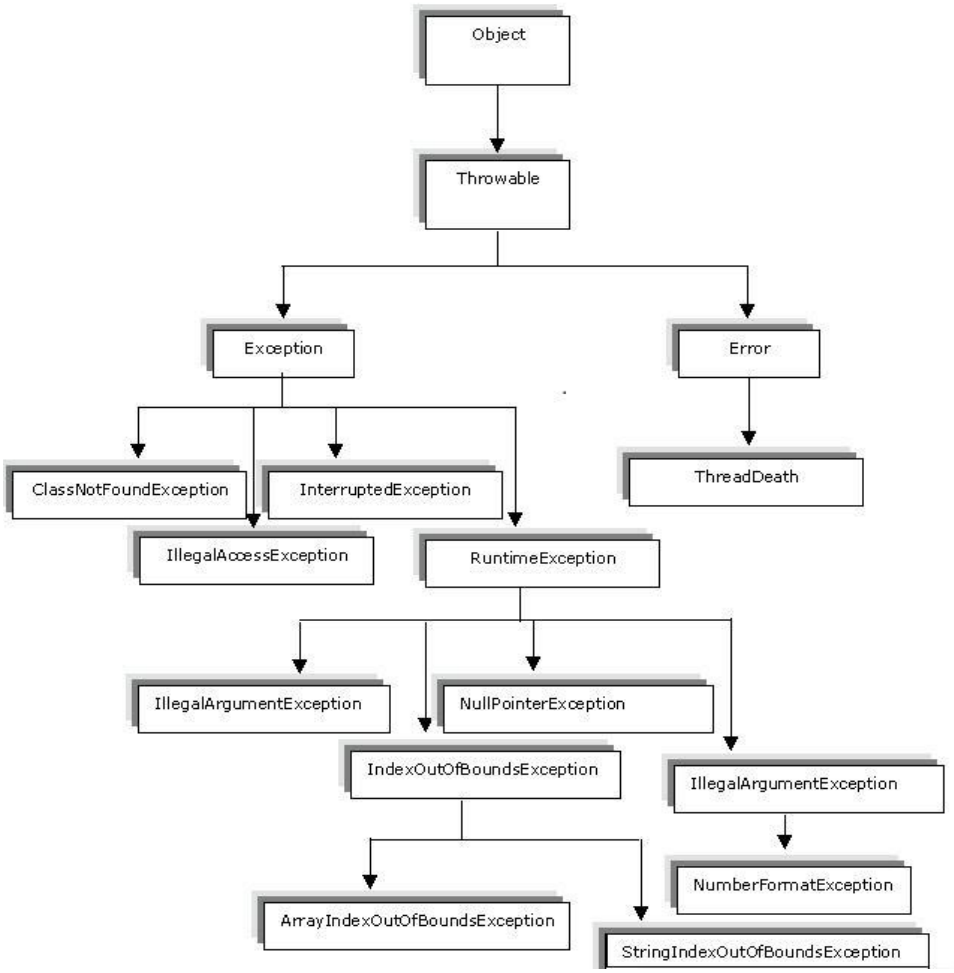
- DivideByZero exception
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException

Advantages of Exception Handling

- Exception handling allows us to control the normal flow of the program by using exception handling in program.

- It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.
- It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

Exception hierarchy



Java Exception Handling Keywords

Java provides specific keywords for exception handling purposes,

1. try
2. catch
3. finally
4. throw
5. throws

try-catch –

try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.

A catch block must be associated with a try block. The corresponding catch block executes if an exception of a particular type occurs within the try block.

```
class Excp
{
 public static void main(String args[])
 {
  int a,b,c;
  try
```

```
import java.io.*;

public class ExcepTest
{
   public static void main(String args[])
 {
     try {
        int a[] = new int[2];
        System.out.println("Access element three :" + a[3]);
     }
    catch(ArrayIndexOutOfBoundsException e)
     {
        System.out.println("Exception thrown :" + e);
     }
System.out.println("Out of the block");
   }
}
```

For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try catch in java

```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

```
  {
     a=0;
     b=10;
     c=b/a;
      System.out.println("This line will not be executed");
  }
  catch(ArithmeticException e)
  {
   System.out.println("Divided by zero");
  }
  System.out.println("After exception is handled");
 }
}
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks.
If the try block **throws an exception,** the appropriate catch block (if one exists) will catch it –catch(ArithmeticException e) is a catch block that can catch ArithmeticException – catch(NullPointerException e) is a catch block that can catch NullPointerException

All the statements in the catch block will be executed and then the program continues. If the exception type of exception, matches with the first catch block it gets caught, if not the exception is passed down to the next catch block.

```
class Example2{
   public static void main(String args[]){
     try{
         int a[]=new int[7];
         a[4]=30/0;
         System.out.println("First print statement in try block");
     }
     catch(ArithmeticException e){
        System.out.println("Warning: ArithmeticException");
     }
     catch(ArrayIndexOutOfBoundsException e){
        System.out.println("Warning: ArrayIndexOutOfBoundsException");
     }
     catch(Exception e){
        System.out.println("Warning: Some Other exception");
     }
   System.out.println("Out of try-catch block...");
  }
}
```

```
Warning: ArithmeticException
Out of try-catch block...
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.Java finally block is always executed whether exception is handled or not.

Finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurred or not

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e)
    {
          System.out.println(e);
     }
 finally
    {
      System.out.println("finally block is always executed");
     }
  System.out.println("rest of the code...");
}
}
```

Output:finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

throws – When we are throwing any exception in a method and not handling it, then we need to use **throws** keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to its caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with main() method also.

Syntax of java throws

```
return_type method_name() throws exception_class_name
{
//method code
}
```

throw –

We know that if any exception occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometime we might want to generate exception explicitly in our code, for example in a user authentication program we should throw exception to client if the password is null. **throw** keyword is used to throw exception to the runtime to handle it.

```
import java.io.*;
class M
{
 void method()throws IOException
{
  throw new IOException("device error");
 }
}
class Testthrows4
{
   public static void main(String args[]) throws IOException
{
     M m=new M();
     m.method();
    System.out.println("normal flow...");
  }
}
```

```
Exception in thread "main" java.io.IOException: device error
```

Final Class, Final methods & Final variables:

Final methods:

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

The following fragment illustrates **final**:

```
class Bike
{
  final void run()
{
System.out.println("running");
}
}

class Honda extends Bike
{
   void run()
{
    System.out.println("running safely with 100kmph");
}
 public static void main(String args[]){
   Honda honda= new Honda();
```

```
  honda.run();
  }
}
```

Complie Time Error as Final methods cannot be overridden

Final Class:

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a final class:

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda();
  honda.run();
  }
}
```

```
Output:Compile Time Error
```

Final variable

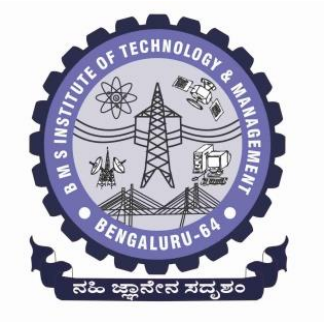
If you make any variable as final, you cannot change the value of final variable (It will be constant).

```
class Bike
{
 final int speedlimit=90; //final variable
 void run()
{
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new Bike9();
 obj.run(); //Compile Time Error
 }
}
```

Compile Time Error

B M S INSTITUTE OF TECHNOLOGY & MANAGEMENT

YELAHANKA, BENGALURU – 560064.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE -4 NOTES OF

OBJECT ORIENTED CONCEPTS -18CS45

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2018 -2019)

SEMESTER – IV

Prepared by,

Mr. Muneshwara M S

Asst. Prof, Dept. of CSE

VISION AND MISSION OF THE CS&E DEPARTMENT

Vision

To develop technical professionals acquainted with recent trends and technologies of computer science to serve as valuable resource for the nation/society.

Mission:

Facilitating and exposing the students to various learning opportunities through dedicated academic teaching, guidance and monitoring.

VISION AND MISSION OF THE INSTITUTE

Vision

To emerge as one of the finest technical institutions of higher learning, to develop engineering professionals who are technically competent, ethical and environment friendly for betterment of the society.

Mission

Accomplish stimulating learning environment through high quality academic instruction, innovation and industry-institute interface

MODULE 4 PACKAGES AND INTERFACES

The following concepts should be learn in this Module

Packages, Access Protection,Importing Packages.Interfaces. Multi Threaded Programming:Multi Threaded Programming: What are threads? How to make the classes threadable ; Extending threads; Implementing runnable; Synchronization; Changing state of the thread; Bounded buffer problems, producer consumer problems.

Text book 2: CH: 9 Ch 11: RBT: L1, L2, L3

Packages:

Packages in Java is a mechanism to encapsulate a group of classes, interfaces and sub packages.To create a package is quite easy: 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 you omit the **package** statement, the class names are put into the default package, which has no name.

This is the general form of the **package** statement:

package *pkg*;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

You can create a hierarchy of packages.

package *pkg1*[*.pkg2*[*.pkg3*]];

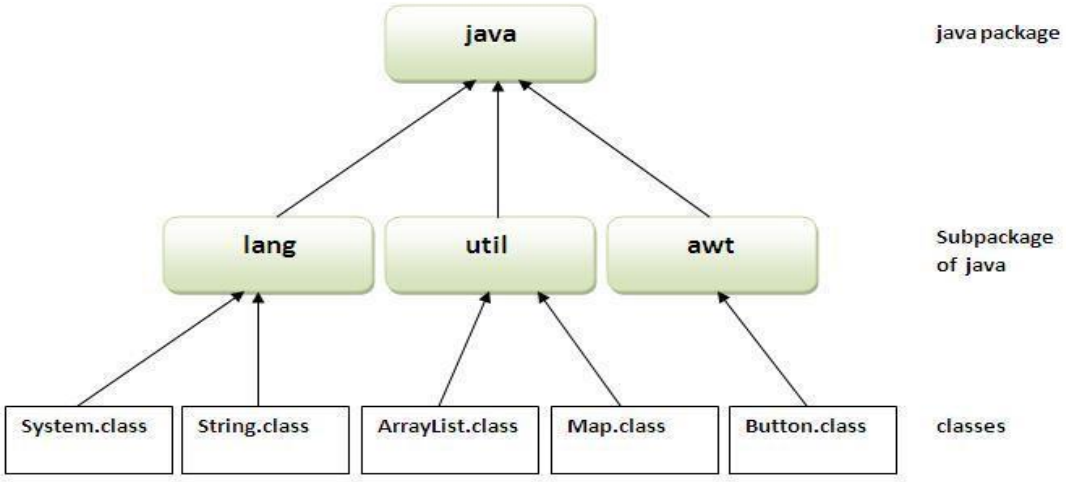
Ex: package java.awt.image;

Advantages of using a package

- ***Reusability:*** Reusability of code is one of the most important requirements in the software industry. Reusability saves time, effort and also ensures consistency. A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program.
- ***Easy*** to locate the files.
- In real life situation there may arise scenarios where we need to define files of the same name. This may lead to "name-space collisions". Packages are a way of avoiding "name-space collisions".

Package are categorized into two forms

- **Built-in Package**:-Existing Java package for example java.lang, java.util etc.
- **User-defined-package:-** Java package created by user to categorized classes and interface



How to compile & Run java package?

If you are not using any IDE, you need to follow this:

Compile :- javac -d . Simple.java

Run :- java mypack.Simple

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not sub packages. The import keyword is used to make the classes and interface of another package accessible to the current package.

```
//save by A.java

package pack;

public class A{

  public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.*;

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

A obj = new A();

obj.msg();

}

}
```

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

//save by A.java

```
package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

import pack.A;

class B{

public static void main(String args[]){

A obj = new A();

obj.msg();

}

}
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

```
//save by A.java

package pack;

public class A{

public void msg(){System.out.println("Hello");}

}

//save by B.java

package mypack;

class B{

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

```
pack.A obj = new pack.A();//using fully qualified name

obj.msg();}}
```

Access Specifiers

- private: accessible only in the class
- default : so-called “package” access — accessible only in the same package
- protected: accessible (inherited) by subclasses, and accessible by code in same package
- public: accessible anywhere the class is accessible, and inherited by subclasses

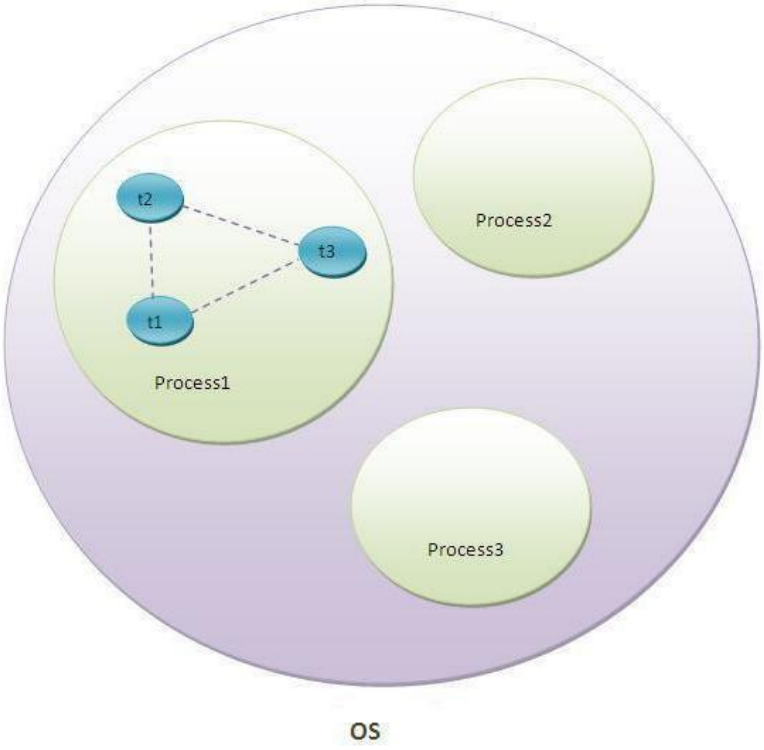
Notice that private protected is not syntactically legal.

Access By	private	package	protected	public
the class itself	yes	yes	yes	yes
a subclass in same package	no	yes	yes	yes
non-subclass in same package	no	yes	yes	yes
a subclass in other package	no	no	yes	yes
non-subclass in other package	no	no	no	yes

Multi-Threaded Programming

Thread:

- A thread is a lightweight sub process, a smallest unit of processing.
- It is a separate path of execution.
- Threads are independent, if there occurs exception in one thread, it doesn't affect other threads.
- It shares a common memory area.



t1, t2, t3 are threads

Java provides built-in support for multithreaded programming. **The process of executing multiple threads simultaneously is known as multithreading.** 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. Thus, multithreading is a specialized form of multitasking.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the processes is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.
- Less efficient

2) Thread-based Multitasking (Multithreading)

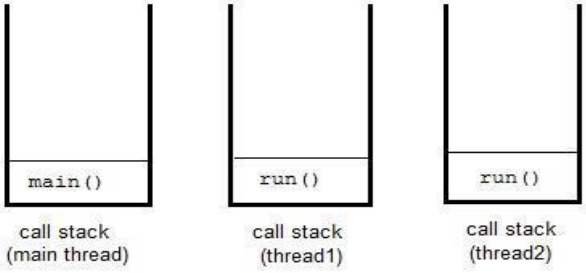
- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the threads is low.
- Highly efficient

Multithreading has several advantages over Multiprocessing such as;

- Threads are lightweight compared to processes
- Threads share the same address space and therefore can share both data and code.

- Context switching between threads is usually less expensive than between processes
- Cost of thread intercommunication is relatively low that that of process intercommunication
- Threads allow different tasks to be performed concurrently.

An instance of Thread class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack



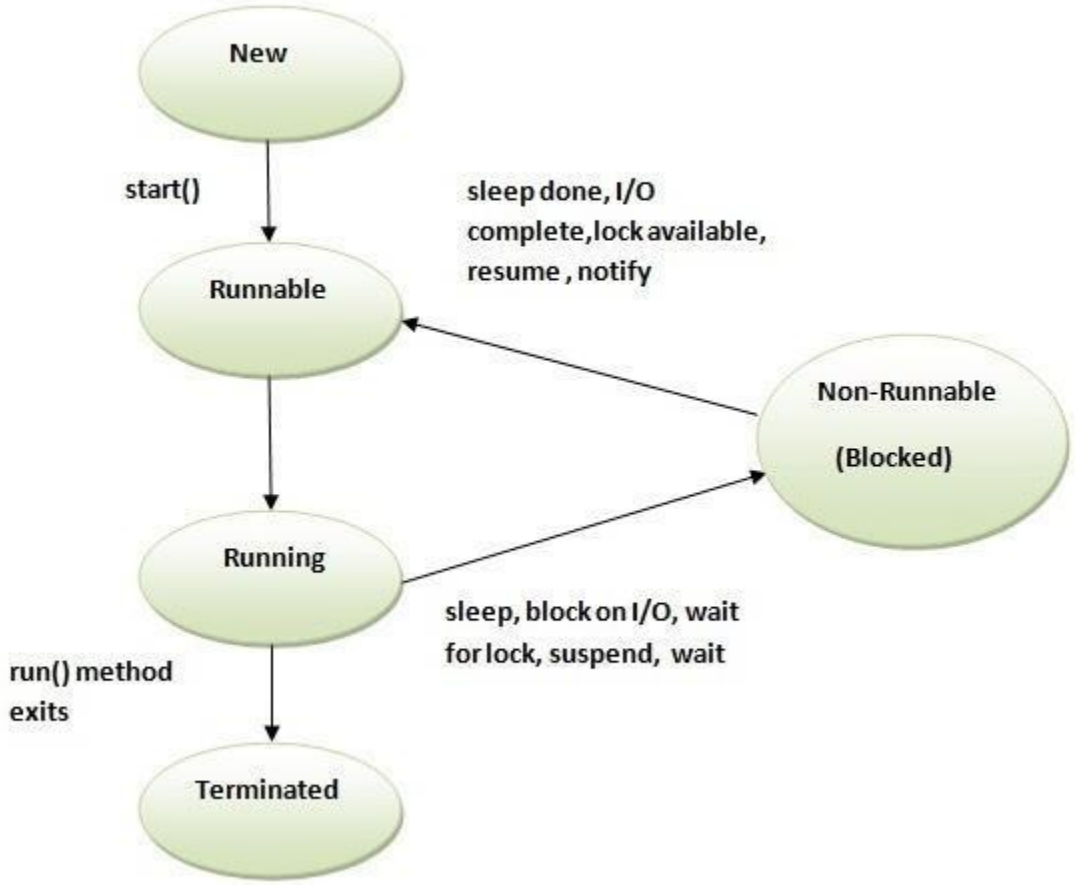
The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

• It is the thread from which other “child” threads will be spawned.

• Often, it must be the last thread to finish execution because it performs various shutdown actions.

Life cycle of a Thread (Thread States)



The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

When we call start() function on Thread object, it's state is changed to Runnable. The control is given to Thread scheduler to finish it's execution. Whether to run this thread instantly or keep it in runnable thread pool before running, depends on the OS implementation of thread scheduler.

3) Running

When thread is executing, it's state is changed to Running. Thread scheduler picks one of the thread from the runnable thread pool and change its state to Running. Then CPU starts executing this thread. A thread can change state to Runnable, Dead or Blocked from running state depends on time slicing, thread completion of run() method or waiting for some resources.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run. Waiting − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state, only when another thread signals the waiting thread to continue its execution.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

Thread Creation

When the thread starts it schedules the time slots for several sub threads and the JVM scheduler schedules the time slot to every thread based on round robin technique or priority based.

Method Signature	**Description**
String getName()	Retrieves the name of running thread in the current context in String format
void start()	This method will start a new thread of execution by calling run() method of Thread/runnable object.
void run()	This method is the entry point of the thread. Execution of thread starts from this method.
void sleep(int sleeptime)	This method suspend the thread for mentioned time duration in argument (sleeptime in ms)
void yield()	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.
void join()	This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution

There are two ways to create thread in java;

- Implement the Runnable interface (java.lang.Runnable)
- By Extending the Thread class (java.lang.Thread)

Extending Thread class

Creates a thread by a new class that extends Thread class. This creates an instance of that class. The extending class must override run() method which is the entry point of new thread.

```
class Multi extends Thread{

public void run(){

System.out.println("thread is running...");

}

public static void main(String args[]){

Multi t1=new Multi();

t1.start();

 }

}
```

thread is running

Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface, the class needs to implement the **run()** method, which is of form,

public void run()

- run() method introduces a concurrent thread into your program. This thread will end when run() returns.
- You must specify the code for your thread inside run() method.
- run() method can call other methods, can use other classes and declare variables just like any other normal method.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Syntax :

Thread(Runnable threadOb, String threadName);

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.

Ex : ***Thread t = new Thread(mt);***

```
class MyThread implements Runnable
{
 public void run()

 {
  System.out.println("concurrent thread started running..");
 }
}
```

```
class MyThreadDemo
{
 public static void main( String args[] )
 {
  MyThread mt = new MyThread();
  Thread t = new Thread(mt);
  t.start();
 }
```

concurrent thread started running..

Creating Multiple Threads by implementing Runnable Interface

```
package applet1;
import java.io.*;
import java.util.*;
class A implements Runnable {
  public void run() {
    try {
      for (int i = 5; i > 0; i--) {
        System.out.println(i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println( "Interrupted");
    }
```

```
    System.out.println(" exiting.");
  }
}
class B implements Runnable {

        public void run() {
          try {
            for (int i = 15; i > 10; i--) {
              System.out.println(i);
              Thread.sleep(1000);
            }

          } catch (InterruptedException e) {
            System.out.println("Interrupted");
          }

          System.out.println(" exiting.");
        }
      }
class multithread {
  public static void main(String args[]) {
    A obj1 = new A();
    B obj2= new B();
    Thread t1=new Thread(obj1);
    Thread t2=new Thread(obj2);
    t1.start();
    t2.start();
  }}
```

5

15

14

4

13

3

12

2

11

1

exiting.

exiting.

Notice the call to sleep(10000) in main( ). This causes threads B and A to sleep for ten seconds and ensures that it will finish last.

Note : Write a java application program for generating 3 threads to perform the following operations.

i)Reading n numbers ii) Printing prime numbers iii) Computing average of n numbers

isAlive() and join() methods

In java, isAlive() and join() are two different methods to check whether a thread has finished its execution.

The isAlive() method returns true if the thread upon which it is called is still running otherwise it returns false.

final boolean isAlive()

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("r1 ");
                try {
                Thread.sleep(500);
                }
                catch(InterruptedException ie) { }
                System.out.println("r2 ");
        }
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                MyThread t2=new MyThread();
                t1.start();
                t2.start();
```

```
            System.out.println(t1.isAlive());
            System.out.println(t2.isAlive());
        }
}
```

Output :

```
r1
true
true

r1
r2
r2
```

But, join() method is used more commonly than isAlive(). This method waits until the thread on which it is called terminates

final void join() throws InterruptedException

Using join() method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of join() method, which allows us to specify time for which you want to wait for the specified thread to terminate.

```
public class MyThread extends Thread
{
    public void run()
    {
        System.out.println("r1 ");
        try {
        Thread.sleep(500);
```

```
            }catch(InterruptedException ie){ }
            System.out.println("r2 ");

        }
        public static void main(String[] args)
        {
            MyThread t1=new MyThread();
            MyThread t2=new MyThread();
            t1.start();

            try{
                t1.join();              //Waiting for t1 to finish
            }catch(InterruptedException ie){}

            t2.start();
        }
}
```

r1

r2

r1

r2

In this above program join() method on thread t1 ensures that t1 finishes it process before thread t2 starts.

Specifying time with join()

If in the above program, we specify time while using join() with t1, then t1 will execute for that time, and then t2 will join it.

```
t1.join(1500);
```

Synchronization

At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is

called synchronization. The synchronization keyword in java creates a block of code referred to as critical section.

Key to synchronization is the concept of the **monitor**. A **monitor** is an object that is used as a mutually exclusive lock. 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. These other threads are said to be waiting for the monitor.

Why use Synchronization

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Thread Synchronization & Mutual Exclusive

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. static synchronization.

Using Synchronized Blocks

Synchronized block can be used to perform synchronization on any specific resource of the method. Synchronized block is used to lock an object for any shared resource. Scope of synchronized block is smaller than the method.

General Syntax :

```
synchronized (object)
{
//statement to be synchronized

}
```

```
class Table{


 void display(int n)
{

  synchronized(this)
  {

    for(int i=1;i<=5;i++)
        {
                System.out.println(n*i);
                try{
                 Thread.sleep(400);
                }
                catch(Exception e){
                System.out.println(e);
                }
```

```
    }
   }
  }//end of the method
}

class A extends Thread
{
Table t;

        A(Table t)
        {
        this.t=t;
        }
public void run(){
        t.display(5);
        }

}
class B extends Thread{
Table t;
        B(Table t)
         {
        this.t=t;
        }
public void run(){
        t.display(100);
```

```
        }
}

public class synch
{
public static void main(String args[])
{
Table obj = new Table();      //only one object

A t1=new A(obj);
B t2=new B(obj);
t1.start();
t2.start();
}
}
```

5
10
15
20
25
100
200
300
400
500

Using Synchronized Methods

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
class Table{


 synchronized void display(int n)
{
  for(int i=1;i<=5;i++)
        {
                System.out.println(n*i);
                try{
                 Thread.sleep(400);
                }
                catch(Exception e){
                System.out.println(e);
                }
    }
   }
  }//end of the method



class A extends Thread
{
Table t;
```

```
        A(Table t)
        {
        this.t=t;
        }

public void run(){
        t.display(5);
        }

}
class B extends Thread{
Table t;
        B(Table t){
        this.t=t;
        }
public void run(){
        t.display(100);
}
}

public class synch
{
public static void main(String args[])
{
Table obj = new Table();      //only one object
A t1=new A(obj);
```

```
B t2=new B(obj);
t1.start();

t2.start();
}
}
```

5

10

15

20

25

100

200

300

400

In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

Solution:

This problem can be implemented or solved by different ways in Java, classical way is using **wait** and **notify** method to communicate between Producer and Consumer thread and blocking each of them on individual condition like full queue and empty queue.

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( ) or notifyAll( ).

notify( ) wakes up a thread that called wait( ) on the same object.

notifyAll( ) wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

These methods are declared within Object, as shown here:

- ***final void wait( ) throws InterruptedException***
- ***final void notify( )***
- ***final void notify All( )***

```
public class ProducerConsumerTest {
  public static void main(String[] args) {
    Myclass c = new Myclass();
    Producer p1 = new Producer(c, 1);
    Consumer c1 = new Consumer(c, 1);
    p1.start();

    c1.start();
  }
}
```

```
class Myclass {
   private int contents;
   private boolean available = false;

   public synchronized int get() {
      while (available == false) {
         try {
            wait();
         } catch (InterruptedException e) {}
      }
      available = false;
      notifyAll();
      return contents;
   }
   public synchronized void put(int value) {
      while (available == true) {
         try {
            wait();
         } catch (InterruptedException e) { }
      }
      contents = value;
      available = true;
      notifyAll();

   }
}
class Consumer extends Thread {
```

```
  private Myclass Myclass;
  private int number;

  public Consumer(Myclass c, int number) {
    Myclass = c;

    this.number = number;
  }
  public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
      value = Myclass.get();
      System.out.println("Consumer #" + this.number + " got: " + value);
    }
  }
}
class Producer extends Thread {
  private Myclass Myclass;
  private int number;
  public Producer(Myclass c, int number) {
    Myclass = c;
    this.number = number;

  }
  public void run() {
    for (int i = 0; i < 10; i++) {
      Myclass.put(i);
```

```
        System.out.println("Producer #" + this.number + " put: " + i); try
        {
          sleep((int)(Math.random() * 100));
        } catch (InterruptedException e) { }
      }
    }}
```

Producer #1 put: 0

Consumer #1 got: 0

Producer #1 put: 1

Consumer #1 got: 1

Producer #1 put: 2

Consumer #1 got: 2

Producer #1 put: 3

Consumer #1 got: 3

Producer #1 put: 4

Consumer #1 got: 4

Producer #1 put: 5

Consumer #1 got: 5

Producer #1 put: 6

Consumer #1 got: 6

Producer #1 put: 7

Consumer #1 got: 7

Producer #1 put: 8

Consumer #1 got: 8

Producer #1 put: 9

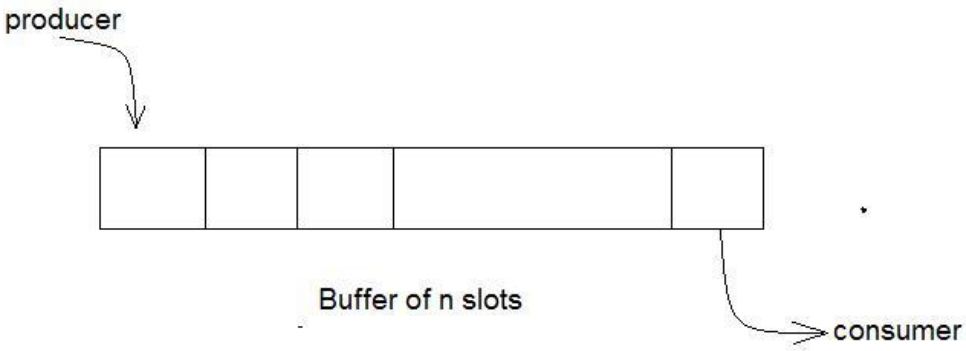
Consumer #1 got: 9

Bounded Buffer Problem

Bounded buffer problem, which is also called **producer consumer problem**, is one of the classic problems of synchronization.

Problem Statement:

There is a buffer of **n** slots and each slot is capable of storing one unit of data. There are two processes running, namely, **producer** and **consumer**, which are operating on the buffer.



A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

There needs to be a way to make the producer and consumer work in an independent manner.

```
import java.io.*;
import java.util.*;

 class Buffer
 {
private final int MaxBuffSize;
private int[] store;
private int BufferStart, BufferEnd, BufferSize;

public Buffer(int size)
{
MaxBuffSize = size;
BufferEnd = -1;
BufferStart = 0;
BufferSize = 0;
store = new int[MaxBuffSize];

}
public synchronized void insert(int ch)
{
try
{
        while (BufferSize == MaxBuffSize) {
        wait();
        }
        BufferEnd = (BufferEnd + 1) % MaxBuffSize;
```

```
        BufferSize++;
        notifyAll();
        }
catch (InterruptedException e)
        {

        }
}
public synchronized int delete() {
int ch=0;
        try {
while (BufferSize == 0) {
        wait();
        }

        ch = store[BufferStart];
BufferStart = (BufferStart + 1) % MaxBuffSize;
BufferSize--;
notifyAll();

} catch (InterruptedException e) {

}
        return ch;
}
}
```

```
class Consumer extends Thread {
private final Buffer buffer;
public Consumer(Buffer b) {
buffer = b;
}
public void run() {
while (!Thread.currentThread().isInterrupted()) {
int c = buffer.delete();
System.out.print(c);
}
}
}
class Producer extends Thread {
private final Buffer buffer;

public Producer(Buffer b) {
buffer = b;
}
public void run() {
for(int c=0;c<10;c++)
        buffer.insert(c);
}
}
```

```
class boundedbuffer {

public static void main(String[] args) {

System.out.println("program starting");

Buffer buffer = new Buffer(5); // buffer has size 5

Producer prod = new Producer(buffer);

Consumer cons = new Consumer(buffer);

prod.start();

cons.start();

try {

prod.join();

cons.interrupt();

} catch (InterruptedException e) {}
System.out.println("End of Program");



}

}
```

0

1

2

3

4

5

6

End of Program

Readers Writer Problem

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

Problem Statement:

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non-zero number of readers accessing the resource.

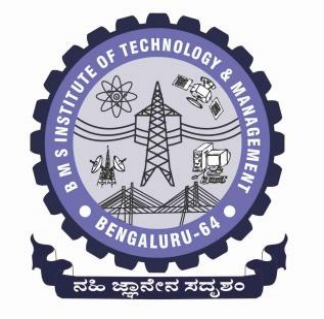
Solution:

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Refer any sample program for reader writer problem

B M S INSTITUTE OF TECHNOLOGY & MANAGEMENT

YELAHANKA, BENGALURU – 560064.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE -5 NOTES OF

OBJECT ORIENTED CONCEPTS -18CS45

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2018 -2019)

SEMESTER – IV

Prepared by,

Mr. Muneshwara M S

Asst. Prof, Dept. of CSE

VISION AND MISSION OF THE CS&E DEPARTMENT

Vision

To develop technical professionals acquainted with recent trends and technologies of computer science to serve as valuable resource for the nation/society.

Mission:

Facilitating and exposing the students to various learning opportunities through dedicated academic teaching, guidance and monitoring.

VISION AND MISSION OF THE INSTITUTE

Vision

To emerge as one of the finest technical institutions of higher learning, to develop engineering professionals who are technically competent, ethical and environment friendly for betterment of the society.

Mission

Accomplish stimulating learning environment through high quality academic instruction, innovation and industry-institute interface

MODULE 5 EVENT HANDLING AND SWINGS

Two event handling mechanisms; The delegation event model; Event classes; Sources of events; Event listener interfaces; Using the delegation event model; Adapter classes; Inner classes. **Swings:** Swings:

The origins of Swing; Two key Swing features; Components and Containers; The Swing Packages; A simple Swing Application; Create a Swing Applet; Jlabel and ImageIcon; JTextField;The Swing Buttons; JTabbedpane; JScrollPane; JList; JComboBox; JTable.

Text book 2: Ch 22: Ch: 29 Ch: 30, RBT: L1, L2, L3

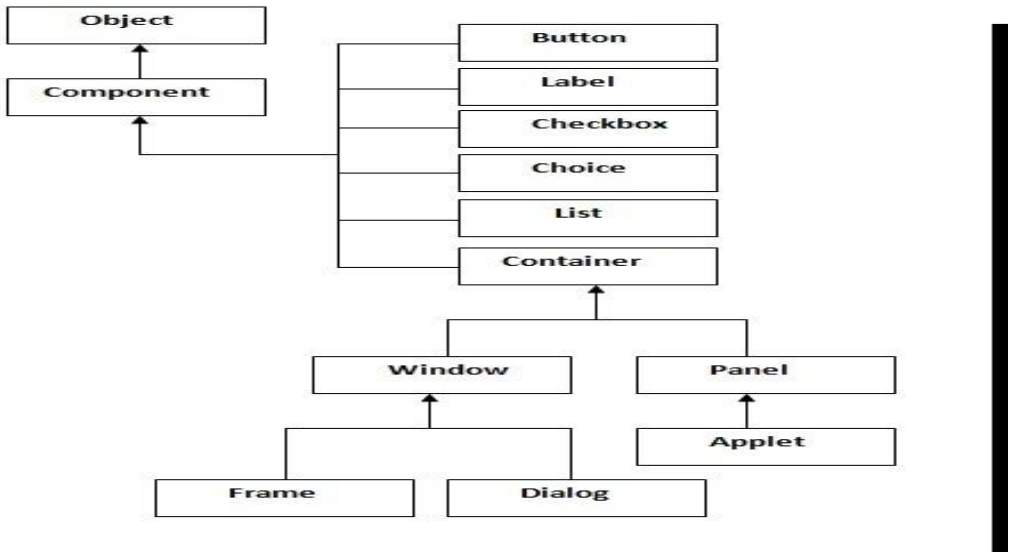
EVENT HANDLING

AWT

Java AWT (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.

- Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system.
- The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.
- In AWT components, (except Panel and Label) generates events when interacted by the user like clicking over a button or pressing enter key in a text field etc. Listeners handle the events.

JAVA AWT HIERARCHY



COMPONENTS OF EVENT HANDLING

Event handling has three main components

1. **Events :** An event is a change of state of an object. *Ex: Button presses, Text field is changed*
2. **Events Source :** Event source is an object that generates an event. *Ex : Button, Checkbox, List, menu Item, window*
3. **Event Listeners :** A listener is an object that listens to the event. A listener gets notified when an event occurs.

Special Group of interfaces

Ex: AdjustmentListener handles the events for scrollbar

ActionListener(Button)

TWO TYPES OF EVENT HANDLING MECHANISMS

The way in which events are handled by an applet changed significantly between the original version of Java(1.0) and modern version of Java, beginning with version 1.1 . The 1.0 method or event handling mechanism is still support the old 1.0 event model have been deprecated.

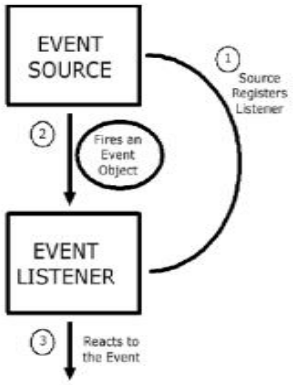
The modern approach is the way that events should be handled by all new programs, including those written for Java2. *Delegation Event Model.*

1. THE DELEGATION EVENT MODEL

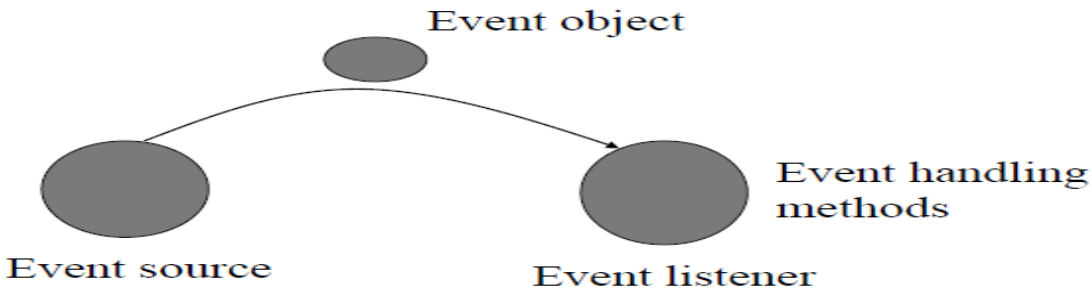
Defines standard and consistent mechanisms to generate and process events.

Listeners must register with a source in order to receive an event notification.

- a *source* generates an event and sends it to one or more *listeners*.
- In this scheme, the listener simply waits until it receives an event.
- Once an event is received, the listener processes the event and reacts to the event.



EVENT HANDLING MODEL OF AWT



EXAMPLE

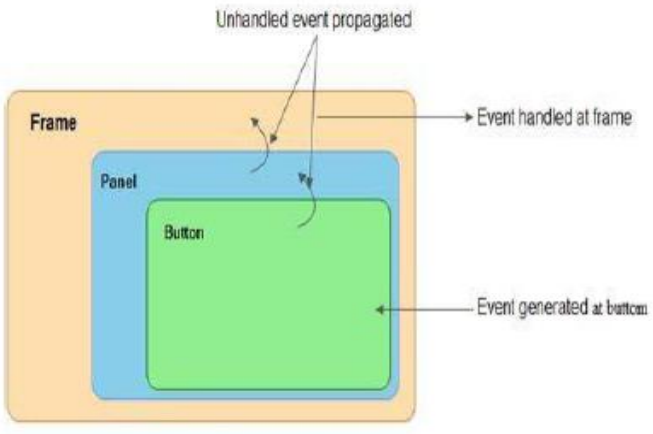
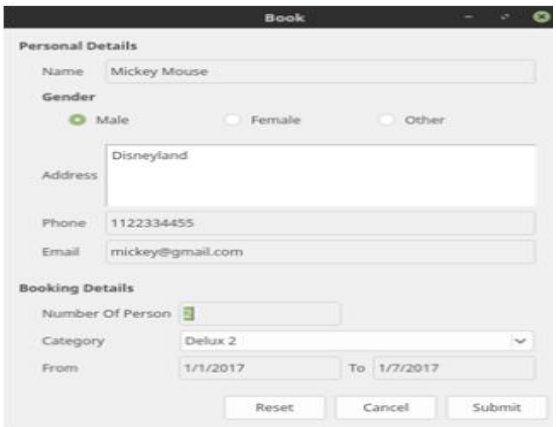


Figure Java 1.0 Event Handling Mechanism

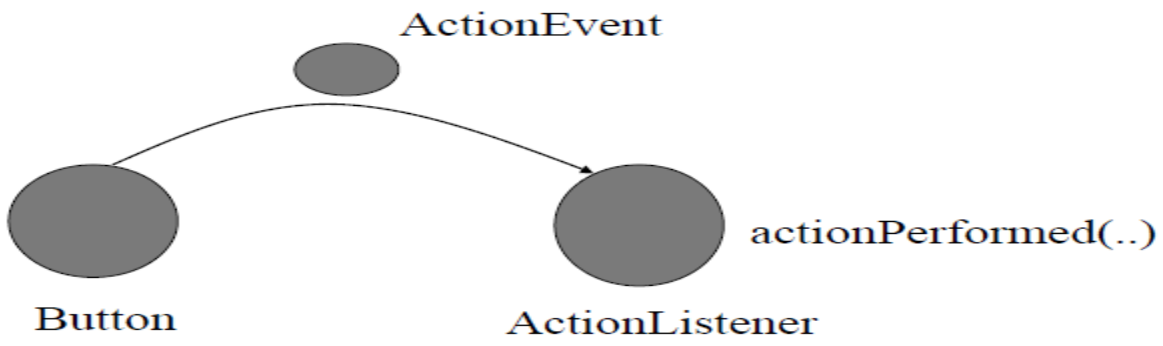


IMPORT

To implement event listener we need to import package : **Import java.awt.event.*;**

And event source in : **Import java.awt.*;**

ACTION EVENTS ON BUTTONS



ADVANTAGES OF EVENT DELEGATION MODEL

- Notifications are sent only to listeners that want to receive them which is more efficient way to handle event.
- The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.
- Accelerates the performance of the application which has multiple events.

DELEGATION EVENT MODEL

It has four main components/classes

1) Event Sources

2) Event classes

3) Event Listeners

4) Event Adapters

1. EVENT SOURCES

- A source is an object that generates an event.
- Event sources are components, subclasses of java.awt.Component, capable to generate events. The event source can be a button, TextField or a Frame etc.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.

2. EVENT CLASSES

The classes that are responsible for handling events in event handling mechanism are event classes.

- Java event classes are present in
- java.util and java.awt.event
    - EventObject is a superclass of all events. [java.util]
    - AWTEvent is a superclass of all AWT events that are handled by the delegation event model. [java.awt.event]

3. EVENT LISTENERS

- A listener is an object that is notified when an event occurs.
- It has two major requirements.
- First, it must have been registered with one or more sources to receive notifications about specific types of events.
- Second, it must implement methods to receive and process these notifications.

EVENT CLASSES

Event Classes	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released also when the enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

EVENT ADAPTERS

Java adapterclasses provide the default implementation of listener interfaces. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.

Adapters are abstract classes for receiving various events. The methods in these classes are empty. These classes exists as convenience for creating listener objects.

SIGNIFICANCE OF ADAPTER CLASS

Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

E.g. Suppose you want to use MouseClicked Event or method from MouseListener, if you do not use adapter class then unnecessarily you have to define all other methods from Mouse

But If you use adapter class then you can only define MouseClicked method and don't worry about other method definition because class provides an empty implementation of all methods in an event

listener interfaceseListener such as MouseReleased, MousePressed etc.

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

EXAMPLE FOR ADAPTER CLASS

```
public class sample
{
   Frame f;
   sample()
   {
      f=new Frame("Window Adapter");
      f.addWindowListener(new WindowAdapter()
      {
         public void windowClosing(WindowEvent e)
         {
            f.dispose();
         }
      }
      );
```

```
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
public static void main(String[] args)
 {
    new sample();

}
}
```

CREATE A FRAME

```
 import java.awt.*;
public class event1
  {
   event1()
   {
    Frame fm=new Frame();
    Label lb = new Label("welcome to java graphics");
    fm.add(lb);
    fm.setSize(300, 300);
    fm.setVisible(true);
   }
   public static void main(String args[])
   {
    event1 ta = new event1();
   }
  }
```

CLOSING FRAME

```
 fm.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent we)
{
System.exit(0);
}
}
);
```

ADDING LAYOUT

Note: Class should be extended with – extends Frame

FlowLayout fm = new FlowLayout(); setLayout(fm);

ADD BUTTON IN LAYOUT

Button btn=new Button("Hello World"); add(btn

ADD BUTTON

Button btn=new Button("Hello World"); fm.add(btn);

OTHER

setSize(400, 500); //setting size.

setTitle(“Welcome"); //setting title.

setLayout(new FlowLayout()); //set default layout for frame.

TEXT FIELD & CHECKBOX

TextField tf= new TextField();

fm.add(tf);

CheckboxGroup cbg = new CheckboxGroup();

Checkbox checkBox1 = new Checkbox("C++", cbg, true); checkBox1.setBounds(100,100, 50,50);

Checkbox checkBox2 = new Checkbox("Java", cbg, false);

checkBox2.setBounds(100,150, 50,50);

STEPS TO PERFORM EVENT HANDLING

Following steps are required to perform event handling:

- Implement the Listener interface and overrides its methods
- Register the component with the Listener

SYNTAX TO HANDLE THE EVENT

```
class className implements XXXListener
{
  .....
}
addcomponentobject.addXXXListener(this);
.....
```

- override abstract method of given interface and write proper logic public void methodName(XXXEvent e)

```
{
.....
}
.....
}
```

EVENT HANDLING FOR BUTTON COMPONENT

GUI Component (Event Source)	Event class	Listener Interface	Method (abstract method)
Button	ActionEvent	ActionListener	public void actionPerformed(ActionEvent e)

- The ActionEvent is generated when button is clicked or the item of a list is double clicked.
- The object that implements the ActionListener interface gets this ActionEvent when the event occurs.
- ActionEvent(java.lang.Object source, int id, java.lang.String command) Constructs an ActionEvent
- Class methods
- String getActionCommand() [Returns the command string associated with this action.
- int getModifiers() [Returns the modifier keys held down during this action event.]
- long getWhen() [Returns the timestamp of when this event occurred.] java.lang.String paramString() [Returns a parameter string identifying this action event.]

```
import java.awt.*;
import java.awt.event.*;
class A implements ActionListener
{
Frame f;
Button b1,b2,b3,b4;
A()
        {
        f=new Frame();
        f.setSize(500,500);
        f.setLayout(new BorderLayout());
        Panel p=new Panel();
        p.setBackground(Color.yellow);
        b1=new Button("Red");
        b2=new Button("green");
        b3=new Button("Blue");
        b4=new Button("Exit");
        p.add(b1);
        p.add(b2);
        p.add(b3);
        p.add(b4);
        f.add("North",p);
        f.add("North",p);
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        b4.addActionListener(this);
        f.setVisible(true);
        }
public void actionPerformed(ActionEvent e)
{
if(e.getSource().equals(b1))

        {
        f.setBackground(Color.red);
        }
else if(e.getSource().equals(b2))

{
f.setBackground(Color.green);
}
else if(e.getSource().equals(b3)

{
f.setBackground(Color.blue);
}
else if(e.getSource().equals(b4))

{
System.exit(0);
}
```

```
}

class ActionEventEx
{
public static void main(String[] args)
        {
        A a1=new A();
        }
}
```

EVENT HANDLING FOR TEXTFIELD

GUI Component	Event class	Listener Interface	Method (abstract method)
TextField	TextEvent	TextListener	public void textValueChanged(TextEvent e)

```
class AEvent extends Frame implements ActionListener
{
TextField tf;
Button b1,b;
AEvent(){
tf=new TextField();
tf.setBounds(60,50,170,20);
b=new Button("click me");
b.setBounds(100,120,80,30);
b1=new Button("Exit");
b1.setBounds(200,320,80,30);
//register listener
b.addActionListener(this); //passing current instance b1.addActionListener(this);
//add components and set size, layout and visibility
add(b);add(tf); add(b1);
```

```
setSize(300,300);
setVisible(true);
}
public void actionPerformed(ActionEvent e)
{ if(e.getSource().equals(b)) { tf.setText("Welcome");
}
Else if(e.getSource().equals(b1))
{
System.exit(0);
}
public static void main(String args[])
{
new AEvent();
}
}
```

CODE TO PERFORM ADDITION

```
public void actionPerformed(ActionEvent e)
{
String s1=tf1.getText();
String s2=tf2.getText();
int a=Integer.parseInt(s1);
int b=Integer.parseInt(s2);
int c=0;
if(e.getSource()==b1)
        {
        c=a+b;
        }
else if(e.getSource()==b2)
{
c=a-b;
}
String result=String.valueOf(c);
```

```
tf3.setText(result);
}
```

EVENT HANDLING FOR MOUSE

GUI Component	Event class	Listener Interface
Mouse	MouseEvent	MouseListener

Methods:

mousePressed(MouseEvent e): This method will execute whenever mouse button is pressed (not released).

mouseReleased(MouseEvent e): This method will execute whenever mouse button is only released (if already it is pressed).

mouseClicked(MouseEvent e): This method will execute whenever mouse button is both pressed and released.

mouseEntered(MouseEvent e): This method will execute whenever mouse cursor position is placed on specific location or component.

mouseExited(MouseEvent e): This method will execute whenever mouse cursor position is taken back from any location or component.

GUI Component	Event class	Listener Interface	Method (abstract method)
RadioButton, Checkbox, Lis Choice	ItemEvent	ItemListener	public void itemStateChanged(ItemEvent e

Mouse	MouseEvent	MouseListener	Void mousePressed(MouseEvent e) Void mouseReleased(MouseEvent e) Void mouseClicked (MouseEvent e) Void mouseEnetered(MouseEvent e) Void mouseExited(MouseEvent e)
TextField	TextEvent	TextListener	public void textValueChanged(TextEvent e
Button	ActionEvent	ActionListener	public void actionPerformed(ActionEvent e
Scrollbar	AdjustmentEver	AdjustmentListene	public void adjustmentValueChanged(ItemEvent e)
Keyboard	KeyEvent	KeyListener	Void keyPressed(keyEvent k) Void keyReleased(keyEvent k) Void keyTyped(keyEvent k)
Window	WindowEvent	WindowListener	void windowClosing(WindowEvent we)

ADJUSTMENTEVENT

The Class AdjustmentEvent represents adjustment event emitted by Adjustable objects like scroll bar.

Syntax :

- AdjustmentEvent(Adjustable source, int id, int type, int value)
- Constructs an AdjustmentEvent object with the specified Adjustable source, event type, adjustment type, and value.

Methods

1) **Adjustable getAdjustable()** [Returns the Adjustable object where this event originated.]
2) **int getAdjustmentType()** [Returns the type of adjustment which caused the value changed event.]
3) **int getValue()** [Returns the current value in the adjustment event.]
4) **boolean getValueIsAdjusting()** [Returns true if this is one of multiple adjustment events.]
5) **String paramString() [**Returns a string representing the state of this Event.]

PROGRAMS

1. WAP in java to implement all methods available to handle Mouse Events . Change background color for every event.
2. WAP in java to handle Keyboard. Implement the methods for Keyboard handling events. Display the characters
3. pressed, in Textfield . *[Hint : Override KeyTyped() and call e.getKeyChar(); and display it in Text fied]*
4. WAP to create a Registration Form that contains Username, Password, Languages Known. Display it on Frame
5. WAP in java to simulate simple calculator using event handling mechanisms.

MOUSE EVENT

```
import java.awt.*;
import java.awt.event.*;
class MouseListenerExample extends Frame implements MouseListener

{
Label l;
MouseListenerExample()

        {
        addMouseListener(this);
        l=new Label();
        l.setBounds(20,50,100,20);
        add(l);
        setSize(300,300);
        setLayout(null);
        setVisible(true);
        addWindowListener(new WindowAdapter()

                {
                public void windowClosing(WindowEvent we)
                        {System.exit(0);
                        }
                });
        }
public void mouseClicked(MouseEvent e) { l.setText("Mouse Clicked");
}
public void mouseEntered(MouseEvent e) { l.setText("Mouse Entered");
}
public void mouseExited(MouseEvent e) { l.setText("Mouse Exited");
}
public void mousePressed(MouseEvent e) { l.setText("Mouse Pressed");
}
public void mouseReleased(MouseEvent e) { l.setText("Mouse Released");
}
}
public class mouse

{
public static void main(String[] args)

        {
        new MouseListenerExample();
        }
}
```

KEYBOARD EVENT

```
import java.awt.*;
import java.awt.event.*;
class KeyListenerExample extends Frame implements KeyListener

{ Label l;
TextArea area;
KeyListenerExample()

        {
        l=new Label();
        l.setBounds(20,50,100,20);
        area=new TextArea();
        area.setBounds(20,80,300, 300);
        area.addKeyListener(this);
        add(l);add(area);
        setSize(400,400);
        setLayout(null);
        setVisible(true);
        }

public void keyPressed(KeyEvent e)

{
l.setText("Key Pressed");
}
public void keyReleased(KeyEvent e)

{
l.setText("Key Released");
}
public void keyTyped(KeyEvent e)

{
l.setText("Key Typed");
}
}
public class keyboard{
public static void main(String[] args)

{
new KeyListenerExample();
}
}
```

BUTTON & TEXT FIELD

```
import java.awt.*;
import java.awt.event.*;
public class test implements ActionListener
{
Frame fm;
Button b1,b2;
TextField tf;
test()
        {
        fm=new Frame();
        fm.setSize(500, 500); //Creating a frame.
        fm.setBackground(Color.cyan);
        fm.setLayout(new FlowLayout());
        Label lb = new Label("welcome to java graphics");        //Creating a label
        fm.add(lb);                                              //adding label to the frame.
        fm.setVisible(true);
        m.setResizable(true);//set frame visibilty true.
        fm.addWindowListener(new WindowAdapter()

                {
                public void windowClosing(WindowEvent we)
                        {
                        System.exit(0);
                        }
                });
        b1=new Button("Red");
        fm.add(b1);
        b2=new Button("Green");
        fm.add(b2);
        tf= new TextField();
        tf.setBounds(30, 250, 300, 500);
        tf.setSize(350, 30);
        fm.add(tf);
        b1.addActionListener(this);
        b2.addActionListener(this);
        }
public void actionPerformed(ActionEvent e)
{
if(e.getSource().equals(b1))

        {
        fm.setBackground(Color.red);
        tf.setText("Hurray Its red");
        }
else if(e.getSource().equals(b2))

{
fm.setBackground(Color.green);
tf.setText("Hurray Its green");
```

```
}
}
public static void main(String args[])
{
test ta = new test();
}
}
```

Event

Change in the state of an object is known as event i.e. change in state of source. The activities that is carried out between user and application is called an event.

For example,

1. clicking on a button,
2. moving the mouse,
3. entering a character through keyboard,
4. selecting an item from list,
5. scrolling the page are the activities that causes an event to happen

Types of Event

The events can be broadly classified into two categories:

Foreground Events -

Those events which require the direct interaction of user.

They are generated as consequences of a person interacting with the graphical components in Graphical User Interface.

For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page etc.

2. Background Events -

Those events that require the interaction of end user are known as background events.

Example of background events.

Operating system interrupts, hardware or software failure, timer expires

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs.

Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

Steps involved in event handling

1. The User clicks the button and the event is generated.
2. The object of concerned event class is created automatically and information about the source and the event get populated with in same object.
3. Event object is forwarded to the method of registered listener class.
4. the method is now get executed and returns.

Delegation Event Model

The Delegation Event Model has the following key participants namely:

Source -

- The source is an object on which event occurs.
- Source is responsible for providing information of the occurred event to it's handler.
- Java provide as with classes for source object.

Listener -

- It is also known as event handler.
- Listener is responsible for generating response to an event.
- From java implementation point of view the listener is also an object.

- Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

Event and Listener

Changing the state of an object is known as an event.

For example, click on button, dragging mouse etc.

The java.awt.event package provides many event classes and Listener interfaces for event handling.

1. Event Sources

Event sources are components, subclasses of java.awt.Component, capable to generate events. The event source can be a button, TextField or a Frame etc.

2. Event classes

- Almost every event source generates an event and is named by some Java class.
- For example, the event generated by button is known as ActionEvent and that of Checkbox is known as ItemEvent.
- All the events are listed in java.awt.event package.

3. Event Listeners interface

- The events generated by the GUI components are handled by a special group of interfaces known as "listeners".
- Listener is an interface.
- Every component has its own listener, say, AdjustmentListener handles the events of scrollbar.
- Some listeners handle the events of multiple components.

For example, ActionListener handles the events of Button, TextField, List and Menus. Listeners are from java.awt.event package.

4. Event Adapter Classes

- When a listener includes many abstract methods to override, the coding becomes heavy to the programmer.
- For example, to close the frame, you override seven abstract methods of WindowListener, in which, infact you are using only one method.
- To avoid this heavy coding, the designers come with another group of classes known as "adapters".
- Adapters are abstract classes defined in java.awt.event package.
- Every listener that has more than one abstract method has got a corresponding adapter class.

EVENT CLASSES	LISTENER INTERFACES
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Java Event Handling Code

We can put the event handling code into one of the following places:

1. Inner/Within class
2. Outer class
3. Anonymous class

SWINGS

Swing is a part of Java Foundation classes (JFC), the other parts of JFC are java2D and Abstract window toolkit (AWT). AWT, Swing & Java 2D are used for building graphical user interfaces (GUIs) in java. Unlike AWT, Java Swing provides platform-independent and lightweight components.

Main Features of Swing Toolkit

6. Platform Independent
7. Customizable
8. Extensible
9. Configurable
10. **Light Weight –** Swing components are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
11. **Rich Controls** - Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, and table controls.
12. **Highly Customizable** - Swing controls can be customized in a very easy way as visual appearance is independent of internal representation.
13. **Pluggable look-and-feel** - SWING based GUI Application look and feel can be changed at run-time, based on available values.

Difference between AWT and Swing

No	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes,colorchooser,tabbedpane etc.
5)	AWT **doesn't follow MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Origins of Swing

Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.

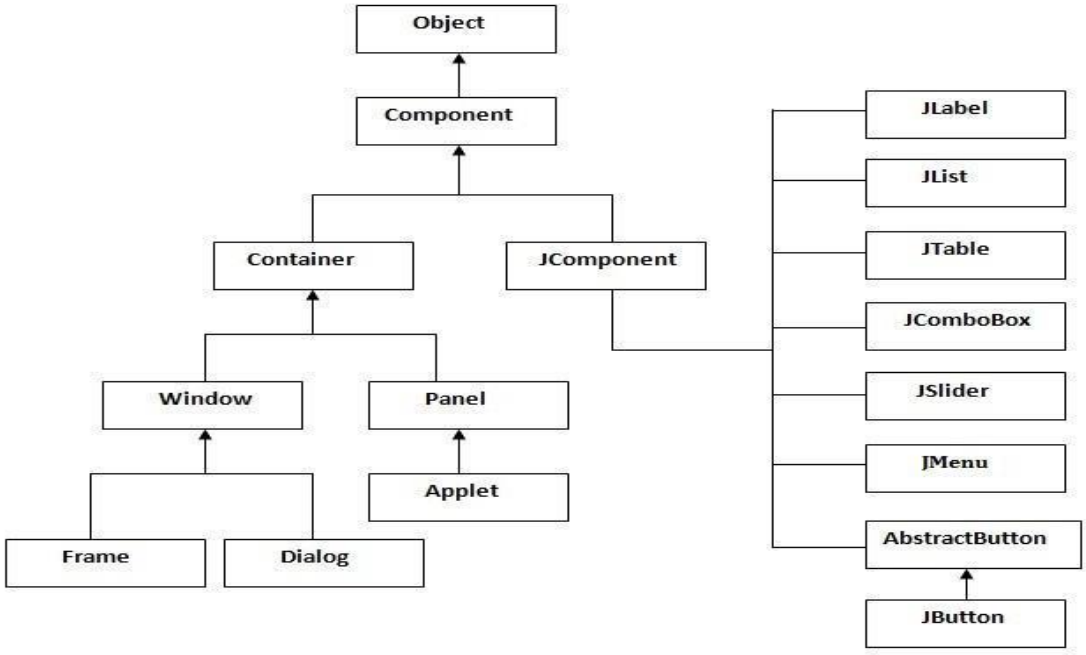
One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as **heavyweight**.

The use of native peers led to several problems.

- First, because of variations between operating systems, a component might look, or even act, differently on different platforms, which affected the potential variability for write once, run anywhere.
- Second, the look and feel of each component was fixed and could not be easily changed.
- Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component was always opaque.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc

Two Key Swing Features

- Swing Components Are Lightweight
- Swing Supports a Pluggable Look and Feel

Swing Components Are Lightweight

With very few exceptions, Swing components are **lightweight**. They are written entirely in Java and do not map directly to platform-specific peers. Thus, lightweight components are more efficient and more flexible.

Swing Supports a Pluggable Look and Feel

Swing supports a pluggable look and feel (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. SWING based GUI Application look and feel can be changed at run-time, based on available values.

Pluggable look-and-feels offer several important advantages.

- It is possible to define a look and feel that is consistent across all platforms.
- Conversely, it is possible to create a look and feel that acts like a specific platform. For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel.
- It is also possible to design a custom look and feel.
- Finally, the look and feel can be changed dynamically at run time.

MVC Architecture [Model-View-Controller]

In MVC terminology,

- **Model** corresponds to the state information associated with the component.

  For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.

- The **view** determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.
- The **controller** determines how the component reacts to the user.

  For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated.

Swing API architecture follows loosely based MVC architecture in the following manner.

- Model represents component's data.
- View represents visual representation of the component's data.
- Controller takes the input from the user on the view and reflects the changes in Component's data.

Swing component has Model as a separate element, while the View and Controller part are clubbed in the User Interface elements. Because of which, Swing has a pluggable look-and-feel architecture.

Components and Containers

A Swing GUI consists of two key items:

- components and
- containers.

In Java, a component is the basic user interface object and is found in all Java applications.

Components include lists, buttons, panels, and windows.

To use components, you need to place them in a container. A container is a component that holds and manages other components. Containers display components using a layout manager.

Components

Swing components inherit from the javax.Swing.JComponent class.

JPanel : JPanel is Swing's version of AWT class Panel and uses the same default layout, FlowLayout. JPanel is descended directly from JComponent.

JFrame : JFrame is Swing's version of Frame and is descended directly from Frame class.

The component which is added to the Frame, is refered as its Content.

JWindow : This is Swing's version of Window and hass descended directly from Window class. Like Window it uses BorderLayout by default.

JLabel : JLabel descended from Jcomponent, and is used to create text labels.

JButton : JButton class provides the functioning of push button. JButton allows an icon, string or both associated with a button.

JTextField : JTextFields allow editing of a single line of text.

JRadioButton is similar to JCheckbox, except for the default icon for each class. A set of radio buttons can be associated as a group in which only one button at a time can be selected. **JCheckBox** is not a member of a checkbox group. A checkbox can be selected and deselected, and it also displays its current state.

JComboBox is like a drop down box. You can click a drop-down arrow and select an option from a list. For example, when the component has focus, pressing a key that corresponds to the first character in some entry's name selects that entry. A vertical scrollbar is used for longer lists.

JList provides a scrollable set of items from which one or more may be selected. JList can

be populated from an Array or Vector.

JMenuBar An implementation of a menu bar.

JMenuItem An implementation of an item in a menu.

JOptionPane JOptionPane makes it easy to pop up a standard dialog box that prompts users for a value or informs them of something.

JPanel JPanel is a generic lightweight container.

JPasswordField JPasswordField is a lightweight component that allows the editing of a single line of text where the view indicates something was typed, but does not show the original characters.

JPopupMenu :An implementation of a popup menu -- a small window that pops up and displays a series of choices.

Containers

There are two types of containers namely,

- top-level containers and
- lightweight containers

Top-Level Containers

- The first are top-level containers: JFrame, JApplet, JWindow, and JDialog.
- These containers do not inherit **JComponent**.
- They do, however, inherit the AWT classes **Component** and **Container**.
- The top-level containers are heavyweight.

Each top-level container defines a set of **panes**. At the top of the hierarchy is an instance of **JRootPane**.

- **JRootPane** is a special container which extends JComponent and manages the appearance of JApplet and JFrame objects. It contains a fixed set of panes, namely,
  - *glass pane*,

- *content pane, and*
- *layered pane.*

- **Glass pane:** A glass pane is a top-level pane which covers all other panes. By default, it is a transparent instance of JPanel class. It is used to handle the mouse events affecting the entire container.

- **Layered pane:** A layered pane is an instance of JLayeredPane class. It holds a container called the content pane and an optional menu bar.

- **Content pane:** A content pane is a pane which is used to hold the components. All the visual components like buttons, labels are added to content pane. By default, it is an opaque instance of JPanel class and uses border layout. The content pane is accessed via getContentPane () method of JApplet and JFrame classes.

Lightweight Containers

- Lightweight containers lie next to the top-level containers in the containment hierarchy.
- They inherit **JComponent**.
- One of the examples of lightweight container is JPanel.
- As lightweight container can be contained within another container, they can be used to organize and manage groups of related components.

Swing Packages

Some of the packages of swing components that are used most are the following:

- *Javax.swing*
- *javax.swing.event*
- *javax.swing.plaf.basic*
- *javax.swing.table*
- *javax.swing.border*
- *javax.swing.tree*

Packages	Description
javax.swing	Provides a set of "lightweight" (all-Java language) components to the maximum degree possible, work the same on all platforms.
javax.swing.border	Provides classes and interface for drawing specialized borders around a Swing component.
javax.swi ng.colorchooser	Contains classes and interfaces used by the JcolorChooser component.
javax.swing.event	Provides for events fired by Swing components
javax.swing.filechooser	Contains classes and interfaces used by the JfileChooser component.
javax.swing.plaf	Provides one interface and many abstract classes that Swing uses to provide its pluggable look-and-feel capabilities.
javax.swing.plaf.basic	Provides user interface objects built according to the Basic look and feel.
javax.swing.plaf.metal	Provides user interface objects built according to the Java look and feel (once condenamed Metal), which is the default look and feel.
javax.swi ng.plaf.mult	Provides user interface objects that combine two or more look and feels.
javax.swing.table	Provides classes and interfaces for dealing with javax.swing.jtable
javax.swing.text	Provides classes and interfaces that deal with editable and noneditable text components
javax.swing.text.html	Provides the class HTML Editor Kit and supporting classes for creating HTML text editors.
javax.swing.text.rtf	Provides a class RTF Editor Kit for creating Rich- Text-Format text editors.
javax.swing.tree	Provides classes and interfaces for dealing with javax.swing.jtree
javax.swing.undo	Allows developers to provide support for undo/redo in applications such as text editors.

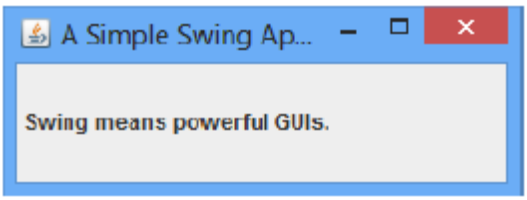
A Simple Swing Application

There are 2 ways of creating Java Swing Programs

1) Creating Swing Application 2) Creating Swing Applet

1)

```
import javax.swing.*;

class SwingDemo
{
SwingDemo()
{
    JFrame jfrm = new JFrame("A Simple Swing Application"); // Create a new frame

    jfrm.setSize(275, 100); // Set the size of a frame

    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //To close the frame

JLabel jlab = new JLabel(" Swing means powerful GUIs.");

jfrm.add(jlab);

jfrm.setVisible(true);

}

public static void main(String args[]) {

SwingUtilities.invokeLater ( new Runnable() { public void run() {new SwingDemo();
} });

}
}
```



Note:

jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

After this call executes, closing the window causes the entire application to terminate.

The general form of setDefaultCloseOperation( ) is shown here:

void setDefaultCloseOperation(int what)

The value passed in what determines what happens when the window is closed. There are several other options in addition to JFrame.EXIT_ON_CLOSE. They are shown here:

- DISPOSE_ON_CLOSE
- HIDE_ON_CLOSE
- DO_NOTHING_ON_CLOSE

6) Swing Applet

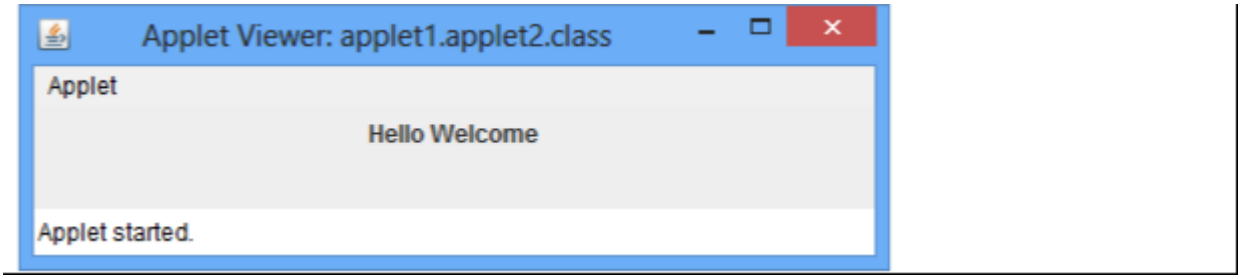
A Swing applet extends JApplet rather than Applet. JApplet is derived from Applet. Thus, JApplet includes all of the functionality found in Applet and adds support for Swing. JApplet is a top-level Swing container, which means that it is not derived from JComponent. Because JApplet is a toplevel container, it includes the various panes described earlier. This means that all components are added to JApplet's content pane in the same way that components are added to JFrame's content pane.

Swing applets use the same four life-cycle methods

- **init( ),**
- **start( ),**
- **stop( ), and**
- **destroy( ).**

```
import java.awt.*;

import javax.swing.*;

/*<APPLET CODE=sample.class WIDTH=510 HEIGHT=210></APPLET>*/

public class sample extends JApplet

{

JLabel jlab;


public void init(){

try

{

SwingUtilities.invokeAndWait(new Runnable(){

public void run()

{

sample();

}

});

}


catch(Exception e){

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

```
}
}
private void sample()
{
jlab=new JLabel("Hello Welcome");

      setSize(300,300);

 setLayout(new FlowLayout());
add(jlab);
}
}
```



Event Handling in Swings

Event: An event is a signal to the program that something has happened. It can be triggered by typing in a text field, selecting an item from the menu etc. The action is initiated outside the scope of the program and it is handled by a piece of code inside the program. Events may also be triggered when timer expires, hardware or software failure occurs, operation completes, counter is increased or decreased by a value etc.

Event handler: The code that performs a task in response to an event. is called event handler.

Event handling: It is process of responding to events that can occur at any time during execution of a program.

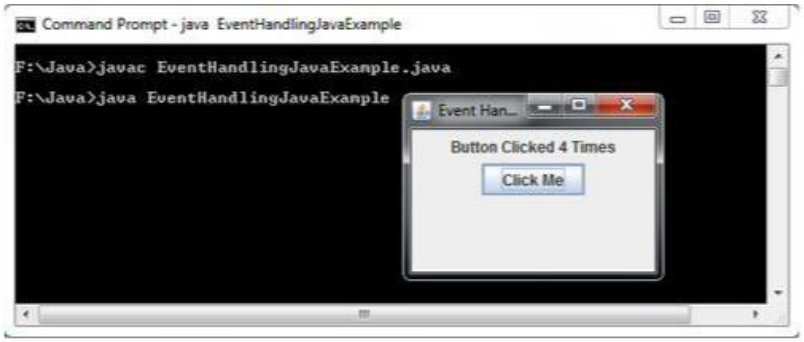
Event Source: It is an object that generates the event(s). Usually the event source is a button or the other component that the user can click but any Swing component can be an event source. The job of the event source is to accept registrations, get events from the user and call the listener's event handling method.

Event Listener: It is an object that watch for (i.e. listen for) events and handles them when they occur. It is basically a consumer that receives events from the source. To sum up, the job of an event listener is to implement the interface, register with the source and provide the eventhandling.

Listener interface: It is an interface which contains methods that the listener must implement and the source of the event invokes when the event occurs.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class EventExample extends JFrame implements ActionListener {

    private int count =0;
    JLabel lb;

    EventExample()
    {
      setLayout(new FlowLayout());
      lb = new JLabel("Button Clicked 0 Times");
```

```
        JButton btn=new JButton("Click Me");
        btn.addActionListener(this);
        add(lb);

        add(btn);
      }
      public void actionPerformed(ActionEvent e)
      {
        count++;
        lb.setText("Button Clicked " + count +" Times");
      }
}
 class sample
{
   public static void main(String args[])
   {

      EventExample frame = new EventExample();
      frame.setTitle("Event Handling Java Example");
      frame.setBounds(200,150,180,150);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);

   }
}
```


Command Prompt - java EventHandlingJavaExample
F:\Java>javac EventHandlingJavaExample.java
F:\Java>java EventHandlingJavaExample
Event Han...
Button Clicked 4 Times
Click Me

JLabel and ImageIcon within JApplet

A label is a simple control which is used to display text (non-editable) on the window. Some of the constructors defined by

JLabel class are as follows.

JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(String s)	Creates a JLabel instance with the specified text.
JLabel(Icon i)	Creates a JLabel instance with the specified image.
JLabel(String s, Icon i, int horizontalAlignment)	Creates a JLabel instance with the specified text, image, and horizontal alignment

ImageIcon(String filename);	It obtains the image in the file named filename.

```
import java.awt.*;

import javax.swing.*;

/*<APPLET CODE=JavaExampleLabelImageInJApplet.class
WIDTH=510 HEIGHT=210></APPLET>*/

public class sample extends JApplet
```

```
{

   public void init()
  {
    Container c = getContentPane();

    ImageIcon img = new ImageIcon("C:/Users/sri p/workspace/applet1/Koala.jpg");

    JLabel Lbl = new JLabel("Good Evening ",img,
    JLabel.CENTER);
    Lbl.setVerticalTextPosition(JLabel.BOTTOM);
    Lbl.setHorizontalTextPosition(JLabel.CENTER);
    c.add(Lbl);

 }
```


```
}
```

JTextField : Is extended from the JComponent class. This allows us to add a single line of text.

Syntax :

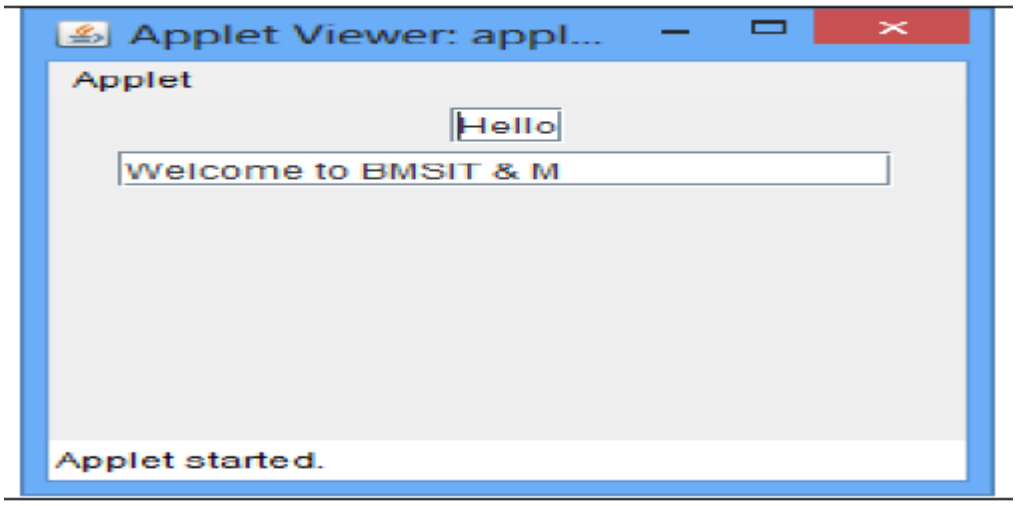
JTextField(int cols)

JTextField(String str, int cols)

JTextField(String str)

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="CreateNewJTextFieldExample" width=200 height=200>
</applet>
*/
public class applet2 extends JApplet{
   public void init(){
        Container c = getContentPane();
        c.setLayout(new FlowLayout());
          JTextField field2 = new JTextField("Hello");
          JTextField field4 = new JTextField("Welcome to BMSIT & M", 20);
```

```
        c.add(field2);

        c.add(field4);


    }
}
```



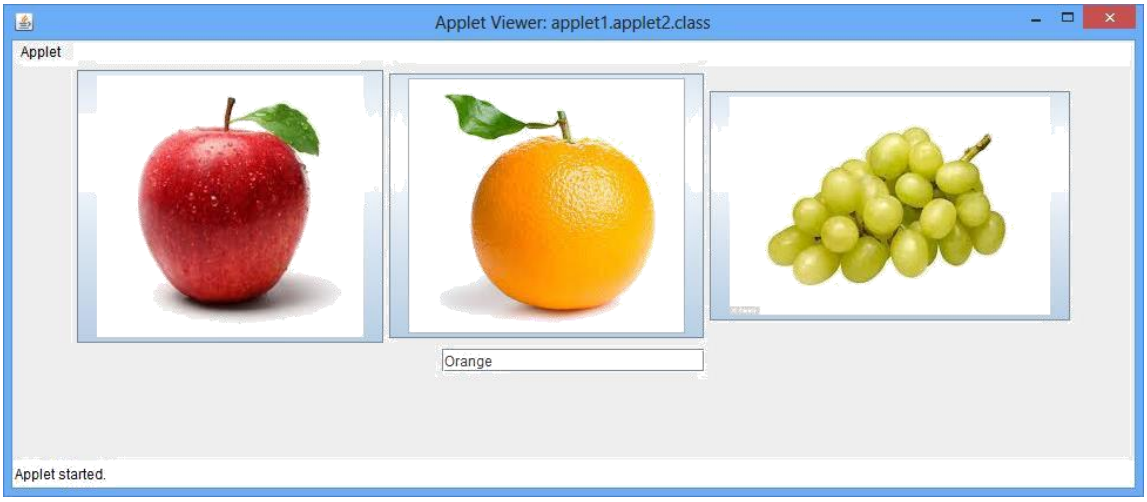
JButton [VERY VERY IMPORTANT TOPIC]

The Buttons component in Swing is similar to the Button component in AWT except that it can contain text, image or both. It can be created by instantiating the **JButton** class. A JButton can display Icon objects.

Method	Description
JButton()	Constructs a button with no text
JButton(String)	Constructs a button with the text entered
JButton(String, Icon)	Constructs abutton with thetext andinformedimage

```
import java.awt.*;
import java.awt.event.*;

import javax.swing.*;

public class applet2 extends JApplet implements ActionListener
{
 JTextField T;

    public void init()
{
        Container c = getContentPane();
         c.setLayout(new FlowLayout());

            ImageIcon appi = new ImageIcon("C:/Users/sri p/workspace/applet1/apple.jpg");
            JButton btn1 = new JButton(appi);
            btn1.setActionCommand("Apple");
            btn1.addActionListener(this);
            c.add(btn1);

            ImageIcon org = new ImageIcon("C:/Users/sri p/workspace/applet1/orange.jpg");
            JButton btn2 = new JButton(org);
            btn2.setActionCommand("Orange");
```

```
        btn2.addActionListener(this);
        c.add(btn2);

        ImageIcon grp = new ImageIcon("C:/Users/sri p/workspace/applet1/grapes.jpg");
       JButton btn3= new JButton(grp);
        btn3.setActionCommand("Grapes");
        btn3.addActionListener(this);

        c.add(btn3);

        T = new JTextField(100);
        c.add(T);
  }
  public void actionPerformed(ActionEvent e)
  {
  T.setText(e.getActionCommand());
 }
  }
```



JTabbedpane

Tabbed pane is a common user interface component that provides an easy access to more than one panel. Each tab is associated with a single component that will be displayed when the tab is selected.

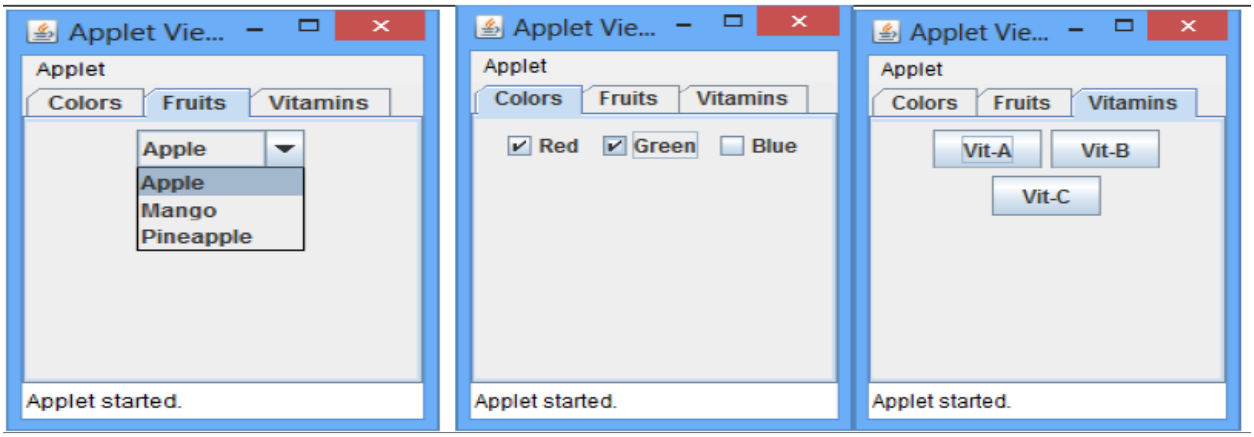
Syntax :

Void addTab(String str, Component comp);

```
import java.awt.*;
import javax.swing.*;

public class JTPDemo extends JApplet
{
  public void init()
  {
    JTabbedPane jt = new JTabbedPane();

    jt.add("Colors", new CPanel());
    jt.add( "Fruits", new FPanel());
    jt.add("Vitamins", new VPanel( ) ) ;
```

```
    getContentPane().add(jt);
  }
}
class CPanel extends JPanel
{
  public CPanel()
  {
    JCheckBox cb1 = new JCheckBox("Red");
    JCheckBox cb2 = new JCheckBox("Green");
    JCheckBox cb3 = new JCheckBox("Blue");
    add(cb1);    add(cb2);    add(cb3) ;
  }
}
class FPanel extends JPanel
{
  public FPanel()
  {
    JComboBox cb = new JComboBox();
    cb.addItem("Apple");
    cb.addItem("Mango");
    cb.addItem("Pineapple");
    add(cb);
  }
}
class VPanel extends JPanel
```

```
{
 public VPanel()
 {
   JButton b1 = new JButton("Vit-A");
   JButton b2 = new JButton("Vit-B");
   JButton b3 = new JButton("Vit-C");
   add(b1);    add(b2);    add(b3);
 }
}
```



JRadioButton

Syntax: ***JRadioButton(String str)***

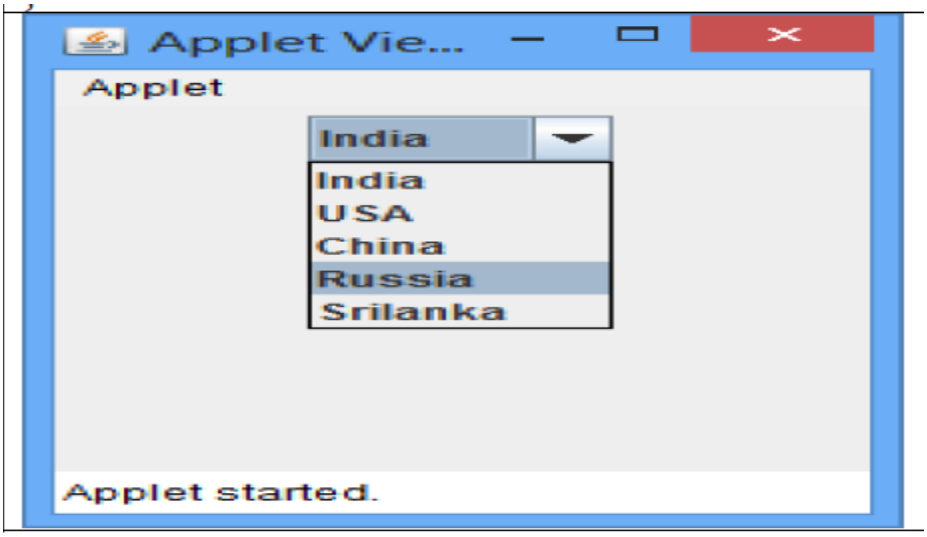
JComboBox

A combobox is a combination of textfield and dropdown list. JComboBox is a subclass of JComponent.

JList allows you to select multiple items in the list but Jcombobox selects only one item from the list

```
import java.awt.*;

import java.awt.event.*;
```

```
import javax.swing.*;

public class sample extends JApplet

{

public void init()

{

Container c = getContentPane();

c.setLayout(new FlowLayout());

JComboBox cb=new JComboBox();

 cb.addItem("India"); cb.addItem("USA"); cb.addItem("China"); cb.addItem("Russia");

cb.addItem("Srilanka"); c.add(cb);



}

}
```



JScrollPane

JScrollPanes are used in GUI development to restrict a widget to a certain size on the screen, then provide a way of scrolling up and down, left and right if the widget becomes any bigger than the 'Viewport' size. This is used primarily in JTextArea, JTable and JList where the data displayed by the widget is changed dynamically and can really affect the size and dimensions of the widget.

Some of the constructors defined by JScrollPane class are as follows.

- JSrollPane ()
- JScrollPane(Component component)
- JScrollPane(int ver, int hor)
- JScrollPane(Component component, int ver, int hor)

where,

- `component` is the component to be added to the scroll pane
- `ver` and `hor` specify the policies to display the vertical and horizontal scroll bar, respectively.

Some of the standard policies are:

- `HORIZONTAL_SCROLLBAR_ALWAYS`
- `HORIZONTAL_SCROLLBAR_AS_NEEDED`
- `VERTICAL_SCROLLBAR_ALWAYS`
- `VERTICAL_SCROLLBAR_AS_NEEDED`

```
import javax.swing.*;

import java.awt.event.*;

import java.awt.*;

public class sample extends JApplet

{
```

```
public void init()

{

Container cp = getContentPane();


JPanel jp = new JPanel( ) ;

jp.setLayout( new GridLayout( 20, 20 ) ) ;

Icon image = new ImageIcon("C:/Users/sri p/workspace/applet1/flower.jpg");

JLabel label = new JLabel(image);

jp.add(label);


int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED ;

int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED ;


JScrollPane js = new JScrollPane( jp, v, h ) ;

cp.add(js, BorderLayout.CENTER) ;

}}
```

JList

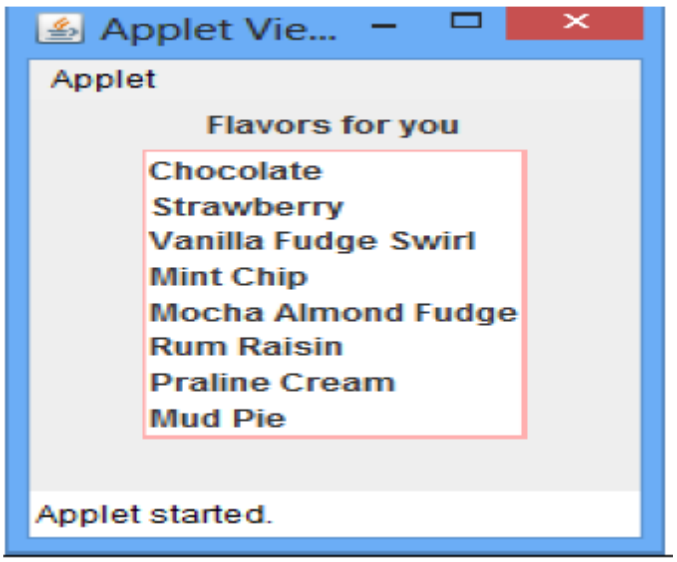
A list components allows user to select a single or multiple items from a given list of items by clicking on each. By default, a user can select multiple items.

A list can be created by instantiating a JList class. Relevant methods:

Method	Description
Public JList()	Creates an empty JList;
Public JList(Object [] listItems)	created with the items contained in the array of objects;
Public JList(VectorlistItems)	created with theVector elements;

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class sample extends JApplet
{
JLabel l;
public void init()
{
        String[] flavors = { "Chocolate", "Strawberry",
```

```
                "Vanilla Fudge Swirl", "Mint Chip", "Mocha Almond Fudge",
                "Rum Raisin", "Praline Cream", "Mud Pie" };
        Container cp = getContentPane();
          cp.setLayout(new FlowLayout());

      l= new JLabel("Flavors for you");
      cp.add(l);

     JList lst = new JList(flavors);
     cp.add(lst);

    Border brd = BorderFactory.createMatteBorder( 1, 1, 2, 2, Color.pink);
    lst.setBorder(brd);

    }
   }
```



Jtable

The JTable class is another Swing component that has no equivalent in AWT. JTable provides a very flexible possibility to create and display tables. It extends JComponent class. The JTable component of swing represents a two dimensional grid of objects. A JTable displays data in rows and column format. You can edit data values in a table as well. The JTable class of swing represents swing tables.

Constructors and methods of the class JTable.

Constructors and Methods	Description
JTable()	Constructs a default JTable Initialized to default data model, default column and selection model.
JTable(int nrows, int ncols)	Constructs a new JTable with *nrows* and *ncols* of empty cells using DelaultTableModel.
JTable(Object[][] rowdata, Object[ ] colnames)	Constructs a JTable to display the data contained in the two-dimensional array, row data, having the column names specified in the array, colnames.
JTable(Vector rowvector, Vector colvector)	Constructs a JTable to display the data available in vector of vectors, rowvector, with column names available in colvector.

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.* ;

public class sample extends JApplet

{

  public void init()
```

```
  {
    Container c = getContentPane();
    c.setLayout( new BorderLayout()) ;
    String fields[] = { "empid", "empname",  "empsal" } ;
    Object details[][] = {{ "1", "ABC", "4500.50" },          // instead of Object array
                          { "2","XYZ","4567.50" },            // String array can also be used
                          { "3", "PQR", "2246.30" },
                          { "4", "KLM", "3245.75" },
                          { "5", "Jyostna", "2500.25" }
                        };
    JTable jt = new JTable(details, fields);
// if the rows are more than height of the applet, scroll bar is added
    int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
    int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
    JScrollPane jsp = new JScrollPane(jt , v , h); c.add(jsp, "Center");
  }
}
```