**Class**

Object oriented programming constructs support a data type called class. A class encloses both the data and functions. The enclosed data and function in a class are called data member and member function respectively.

**Ex:**

 **Access\_Specifier class Classname**

 **{**

 **}**

**Object**

Defining variables of a class data type is known as class instantiation and such variables are called objects. Object is an instance of a class.

***Syntax:***

Classname obj = new Classname();

This single statement performs three actions:

1. **Declaration**:

Classname obj;

 **obj- Object Variable**

Classname obj is a variable declaration that declares to the compiler that the name obj will be used to refer to the Classname object.

1. **Instantiation**: new is a Java operator that creates the new object (allocates space for it).
2. **Initialization**: Classname () is a call to Classname constructor, which initializes the object.

**STATIC AND INSTANCE DATA MEMBER (OR) STATIC AND INSTANCE BLOCK**

**STATIC MEMBERS**

* Static variables and static methods are also common to classes and not tied to a java instance.
* Static methods cannot use the ‘this’ keyword.
* Good practice in java is that, static methods should be invoked with using the class name though it can be invoked using an object.

**ClassName.methodName(arguments) or objectName.methodName(arguments)**

**package** MODEL1;

**public** **class** Static1

{

 **int** Instancedatamember;

 **static** **int** *Staticdatamember*;

 **static**

 {

 System.*out*.println("This is static block"); **// Static block**

 }

 {

 System.*out*.println("This is Instance block"); **// Instance block**

 }

**public** Static1()

 {

 System.*out*.println("This is Constructor"); **// Constructor**

 }

 **public** **void** set**()// Instance method access static and instance members directly**

 {

 Instancedatamember=10; // Can be accessed directly

 *Staticdatamember*=20; // Can be accessed directly

 set1();// Can be accessed directly

 *display*();//Can be accessed directly

 }

 **public** **void** set1()

 {

 System.*out*.println("Set1 method is a Instance member Fn");

 }

 **static** **void** display**()//Static method can access only static members directly**

 {

 System.*out*.println("Display is a static method"+*Staticdatamember*);

 }

**public** **static** **void** main(String[] args)**//Static method access only static members**

**directly**

 {

 Static1 a=**new** Static1();

 **a.Instancedatamember=30**; // Need Instance “a” to access the Instance member

 **a.set();**// Need instance “a” to access the instance member

 ***Staticdatamember*=40;** //Static members can be accessed directly in static method

 ***display*();**//Static members can be accessed directly in static method

 **Static1.*display*();**//Class name to access the static member

 **Static1.*Staticdatamember*=50;**//Class name to access the static member

 System.*out*.println("Static member using Object a="+**a.*Staticdatamember***);

 **//System.out.println("Instance member using Class name="+Static1.Instancedatamember); //Cannot access the Instance member using Classname**

 **//Static1.set1();//Can't access instance method using Classname**

 }

}

**Output:**

This is static block

This is Instance block

This is Constructor

Set1 method is a Instance member Fn

Display is a static method

Display is a static method

Display is a static method

Static member using Object a=50

**ACCESS SPECIFIERS AND PACKAGES:**

Java allows to control access to classes, methods, and members so-called access specifiers.

Java offers four access specifiers, listed below in decreasing accessibility:

* [public](http://staff.science.uva.nl/~heck/JAVAcourse/ch4/ss2_2.html#public)
* [protected](http://staff.science.uva.nl/~heck/JAVAcourse/ch4/ss2_2.html#protected)
* [default (no specifier)](http://staff.science.uva.nl/~heck/JAVAcourse/ch4/ss2_2.html#friendly)
* [private](http://staff.science.uva.nl/~heck/JAVAcourse/ch4/ss2_2.html#private)

|  |  |  |
| --- | --- | --- |
| **VISIBILITY****MODE** | **SAME PACKAGE** | **OUTSIDE PACKAGE** |
| **SUB CLASS** | **NON-SUB** | **SUB CLASS** | **NON-SUB** |
| **PRIVATE** | **No** | **No** | **No** | **No** |
| **PROTECTED** | **Yes** | **Yes** | **Yes** | **No** |
| **PUBLIC** | **Yes** | **Yes** | **Yes** | **Yes** |
| **DEFAULT** | **Yes** | **Yes** | **No** | **No** |

**Packages**

**Package = directory**. Java classes can be grouped together in *packages*. A package name is the same as the directory (folder) name which contains the .java files.

## Package declaration syntax

The statement order is as follows. Comments can go anywhere.

1. Package statment .
2. Imports .
3. Class or interface definitions.

***Imports:***

**1. import package\_name.Classname;**

 Imports the class of the corresponding package. It will not import any instace members or static members directly.

import package\_name.Classname.instance\_member;//Error

**2. import package\_name.\*;**

 Imports all the classes of the package.

**3. import static package\_name.class\_name.static\_member;**

 Imports only the static members of the class in that package.

*Note: static import imports only the static members of the class.It will not import any classes or instance members.*

**A.java:**

**package** Access1;

**public** **class** A

{

 **private** **int** pri;

 **int** def;//DEFAULT SPECIFIER

 **protected** **int** pro;

 **public** **int** pub;

 **static** **public** **int** *s\_var*=10; //DEFAULT SPECIFIER

 **void** display()//DEFAULT SPECIFIER

 {

 pri=10;

 def=20; // all modifier can be accessed

 pro=30;

 pub=40;

 }

 **static** **void** s\_meth()//DEFAULT SPECIFIER

 {

 System.*out*.println("The Value of Static VAriable is" +*s\_var*);

 }

}

**B.java**

**package** Access1;

**public** **class** B **extends** A

{

 **void** display()

 {

 pri=10; // private members cannot be accessed

 def=20; // can be accessed

 pro=30; // can be accessed

 pub=40; // can be accessed

 *s\_var*=20; // can be accessed

 }

}

**C.java:**

**package** Access1;

**public** **class** C

{

 **void** display()

 {

 A v=**new** A(); // Non Subclass create the instance to access the member

 v.pri=10; // private members cannot be accessed

 v.def=20; // can be accessed

 v.pro=30; // can be accessed

 v.pub=40; // can be accessed

 v.*s\_*var=30; // can be accessed

 }

}

**D.java:**

**package** Access2;

**import** Access1.A; // Imports class A to make visible to class D

**public** **class** D **extends** A

{

 **void** display()

 {

 pri=10; // private & default members cannot be accessed

 def=20;

 pro=30; // can be accessed

 pub=40; // can be accessed

 *s\_var*=50; // can be accessed

 }

}

**E.java:**

**package** Access2;

**import** Access1.\*;// Imports all classes of Access1

**import** **static** Access1.A.*s\_var*; //Imports static members

**public** **class** E

{

 **void** display()

 {

 A v=**new** A();// Non Subclass create the instance to access the member

 v.pri=10; // private, default & protected members cannot be accessed

 v.def=20;

 v.pro=30;

 v.pub=40; // can be accessed

 *s\_var*=60; //public and static import member

 }

}

**POLYMORPHISM:**

Polymorphism is the ability of an object to take on many forms.

There are two types of polymorphism in java-**Runtime polymorhism( Dynamic polymorphism)**and**Compile time polymorphism (static polymorphism)**.

**COMPILETIME POLYMORPHISM:**

Compile time polymorphism is nothing but the method overloading in java. A class can have more than one methods with same name but with different number of arguments or different types of arguments or both. The call to a method is known at compile time itself.

Function overloading is an example of Compile time Polymorphism.

**Fun\_over.java:**

**package** Comp\_poly;

**public** **class** Fun\_over

{

 **void** methodA()

 {

 System.*out*.println ("Function Overloading");

 }

 **void** methodA(**int** num)

 {

 System.*out*.println ("methodA:" + num);

 }

 **void** methodA(**int** num1, **int** num2)

 {

 System.*out*.println ("methodA:" + num1 + "," + num2);

 }

 **double** methodA(**double** num)

 {

 System.*out*.println("methodA:" + num);

 **return** num;

 }

 }

**Main.java:**

**package** Comp\_poly;

**public** **class** Main

{

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

 {

 Fun\_over Obj = **new** Fun\_over();

 **double** result;

 Obj.methodA();

 Obj.methodA(20);

 Obj.methodA(20, 30);

 result = Obj.methodA(5.5);

 System.*out*.println("Answer is:" + result);

 }

}

**Output:**

Function Overloading

methodA:20

methodA:20,30

methodA:5.5

Answer is:5.5

**DYNAMIC POLYMORPHISM (or) ABSTRACT CLASS:**

***DYNAMIC OR RUNTIME POLYMORPHISM:***

The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

**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.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

***ABSTRACT CLASS:***

An *abstract class* is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

 abstract void moveTo(double deltaX, double deltaY);

 **Rules:**

* 1. Abstract class must be inherited.
	2. Abstract methods must be declared, not defined.
	3. Abstract methods must be overridden in the subclass.

**Shape.java:**

**package** Dyn\_Poly;

**abstract** **public** **class** Shape //Abstract Class

{

**abstract** **public** **void** area();//Abstract method

}

**Circle.java:**

**package** Dyn\_Poly;

**public** **class** Circle **extends** Shape//Inherits Shape properties

{

**public** **int** rad;

Circle(**int** r)

{

 rad=r;

}

**public** **void** area()//Overridden Method

{

 **double** a\_c=3.14\*rad\*rad;

 System.*out*.println("Area of Circle is"+a\_c);

}

}

**Rectangle.java:**

**package** Dyn\_Poly;

**public** **class** Rectangle **extends** Shape//Inherits Shape properties

{

 **public** **int** len,bre;

 **public** Rectangle(**int** l,**int** b)

 {

 len=l;

 bre=b;

 }

 **public** **void** area()//Overridden Method

 {

 **int** a\_r=len\*bre;

 System.*out*.println("Area of Circle is"+a\_r);

 }

}

**Main1.java:**

**package** Dyn\_Poly;

**public** **class** Main1

{

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

 {

 Circle c=**new** Circle(10);

 Rectangle r=**new** Rectangle(10, 20);

 Shape s;//s-Object Variable

 s=c;//Points to Circle's Object

 s.area();//Calls the Circle's area() at runtime

 s=r;//Points to Rectangle's Object

 s.area();//Calls the Rectangle's area() at runtime

 }

}

**FINAL KEYWORD:**

The **final keyword** in java is used to restrict the user. The final keyword can be used in many context. Final can be:

1. variable
2. method
3. class
* Final class cannot be sub classed.
* Final variables cannot be reinitialized.
* Final methods cannot be overridden.

**Bike.java:**

**package** Finale;

**final** **class** Bike

{

 **final** **int** speedlimit=90;//final variable

 **final** **void** run()

 {

 //speedlimit=400; // Final var can't be reassigned

 System.*out*.println("Base Class Bike");

 }

}

**Honda.java:**

**package** Finale;

**public** **class** Honda **extends** Bike //Can't inherit final class

{

**public** **void** run()//Can't Override Final Method

{

 System.*out*.println("Sub Class Honda");

}

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

{

 Bike b=**new** Bike();

 b.run();

 Honda d=**new** Honda();

 d.run();

 }

}

**FINALIZE METHOD:**

**Protected void finalize() throws Throwable**

 **{**

 **}**

* Every class inherits the finalize() method from java.lang.Object
* The method is called by the garbage collector when it determines no more references to the object exist
* The Object finalize method performs no actions but it may be overridden by any class
* Normally it should be overridden to clean-up non-Java resources ie closing a file
* If overriding finalize () it is good programming practice to use a try-catch-finally statement and to always call super .finalize() (JPL pg 47-48). This is a safety measure to ensure you do not inadvertently miss closing a resource used by the objects calling class

protected void finalize() throws Throwable

{

try

{

 close(); // close open files

 }

 finally

 {

 super.finalize();

 }

}

* Any exception thrown by finalize() during garbage collection halts the finalization but is otherwise ignored
* Finalize() is never run more than once on any object

**STRINGS:**

 String is an array of characters. In java string is a packaged class. Its an user defined datatype.

 DECLARATION:

 Strings can be created by new keyword and a constructor.

 Syntax:

 Char[] variable\_name=new Char[index];

 String variable\_name=new String();

 STRING INITIALIZATION:

 Char[] a={‘c’,’h’,’a’,’n’,’d’,’r’,’u’};

 String name=”Chandru”;

 SOME STRING FUNCTIONS:

* + String substring(int beginindex)

 Gives the sub string starting with index beginindex

* + String substring(int beginindex , int endindex)

 Gives the substring starting with beginindex(**inclusive**) and ends with endindex (**exclusive**)

* + String toLowerCase()

 Returns the given string in lowercase

* + String toUpperCase()

 Returns the given string in uppercase

 String trim()

 Trims the given string

 Length()

 Returns the length of the string

 Charat(int index)

 Returns the char at the given index

Program:-

**package** string;

**public** **class** Stri {

**public** **static** **void** main(String a1[]){

 String a=**new** String();

 String b=**new** String();

 String c=**new** String();

 **int** i;

 **char** s;

 a="hello";

 b="world";

 c=a.toUpperCase();

 System.*out*.println(c);

 c=a.toLowerCase();

 System.*out*.println(c);

 c=b.concat(a);

 System.*out*.println(c);

 c=a.substring(2);

 System.*out*.println(c);

 c=a.substring(1,3);

 System.*out*.println(c);

 i=a.length();

 System.*out*.println(i);

 s=a.charAt(3);

 System.*out*.println(s);

}

}

Output:

HELLO

hello

worldhello

llo

el

5

l

**ARRAY IN JAVA:**

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

**Declaring Array Variables:**

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

dataType[] arrayRefVar;

**Example:**

double[] myList;

**Creating Arrays:**

You can create an array by using the new operator with the following syntax:

arrayRefVar = new dataType[arraySize];

The above statement does two things:

* It creates an array using new dataType[arraySize];
* It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

dataType[] arrayRefVar = new dataType[arraySize];

Alternatively you can create arrays as follows:

dataType[] arrayRefVar = {value0, value1, ..., valuek};

The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

**Example:**

Following statement declares an array variable, myList, creates an array of 10 elements of double type, and assigns its reference to myList.:

double[] myList = new double[10];

Following picture represents array myList. Here myList holds ten double values and the indices are from 0 to 9.



**Processing Arrays:**

When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

**Example:**

Here is a complete example of showing how to create, initialize and process arrays:

public class TestArray {

 public static void main(String[] args) {

 double[] myList = {1.9, 2.9, 3.4, 3.5};

 // Print all the array elements

 for (int i = 0; i < myList.length; i++) {

 System.out.println(myList[i] + " ");

 }

 // Summing all elements

 double total = 0;

 for (int i = 0; i < myList.length; i++) {

 total += myList[i];

 }

 System.out.println("Total is " + total);

 // Finding the largest element

 double max = myList[0];

 for (int i = 1; i < myList.length; i++) {

 if (myList[i] > max) max = myList[i];

 }

 System.out.println("Max is " + max);

 }}

**Passing Arrays to Methods:**

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an int array:

public static void printArray(int[] array) {

 for (int i = 0; i < array.length; i++) {

 System.out.print(array[i] + " "); }}

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2:

printArray(new int[]{3, 1, 2, 6, 4, 2});

**Returning an Array from a Method:**

A method may also return an array. For example, the method shown below returns an array that is the reversal of another array:

public static int[] reverse(int[] list) {

 int[] result = new int[list.length];

 for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {

 result[j] = list[i];

 } return result; }

Arrays are sequence of variables sharing a common type.

Types of arrays:

* One dimensional arrays.
* Two dimensional arrays
* Multi dimensional arrays

One dimensional arrays:

 Declaration:

 int a[];

 Char b[];

 Float c[];

Allocation:

 a=new int[20];

 b=new char[20];

Initialization:

 a[0]=1;

 b[1]=’a’;

Two dimensional arrays:

 Declaration:

 Int a[][];

 Char b[][];

 Float c[][][];

Allocation:

 a=new int[20][20];

 b=new char[20][20];

Initialization:

 a[0][0]=1;

 b[1][1]=’a’;

Three dimensional arrays:

 Declaration:

 Int a[][][];

 Char b[][][];

 Float c[][][];

Allocation:

 a=new int[20][20][20];

 b=new char[20][20][20];

Initialization:

 a[0][0][0]=1;

 b[1][0][0]=’a’;

In java arrays can be extended to four dimension and so on.

**CONSTRUCTORS:**

Constructors are used to initialize memory for the objects.

When an object is created corresponding constructor is called.

Constructors initialize memory to the instance variables only.

TYPES:

* Default constructor
* Parameterized constructor

Example:

 **package** ME;

**public** **class** A {

 **public** **int** i;

 **public** A()

 {

 System.*out*.println("This is A's default Constructor");

 }

 **public** A(**int** i)

 {

 **this**();

 //super();

 System.*out*.println("This is A's Parametrized Constructor");

 **this**.i=i;

 }

}

**package** ME;

**public** **class** B **extends** A {

 **public** **int** i;

 **public** B(**int** b,**int** i)

 {

 **super**(b);

 //this();

 **this**.i=i;

 System.*out*.println("B' s Constructor");

 }

 **public** B()

 {

 System.*out*.println("B' s Constructor");

 }

 **void** display()

 {

 System.*out*.println("The value of the i is="+i);

 System.*out*.println("The value of the super.i is="+**super**.i);

 }

 **public** **static** **void** main(String a[])

 {

 B f=**new** B(10,20);

 f.display();

 }

}

This is A's default Constructor

This is A's Parametrized Constructor

B's Constructor

The value of the i is=20

The value of the super.i is=10

**INHERITANCE:**

**Inheritance** is the concept of deriving a class from an already existing class.

**Types of inheritance:**

* **Single inheritance**
* **Multiple inheritance(not possible in java but can be implemented by using interface)**
* **Multi-level inheritance**
* **Hybrid inheritance**
* **Hierarchal inheritance**

**Program implementing inheritance with using super and this keywords.**

**SUPER, THIS, SUPER (), THIS () (or) INHERITANCE**

* The keyword **super** refers to the base class
	+ **super( )**
		- invokes the base class constructor
		- base class constructors are automatically invoked
		- super() must be the first statement in a constructor
	+ **super**.method( )
		- invokes the specified method in the base class
	+ **super**.variable
		- to access the specified variable in the base class
* The keyword **this** refers to the current class instance. It is a reference to the object from which the method was invoked.

- **this()**

* + - this() must be the first statement in a constructor
		- To call a constructor from another overloaded constructor in the same class
	+ **this**.method( )
		- invokes the specified method in the same class
	+ **this**.variable
		- to access the specified variable in the same class

**Multilevel Inheritance:**

**Pi\_value.java:**

**package** Inheritance;

**public** **class** Pi\_value

{

 **double** pi;

Pi\_value()

{

 pi=3.14;

}

**void** value()

{

 System.*out*.println("The value of pi"+pi);

}

}

**Circle.java:**

**package** Inheritance;

**public** **class** Circle **extends** Pi\_value

{

**int** r;

**public** Circle()

{

 **super**();

 System.*out*.println("Multilevel Inheritance");

}

**public** Circle(**int** r)

{

 **this**();

 **this**.r=r;

}

**void** areaCircle()

{

 **double** area\_c;

 area\_c=pi\*r\*r;

 System.*out*.println("The value of pi is"+**super**.pi);

 System.*out*.println("Area of Circle is"+area\_c);

}

}

**Semicircle.java:**

**package** Inheritance;

**public** **class** Semicircle **extends** Circle

{

 **public** Semicircle(**int** r)

 {

 **super**(r);

 }

 **void** areasemiCircle()

 {

 **double** area\_s;

 area\_s=0.5\*pi\*r\*r;

 **super**.areaCircle();

 System.*out*.println("Area of SemiCircle is"+area\_s);

 }

}

**Main.java:**

**package** Inheritance;

**public** **class** Main

{

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

 {

 Semicircle s1=**new** Semicircle(5);

 s1.areasemiCircle();

 }

}

**Output:**

Multilevel Inheritance

The value of pi is3.14

Area of Circle is78.5

Area of SemiCircle is39.25

**Hierarical Inheritance:**

**Pi\_value.java:**

**package** Inheritance\_H;

**public** **class** Pi\_value

{

 **double** pi;

Pi\_value()

{

 pi=3.14;

}

**void** value()

{

 System.*out*.println("The value of pi"+pi);

}

}

**Circle.java:**

**package** Inheritance\_H;

**public** **class** Circle **extends** Pi\_value

{

**int** r;

**public** Circle(**int** r)

{

 **this**.r=r;

}

**void** areaCircle()

{

 **double** area\_c;

 area\_c=pi\*r\*r;

 System.*out*.println("Area of Circle is"+area\_c);

}

}

**Semicircle.java:**

**package** Inheritance\_H;

**public** **class** Semicircle **extends** Pi\_value

{

 **int** r;

 **public** Semicircle(**int** r)

 {

 **this**.r=r;

 }

 **void** areasemiCircle()

 {

 **double** area\_s;

 area\_s=0.5\*pi\*r\*r;

 System.*out*.println("Area of SemiCircle is"+area\_s);

 }

}

**Main.java:**

**package** Inheritance\_H;

**public** **class** Main

{

 /\*\*

 \* **@param** args

 \*/

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

 {

 Circle c1=**new** Circle(5);

 c1.areaCircle();

 Semicircle s1=**new** Semicircle(5);

 s1.areasemiCircle();

 }

}

**Output:**

Area of Circle is78.5

Area of SemiCircle is39.25

**EXCEPTION HANDLING:-**

 All exceptions are instances of a class extended from **Throwable**class or **its subclass.**

**EXCEPTION HIERARCHY:-**







**TYPES OF EXCEPTIONS:-**

 **Checked exceptions – inability to acquire system resources (such as insufficient memory, file does not exist)**

 **Unchecked exceptions – exceptions that occur because of the user entering bad data, or failing to enter data at all.**

**GENERAL SYNTAX:**

**try**{

**//statements – one of which is capable of throwing an exception**

}

**catch**(ExceptionTypeNameobjName)

{

**//one or more statements to execute if this exception occurs**

}

**finally**

{

**//statements to be executed whether or not exception occurs**

}

**EXAMPLE:-**

**ArithmeticException:**

**publicclass**ExcepTest{

 **publicstaticvoid** main(String args[]) {

 **int** d, a;

 **try**

 {

 d = 0;

 a = 42 / d;

 System.*out*.println(a);

 System.*out*.println("This will not be printed.");

 }

 **catch** (ArithmeticException e) {

 System.*out*.println("Division by zero.");

 }

 System.*out*.println("After catch statement.");

 }

 }

**OUTPUT:-**

Division by zero.

After catch statement.

**ArrayIndexOutOfBoundsException:**

**publicclass**ExcepTest{

**publicstaticvoid** main(String args[]){

**try**

 {

 **int** size=3;

 **int** a[] = **newint**[size];

 **if**(size>=3)

 **thrownew**ArrayIndexOutOfBoundsException();

 **else**

 System.*out*.println("Access element three :" + a[3]);

 }

**catch**(ArrayIndexOutOfBoundsException e){

System.*out*.println("Exception thrown :" + e);

 }

System.*out*.println("Out of the block");

 }

}

 **STACK (EXCEPTION HANDLING)**

**package** JAVA4;

**class** Stack

{

 **int**stck[]=**newint**[10];

 **int**tos;

 Stack(){

 tos=-1;}

 **void** push(**int** item)

 {

 stck[++tos]=item;

 }

 **int** pop()

 {

 **return**stck[tos--];

 }

 **publicstaticvoid** main(String args[])

 {

 Stack mystack1=**new** Stack();

 Stack mystack2=**new** Stack();

 **try**

 {

 **for**(**int** i=0;i<11;i++)

 mystack1.push(i);

 }

 **catch** (Exception e) {

 System.*out*.println("Stack Overflow: "+e.toString());

 }

 **try**

 {

 System.*out*.println("Stack in mystack1:");

 **for**(**int** i=0;i<11;i++)

 System.*out*.println(mystack1.pop());

 }

 **catch** (Exception e) {

 System.*out*.println("Stack Underflow: "+e.toString());

 }

 }

}

**USER DEFINED EXCEPTION:-**

A programmer makes user defined exception to extend the Exception class which is subclass of Throwable class.

**MyException.java:**

**class**MyException**extends** Exception

{

 String str1;

MyException(String str2) {

str1=str2;

 }

**public** String toString(){

**return** ("Output String = "+str1) ;

 }

}

**ExcepTest.java:**

**class**ExcepTest{

**publicstaticvoid** main(String args[]){

**try**

 {

MyException Ex1=**new**MyException("Custom");

 **throw** Ex1;

// I'm throwing user defined custom exception above

 }

**catch**(MyExceptionexp){

System.*out*.println("Hi this is my catch block") ;

System.*out*.println(exp) ;

 }

 }

}

**OUTPUT:**

Hi this is my catch block

Output String = Custom

**NESTED TRY – CATCH AND MULTIPLE CATCH BLOCK**

**class**ExcepTest{

**publicstaticvoid** main(String args[]){

**try**

 {

**try**

 {

System.*out*.println("going to divide");

**int**b =39/0;

 }

**catch**(ArithmeticException e)

 {

 System.*out*.println(e);

 }

**try**

 {

**int** a[]=**newint**[5];

a[5]=4;

 }

**catch**(ArrayIndexOutOfBoundsException e)

 {

 System.*out*.println(e);

 }

**thrownew** Exception();

 }

**catch**(Exception e)

 {

 System.*out*.println("Exception Handled");

 }

System.*out*.println("Normal flow..");

 }

}

**OUTPUT:**

going to divide

java.lang.ArithmeticException: / by zero

java.lang.ArrayIndexOutOfBoundsException: 5

Exception Handled

Normal flow..

**Finally block:-**An optional finally block can be added at the end of the catch blocks to provide a set of statements that are always executed whether or not an exception occurs. Finally block is executed independent of exception and catch. It is executed before return statement.

The main usage of finally block is to do clean up job.

**EXAMPLE:-**

**publicclass**MyFinallyBlock {

**publicstaticvoid** main(String[] a){

/\*\*

 \* Exception will occur here, after catch block

 \* thecontol will goto finally block.

 \*/

**try**

 {

**int**i = 10/0;

 }

**catch**(Exception ex)

 {

System.*out*.println("Inside 1st catch Block");

 }

**finally**

 {

System.*out*.println("Inside 1st finally block");

 }

/\*\*

 \* In this case exception won't, after executing try block

 \* thecontol will goto finally block.

 \*/

**try**

 {

**int**i = 10/0;

 }

**catch**(Exception ex)

 {

System.*out*.println("Inside 2nd catch Block");

 }

**finally**

 {

System.*out*.println("Inside 2nd finally block");

 }

 }

}

**OUTPUT:-**

Inside 1st catch Block

Inside 1st finally block

Inside 2nd catch Block

Inside 2nd finally block

**Throws:-**

The **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.

**package** test;

**publicclass** Test

{

**void** method()**throws**IOException

 {

**thrownew**IOException();

 }

}

**package** test;

**publicclass** Throws{

**publicstaticvoid** main(String args[])

 {

**try**{

 Test t=**new**Test();

t.method();

 }

**catch**(Exception e)

 {

 System.*out*.println("Exception Handled");

 }

System.*out*.println("Normal flow...");

 }

}

**OUTPUT:-**

Exception Handled

Normal flow...

Interfaces

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.

All the methods of the interface need to be defined in the class **WHICH IMPLEMENTS THE INTERFACE.**

**Shape.java:**

**package UNIT3;**

**public interface Shape**

**{**

 **public String baseclass="shape";**

 **public void Draw();**

**}**

**Circle.java:**

**package UNIT3;**

**public class Circle implements Shape**

**{**

 **public void Draw()**

 **{**

 **System.out.println("Drawing Circle here");**

 **}**

**}**

**I\_Test.java:**

**package UNIT3;**

**public class I\_Test**

**{**

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

 **{**

 **Shape cs=new Circle();//Or Circle cs=new Circle();**

 **cs.Draw();**

 **}**

**}**

**A class can implement more than one interface**

**- Multiple inheritace can be implemented by interface.**

**Stack.java:**

interface Stack

{

public void POP();

public void PUSH(int x);

}

**Queue.java:**

interface Queue

{

public void ENQUEUE(int x);

public void DEQUEUE();

}

**List.java:**

class list implements stack, queue

{

public void POP()

{

System.out.println("POP MethodDefn");

}

public void PUSH(int x)

{

System.out.println("PUSH MethodDefn");

}

public void ENQUEUE(int x)

{ System.out.println("ENQUEUE MethodDefn"); }

public void DEQUEUE()

{ System.out.println("DEQUEUE MethodDefn"); }

}

Interface can be inherited another interface

- Also implement the methods from parent interface if you are imlement sub interface

interface stack

{

public void POP();

public void PUSH(int x);

}

interface queue extends stack

{

public void ENQUEUE(int x);

public void DEQUEUE();

}

class list implements stack, queue

{

public void POP()

{ System.out.println("POP MethodDefn"); }

public void PUSH(int x)

{ System.out.println("PUSH MethodDefn");}

public void ENQUEUE(int x)

{ System.out.println("ENQUEUE MethodDefn"); }

public void DEQUEUE()

{

System.out.println("DEQUEUE MethodDefn"); }

}

**THREADS:**

* Threads are light weight processes
* It is handling the process

**CONTROLLING THE MAIN THREAD (SYSTEM THREAD):**

**Example1:**

public class Step1

{

public static void main(String[] arg)

{

 try

{

 for(int i=5;i>0;i--)

 {

 System.out.println("Main Thread"+i);

 Thread.sleep(2000);

 }

 }

 catch(InterruptedException e)

{

 System.out.println("Main interrupted");

 }

 System.out.println("Exiting Main Thread");

}

}

**Example2:**

class CurrentThreadDemo

{

 public static void main(String args[])

 {

 Thread t = Thread.currentThread();

 System.out.println("Current thread: " + t);

 t.setName("My Thread");

 System.out.println("After name change: " + t);

 try

 {

 for (int n = 5; n > 0; n--)

 {

 System.out.println(n);

 Thread.sleep(1000);

 }

 }

 catch (InterruptedException e)

 {

 System.out.println("Main thread interrupted");

 }

 }

}

**THREAD CAN BE IMPLEMENTED IN TWO WAYS**

1. **Extending Thread Class**
2. **Implementing Runnable Interface**

**EXTENDING THREAD CLASS:**

**Example:**

public class ThreadSample extends Thread

{

 public void run()

 {

 try

 {

 for (int i = 5; i > 0; i--)

 {

 System.out.println("Child Thread" + i);

 Thread.sleep(1000);

 }

 }

 catch (InterruptedException e)

 {

 System.out.println("Child interrupted");

 }

 System.out.println("Exiting Child Thread");

 }

}

public class MainThread

{

 public static void main(String[] arg)

 {

 ThreadSample d = new ThreadSample();

 d.start();

 try

 {

 for (int i = 5; i > 0; i--)

 {

 System.out.println("Main Thread" + i);

 Thread.sleep(5000);

 }

 }

 catch (InterruptedException e)

 {

 System.out.println("Main interrupted");

 }

 System.out.println("Exiting Main Thread");

 }

}

**IMPLEMENTING RUNNABLE INTERFACE**

public class ThreadSample implements Runnable

{

 public void run()

 {

 try

 {

 for (int i = 5; i > 0; i--)

 {

 System.out.println("Child Thread" + i);

 Thread.sleep(1000);

 }

 }

 catch (InterruptedException e)

 {

 System.out.println("Child interrupted");

 }

 System.out.println("Exiting Child Thread");

 }

}

public class MainThread

{

 public static void main(String[] arg)

 {

 ThreadSample d = new ThreadSample();

 Thread s = new Thread(d);

 s.start();

 try

 {

 for (int i = 5; i > 0; i--)

 {

 System.out.println("Main Thread" + i);

 Thread.sleep(5000);

 }

 }

 catch (InterruptedException e)

 {

 System.out.println("Main interrupted");

 }

 System.out.println("Exiting Main Thread");

 }

}

**THREAD PRIORITY**

package thread;

public class MyThread1 extends Thread {

 MyThread1(String s)

 {

 super(s);

 start();

 }

 public void run()

 {

 for(int i=0;i<5;i++)

 {

 Thread cur=Thread.currentThread();

 cur.setPriority(Thread.MAX\_PRIORITY);

 int p=cur.getPriority();

 System.out.println("Thread Name"+Thread.currentThread().getName());

 System.out.println("Thread Priority"+cur);

 }

 }

}

class MyThread2 extends Thread {

 MyThread2(String s)

 {

 super(s);

 start();

 }

 public void run()

 {

 for(int i=0;i<5;i++)

 {

 Thread cur=Thread.currentThread();

 cur.setPriority(Thread.MIN\_PRIORITY);

 System.out.println(cur.getPriority());

 int p=cur.getPriority();

 System.out.println("Thread Name"+Thread.currentThread().getName());

 System.out.println("Thread Priority"+cur);

 }

 }

}

public class ThreadPriority {

 public static void main(String[] args)

 {

 MyThread1 m1=new MyThread1("MyThread1");

 MyThread2 m2=new MyThread2("MyThread2");

 }

 }

**THREAD SYNCHRONIZATION:**

package Thread;

public class SynThread

{

 public static void main(String args[])

 {

 share s = new share();

 MyThread m1 = new MyThread(s, "Thread1");

 MyThread m2 = new MyThread(s, "Thread2");

 MyThread m3 = new MyThread(s, "Thread3");

 }

}

class MyThread extends Thread

{

 share s;

 MyThread(share s, String str)

 {

 super(str);

 this.s = s;

 start();

 }

 public void run()

 {

 s.doword(Thread.currentThread().getName());

 }

}

class share

{

 public synchronized void doword(String str)

 {

 for (int i = 0; i < 5; i++)

 {

 System.out.println("Started:" + str);

 try

 {

 Thread.sleep(1000);

 }

 catch (Exception e)

 {

 }

 }

 }

}

## The Java Utility Package

The final Java package, java.util, contains a collection of utility classes.

**Data Structure Classes**

A variety of useful classes implementing standard computer science data structures: including BitSet, Dictionary, Hashtable, Stack and Vector. The java.util package also defines the Enumeration interface which provides a protocol for classes to count through a set of values.

**Date**

Use the Date class to create and manipulate calendar dates in a system-independent fashion.

**StringTokenizer**

This class converts a String of text into its tokens.

**Properties**

This class implements persistent properties. The properties table contains key/value pairs where both the key and the value are Strings. This class is used by the System class to implement System properties.

**Observer and Observable**

Classes that implement the Observer interface can "watch" Observable objects for state changes. When an Observable object changes it notifies all of its Observers of the change.

**Random-Number Generator**

The Random class implements a random-number generator.

**Enumeration**

The Enumeration interface defines a generic programming interface for iterating through a set of values.

**java.util.ArrayList**

/\*

 \* To change this license header, choose License Headers in Project Properties.

 \* To change this template file, choose Tools | Templates

 \* and open the template in the editor.

 \*/

package Util;

import java.util.ArrayList;

public class ArrayList1 {

 public static void main(String[] args) {

 //create an ArrayList object

 //ArrayList<Integer> arrayList = new ArrayList<Integer>();

 ArrayList arrayList = new ArrayList();

 /\*

 Add elements to Arraylist using

 boolean add(Object o) method. It returns true as a general behavior

 of Collection.add method. The specified object is appended at the end

 of the ArrayList.

 \*/

 arrayList.**add**(1);

 arrayList.**add**(2);

 arrayList.**add**(3);

 arrayList.**add**("Str");

 /\*

 Use get method of Java ArrayList class to display elements of ArrayList.

 Object get(int index) returns and element at the specified index in

 the ArrayList

 \*/

 System.out.println("Getting elements of ArrayList");

 System.out.println(arrayList.get(0));

 System.out.println(arrayList.get(1));

 System.out.println(arrayList.get(2));

 System.out.println(arrayList.get(3));

 arrayList.**remove**(1);

 System.out.println("After remove of ArrayList");

 System.out.println(arrayList.get(0));

 System.out.println(arrayList.get(1));

 System.out.println(arrayList.get(2));

 //System.out.println(arrayList.get(3));

 }

}

**java.util.Date**

import java.util.Date;

/\*

Most of the methods of the Java Date class have been depricated.

Java Calendar class should be used for date manipulation instead.

\*/

public class JavaDateExample{

 public static void main(String args[]){

 /\*

 Create date object with current date and time.

 \*/

 Date date = new Date();

 System.out.println("Today is " + date);

 }

}

/\*

OUTPUT of the above given Java Date Example would be :

Today is Sat Feb 04 18:10:21 IST 2005

\*/

**java.util.Arrays :**

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

**public static int binarySearch(Object[] a, Object key)**

Searches the specified array of Object ( Byte, Int , double etc) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, (-(insertion point + 1).

**public static boolean equals(long[] a, long[] a2)**

Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other premitive data types ( Byte, short, Int etc.)

**public static void fill(int[] a, int val)**

Assigns the specified int value to each element of the specified array of ints. Same method could be used by all other premitive data types ( Byte, short, Int etc.)

|  |
| --- |
| **public static void sort(Object[] a)** Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. Same method could be used by all other premitive data types ( Byte, short, Int etc.)**Arrays built-in Class Program:****package** Arrays;**import** java.util.Arrays;**class** Arraytest { **public** **static** **void** main(String[] args) { **int**[] myArray = **new** **int**[] { 3, 4, 6, 8, 2, 1, 9}; **int**[] myArray1 = **new** **int**[] { 3, 4, 6, 8, 2, 1, 9};   System.*out*.println("Before Sort:"+Arrays.*toString*(myArray));  Arrays.*sort*(myArray); System.*out*.println("After Sort:"+Arrays.*toString*(myArray));   System.*out*.println("Binary Search:"+Arrays.*binarySearch*(myArray, 8));  **boolean** b= Arrays.*equals*(myArray, myArray1); System.*out*.println("Equals:"+b);   **for**(**int** i = 0; i < myArray1.length; i++)  { Arrays.*fill*(myArray1,10); } System.*out*.println("Fill:"+Arrays.*toString*(myArray1));  }} |

**OUTPUT:**

Before Sort:[3, 4, 6, 8, 2, 1, 9]

After Sort:[1, 2, 3, 4, 6, 8, 9]

Binary Search:5

Equals:false

Fill:[10, 10, 10, 10, 10, 10, 10]

**INPUT/ OUTPUT:**

**Java I/O** (Input and Output) is used to process the input and produce the output based on the input.

Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

A stream is a sequence of data.In Java a stream is composed of bytes.

 In java, 3 streams are created for us automatically. All these streams are attached with console.

**1) System.out:**standard output stream

**2) System.in:**standard input stream

**3) System.err:**standard error stream

**Java Scanner:**

There are various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The **Java Scanner** class breaks the input into tokens using a delimiter that is whitespace bydefault. It provides many methods to read and parse various primitive values.

Java Scanner class is widely used to parse text for string and primitive types using regular expression.

|  |  |
| --- | --- |
| public String next() | it returns the next token from the scanner. |
| public String nextLine() | it moves the scanner position to the next line and returns the value as a string. |
| public byte nextByte() | it scans the next token as a byte. |
| public short nextShort() | it scans the next token as a short value. |
| public int nextInt() | it scans the next token as an int value. |
| public long nextLong() | it scans the next token as a long value. |
| public float nextFloat() | it scans the next token as a float value. |
| public double nextDouble() | it scans the next token as a double value. |

package IO;

import java.util.Scanner;

public class Scanner1

{

 public static void main(String args[])

 {

 Scanner sc=new Scanner(System.in);

 System.out.println("Enter your roll no");

 int rollno=sc.nextInt();

 System.out.println("Enter your name");

 String name=sc.next();

 System.out.println("Enter your fee");

 double fee=sc.nextDouble();

 System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);

 sc.close();

 }

}

**FILES:**

The **Java.io.File** class is an abstract representation of file and directory pathnames. Following are the important points about File:

* Instances may or may not denote an actual file-system object such as a file or a directory. If it does denote such an object then that object resides in a partition. A partition is an operating system-specific portion of storage for a file system.
* A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
* Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

package IO;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.PrintWriter;

import java.util.Scanner;

public class ScannerF {

 public static void main(String args[]) throws FileNotFoundException

 {

 String inf="D:\\ex.txt";

 String outf="D:\\out.txt";

 File file=new File(inf);

 File file1=new File(outf);

 Scanner sc=new Scanner(file);

 String line=new String();

 StringBuffer s=new StringBuffer();

 while(sc.hasNext())

 {

 line=sc.nextLine();

 s=s.append(line+"\_\_\_\_\_");

 }

 System.out.println(s);

 sc.close();

 PrintWriter out = new PrintWriter(file1);

 out.print(s);

 out.close();

 }

}

**FEATURES OF JAVA:**

**Simple**

 Looks familiar to existing programmers: related to C and C++:

 Omits many rarely used, poorly understood, confusing features of C++, like operator overloading, multiple inheritance, automatic coercions, etc.

 Contains no *goto* statement, but *break* and *continue*

 Has no header files and eliminated C preprocessor

 Eliminates much redundancy (e.g. no structs, unions, or functions)

 has no pointers

Added features to simplify:

 Garbage collection, so the programmer won't have to worry about storage management, which leads to fewer bugs.

 A rich predefined class library

**Object-Oriented**

Java is an object-oriented language, which means that you focus on the *data* in your application and *methods* that manipulate that data, rather than thinking strictly in terms of procedures.

In an object-oriented system, a *class* is a collection of data and methods that operate on that data. Taken together, the data and methods describe the state and behavior of an *object*. Classes are arranged in a hierarchy, so that a subclass can inherit behavior from its superclass.

Java comes with an extensive set of classes, arranged in *packages*, that you can use in your programs.

**Distributed**

 It has a spring-like transparent RPC system

 Now uses mostly tcp-ip based protocols like ftp & http

Java supports various levels of network connectivity through classes in the java.net package (e.g. the URL class allows a Java application to open and access remote objects on the internet).

**Interpreted**

The Java compiler generates *byte-codes*, rather than native machine code. To actually run a Java program, you use the Java interpreter to execute the compiled byte-codes. Java byte-codes provide an architecture-neutral object file format. The code is designed to transport programs efficiently to multiple platforms.

 rapid turn-around development

 Software author is protected, since binary byte streams are downloaded and not the source code

**Robust**

Java has been designed for writing highly reliable or robust software:

 language restrictions (e.g. no pointer arithmetic and real arrays) to make it impossible for applications to smash memory (e.g overwriting memory and corrupting data)

 Java does **automatic garbage collection**, which prevents memory leaks

 extensive compile-time checking so bugs can be found early; this is repeated at runtime for flexibilty and to check consistency

**Secure**

Security is an important concern, since Java is meant to be used in networked environments. Without some assurance of security, you certainly wouldn't want to download an applet from a random site on the net and let it run on your computer. Java's memory allocation model is one of its main defenses against malicious code (e.g can't cast integers to pointers, so can't forge access). Furthermore:

 access restrictions are enforced (public, private)

 byte codes are verified, which copes with the threat of a hostile compiler

**Architecture-Neutral**

 compiler generates bytecodes, which have nothing to do with a particular computer architecture

 easy to interpret on any machine

**Portable**

Java goes further than just being architecture-neutral:

 no "implementation dependent" notes in the spec (arithmetic and evaluation order)

 standard libraries hide system differences

 the Java environment itself is also portable: the portability boundary is POSIX compliant

**High-Performance**

Java is an interpreted language, so it will never be as fast as a compiled language as C or C++. In fact, it is about 20 times as slow as C. However, this speed is more than enough to run interactive, GUI and network-based applications, where the application is often idle, waiting for the user to do something, or waiting for data from the network.

**Multithreaded**

Java allows multiple concurrent threads of execution to be active at once. This means that you could be listening to an audio clip while scrolling the page and in the background downloading an image. Java contains sophisticated synchronization primitives (monitors and condition variables), that are integrated into the language to make them easy to use and robust. The *java.lang* package provides a *Thread* class that supports methods to start, run, and stop a thread, and check on its status.

**Dynamic**

Java was designed to adapt to an evolving environment:

 Even after binaries have been released, they can adapt to a changing environment

 Java loads in classes as they are needed, even from across the network

 It defers many decisions (like object layout) to runtime, which solves many of the version problems that C++ has

 Dynamic linking is the only kind there is