**Java**

Java is an object-oriented programming language developed by Sun Microsystems, a company best known for its high-end UNIX workstations. Modeled after C++, the Java language was designed to be small, simple, and portable across platforms and operating systems, both at the source and at the binary level, which means that Java programs

(Applets and applications) can run on any machine that has the Java virtual machine installed.

**Java history:** is interesting to know. The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.

There are given the major points that describes the history of java.

1) **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling and file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.

**Java Is Platform Independent:**

Platform independence-that is, the ability of a program to move easily from one computer system to another-is one of the most significant advantages that Java has over other programming languages, particularly if your software needs to run on many different platforms. If you're writing software for the World Wide Web, being able to run the same program on many different systems is crucial to that program's success. Java is platform independent at both the source and the binary level.

Or

Platform independence means that a program can run on any computer system. Java

Programs can run on any system for which a Java virtual machine has been installed.

### Java Is Secured

|  |
| --- |
| Java is secured because: |
| * No explicit pointer * Programs run inside virtual machine sandbox. |

|  |
| --- |
| * **Classloader-** adds security by separating the package for the classes of the local file system from those that are imported from network sources. * **Bytecode Verifier-** checks the code fragments for illegal code that can violate access right to objects. * **Security Manager-** determines what resources a class can access such as reading and writing to the local disk. |
| These security are provided by java language. Some security can also be provided by application developer through SSL, JAAS, cryptography etc. Java Is Robust  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust. What is JVM? It is:   1. **A specification** where working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies. 2. **An implementation** Its implementation is known as JRE (Java Runtime Environment). 3. **Runtime Instance** Whenever you write java command on the command prompt to run the java class, and instance of JVM is created.   The JVM performs following operation:   * Loads code * Verifies code * Executes code * Provides runtime environment   JVM provides definitions for the:   * Memory area * Class file format * Register set * Garbage-collected heap * Fatal error reporting etc.  Internal Architecture of JVM:  1) Classloader: Classloader is a subsystem of JVM that is used to load class files. 2) Class (Method) Area: Class (Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods. 3) Heap: It is the runtime data area in which objects are allocated. 4) Stack:  |  | | --- | | Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return. | | Each thread has a private JVM stack, created at the same time as thread. | | A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes. |  5) Program Counter Register: PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed. 6) Native Method Stack: It contains all the native methods used in the application. 7) Execution Engine:  |  | | --- | | It contains: | | **1) A virtual processor** | | **2) Interpreter:** Read bytecode stream then execute the instructions. | | **3) Just-In-Time(JIT) compiler:** It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here the term? Compiler? refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU. | | |

**To run java file (Byte code):**

Java bytecodes are a special set of machine instructions that are not specific to any one

Processor or computer system. A platform-specific bytecode interpreter executes the Java bytecodes. The bytecode interpreter is also called the Java virtual machine or the Java runtime interpreter.

**Java applications:**

Java applications, however, are more general programs written in the Java language. Java applications don't require a browser to run; in fact, Java can be used to create all the kinds of applications that you would normally use a more conventional programming language to create.

**JAVA FEATURES**:-

Java 2 Features Additional Features of J2SE 5.0

* Compiled and interpreted Ease of Development
* Platform-Independent and Portable Scalability and Performance
* Object-Oriented Monitoring and Manageability
* Robust and Secure Desktop Client
* Distributed
* Familiar, Simple and Small
* Multithreaded and Interactive
* High Performance
* Dynamic and Extensible

**Compiled and Interpreted**:- First, java compiler translates source code into what is known as *bytecode* instructions. Bytecodes are not machine instructions and therefore, in the second stage, java interpreter generates machine code that can be directly executed by the machine that is running the java program.

**Platform-Independent and Portable**:- Java programs can be easily moved from one computer system to another, *anywhere* and *anytime.*

**Robust and Secure**:- Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types.

**Multithreaded and Interactive**:- Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programs.

**High Performance**:- Java performance is impressive for an interpreted language, mainly due to the use of intermediate bytecode.

**Java Support System**:-

*Support System* *Description*

* Internet Connection Local computer should be connected to the internet.
* Web Server A program that accepts requests for information and sends the required documents.
* Web Browser A program that provides access to WWW and runs java applets.
* HTML A language for creating hypertext for the Web.
* APPLET Tag For placing java applets in HTML document.
* Java Code Java code is used for defining java applets.
* Bytecode Compiled java code that is referred to in the APPLET tag and transferred to the user computer.
* Proxy Server An intermediate server between the requesting client workstation and the original server. It is typically implemented for ensuring security.
* Mail Server A computer server that facilitates exchange (sending and receiving) of electronic mails across networks.
* **Java Development Kit***:-* The java development kit comes with a collection of tools that are used for developing and running Java programs. They include:-
* applet viewer(for viewing java applets)
* javac(java compiler)
* java (java interpreter)
* javap(java disassembler)
* javah(for C header files)
* javadoc(for creating HTML documents)
* jdb(java debugger)

**COMPARISON B/W JAVA AND C++**

1) Java does not support operator overloading.

2) Java does not support multiple inheritance for this we use new feature called interface.

3) Java does not support global variable.

4) Java does not support pointers.

5) Java uses finalize () is case of destructor. There is no use of header files.

JAVA OFFERS TWO FLAVOUR OF PROGRAMING

1. Application programming
2. Applet programming.

**JAVA TOKENS***:-* Smallest individual units in a program are known as *tokens.* Java language includes five types of tokens*. They are:-*

* Reserved Keywords
* Identifiers
* Literals
* Operators
* Separators

**Variables and Data Types**

Variables are locations in memory in which values can be stored. Each one has a name, a type, and a value. Before you can use a variable, you have to declare it. After it is declared, you can then assign values to it.

**Java actually has three kinds of variables:**

1. Instance variables 2. Class variables 3. Local variables.

**Instance variables**: are used to define the attributes of a particular object.

**Class variables** are similar to instance variables, except their values apply to all that class's instances (and to the class itself) rather than having different values for each object.

**Local variables** are declared and used inside method definitions, for example, for index counters in loops, as temporary variables, or to hold values that you need only inside the method definition itself. Use local variables to store information needed by a single method and instance variables to store information needed by multiple methods in the object. Although all three kinds of variables are declared in much the same ways, class and instance variables are accessed and assigned in slightly different ways from local variables.

**Note:**Unlike other languages, Java does not have global variables-that is, variables that are global to all parts of a program. Instance and class variables can be used to communicate global information between and among objects. Remember that Java is an object-oriented language, so you should think in terms of objects and how they interact, rather than in terms of programs.

**Declaring Variables** To use any variable in a Java program, you must first declare it. Variable declarations consist of a type and a variable name:

int myAge;

String myName;

boolean isTired;

# **Notes on Variable Names** Variable names in Java can start with a letter, an underscore (\_), or a dollar sign ($). They cannot start with a number. After the first character, your variable names can include any letter or number. Symbols, such as %, \*, @, and so on, are often reserved for operators in Java, so be careful when using symbols in variable names.

# In addition, the Java language uses the Unicode character set. Unicode is a character set definition that not only offers characters in the standard ASCII character set, but also includes several thousand other characters for representing most international alphabets. This means that you can use accented characters and other glyphs as legal characters in variable names, as long as they have a Unicode character number above 00C0.

# **Note:** that the Java language is case sensitive, which means that uppercase letters are different from lowercase letters.

# **Primitive Types**

# The eight primitive data types handle common types for integers, floating-point numbers, characters, and boolean values (true or false). They're called primitive because they're built into the system and are not actual objects, which makes them more efficient to use.

# **Note** that these data types are machine-independent, which means that you can rely on their sizes and characteristics to be consistent across your Java programs.

# There are four Java integer types, each with a different range of values. All are signed, which means they can hold either positive or negative numbers. Which type you choose for your variables depends on the range of values you expect that variable to hold; if a value becomes too big for the variable type, it is silently truncated.

**Size and Range of Integer Types**:-

Type Size Minimum value Maximum value

byte One byte -128 127

short Two bytes -32,768 32,767

int Four bytes -2,147,483,648 2,147,483,647

long Eight bytes -9,223,372,036,854,775,808 9,223,372,036,854,775,807

**Q. What is object?**

Ans. The instance of a class is known as object. It mainly serve the following purpose:-

* Understanding the real world.
* Decomposition of a problem into objects depends upon judgment and nature of the problem

E.g.

class x

{

------;

void method( )

{

----------;

----------;

}

}

class p2

{

public static void main(String args[])

{

x obj=new x( ); //Object Creation

obj1.method( ); //Method call

}

}

**CLASS**: -

The object with the same data structure and behavior are grouped into a class.

Each class describes possibly infinite set of individual objects. The following points can be noted:-

* A class is a template that unites data and operations.
* A class is an abstraction of real world entities with similar properties.
* A class identifies a set of similar objects.
* Class is an implementation of abstract data types (ADT).

# **Constructor**: -

# Constructor is a public member method having same name as that of the class with not return type.

## class j

## {

## j( )// Constructor

## {

## val=0;

## }

## }

## 

## class Cons

## {

## public static void main(String args[])throws IOException

## {

## j obj=new j( );

## }

## }

**Constructor overloading**: -Constructor is overloaded according to the argument when the object is created.( Object passed as argument)

class j

{

int b;

j( )

{

b=100;

}

j( j obj)

{

b=obj.b;

}

void disp()

{

System.out.println("Value="+b);

}

protected void finalize()

{

/\*destructor\*/

}

}

class j8

{

public static void main(String args[])

{

j obj=new j();

obj.disp();

j obj1=new j(obj);

obj1.disp();

}

}

**Finalize( )**: - The Finalizer method is simply Finalize( ) and can be added to any class. Java calls that method whenever it is about to reclaim the space for that object. The finalize method should be explicitly define the tasks to be performed.

This is similar to Destructor in C++.

protected void finalize()

{

/\*destructor\*/

}

#### **Access specifier:** -

1. **Private:** -

The private member of a class has strict access control. Only the member method of the same class can access these members. The private members of the class are inaccessible outside the class thus, providing a mechanism for preventing accidental modification of the data members.

2 **Friendly Access:** -

When no access modifier is specified the member defaults to limited version of public accessibility known as friendly level of access.

1. **Protected :** -The visibility level of protected field lies in between the public access and friendly access i.e the protected modifier makes the field visible not only to all classes and sub classes in the same package but also to subclasses in other classes.
2. **Public:** -

The member of a class that is accessible outside the class should be declared in public section. All data members and methods declared in public section of a class can be accessed without a restriction from anywhere in a program either by method that belong to the class or by those external to the class.

**METHODS OVERLOADING**:- In java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called *method overloading*. Method overloading is used when objects are required to perform similar tasks but using different input parameters. This process is known as *polymorphism*.

class a

{

int x;

a()

{

x=100;

}

void disp()

{

System.out.println("Value="+x);

}

void disp(int y)

{

x=y;

System.out.println("Value="+x);

}

void disp(int y,int z)

{

x=y+z;

System.out.println("Value="+x);

}

}

class methodovr

{

public static void main(String args[])

{

a obj=new a();

obj.disp();

obj.disp(200);

obj.disp(100,200);

}

}

**Method Overriding:**

It is a concept that allows multiple methods to share the same name with different argument types. Method overloading implies that the method definition can have multiple forms assigning one or more method body to the same name.

class base

{

int a;

base()

{

}

base(int x)

{

this.a=x;

}

void display()

{

System.out.println("Value in Base:"+a);

}

}

class derived extends base

{

int b;

derived(int x,int y)

{

super(x);

this.b=y;

}

void display()

{

System.out.println("Value of a accessed from derived"+a);

System.out.println("Value of Derived:"+b);

}

}

class overr

{

public static void main(String args[])

{

derived obj=new derived(100,200);

obj.display();

}

}

|  |  |  |
| --- | --- | --- |
| **No.** | **Method Overloading** | **Method Overriding** |
| 1) | Method overloading is used to increase the readability of the program. | Method overriding is used to provide the specific  implementation of the method that is already  provided by its super class. |
| 2) | Method overloading is performed within class. | Method overriding occurs in two classes that have  IS-A (inheritance) relationship. |
| 3) | In case of method overloading, parameter must be different. | In case of method overriding, parameter must be  same. |
| 4) | Method overloading is the example of compile time polymorphism. | Method overriding is the example of run time  polymorphism. |
| 5) | In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading.  But you must have to change the parameter. | Return type must be same or covariant in method  overriding. |

**INHERITANCE: EXTENDING A CLASS** *:-* The mechanism of deriving a new class from an old one is called *inheritance*. The old class is known as the base class or super class or parent class and the new one is called the subclass or dcerived class or child class.

The inheritance allows subclass to inherit all the variables and methods of theit parent classes. Inheritance may take different points:

* Single inheritance(only one super class)
* Multiple inheritance(several super classes)
* Hierarchical inheritance(one super class , many subclasses)
* Multilevel inheritance(Derived from a derived class)

class a

{

int x;

a()

{

x=100;

}

void disp()

{

System.out.print("Base value="+x);

}

}

class b extends a

{

int y;

b()

{

y=200;

}

void disp1()

{

System.out.print("\nChild value="+y);

}

}

class c extends b

{

int z;

c()

{

z=300;

}

void disp2()

{

System.out.print("\nChild1 value="+z);

}

}

class Inherit

{

public static void main(String args[])

{

c obj=new c();

obj.disp();

obj.disp1();

obj.disp2();

}

}

**Multiple inheritance using Interface**

interface in

{

void disp();

}

class base

{

public void disp1()

{

System.out.println("Base Method");

}

}

class child extends base implements in

{

public void disp()

{

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

}

}

class multipleInheritance

{

public static void main(String args[])

{

child obj=new child();

obj.disp1();

obj.disp();

}

}

**FINAL VARIABLES AND METHODS**:-All methods and variables can be overridden by default in subclasses. If we wish to prevent the subclasses from overriding the members of the super class, we can declare them as final using the keyword final as a modifier.

final1 void show status( ) (.............)

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

1. variable
2. method
3. class

Eg:

final class a //It cannot be inherited

{

final int i=10; //it must be initialised

void show() //it cannot be overriden

{

System.out.println("Hello");

}

}

class b extends a

{

public void show()

{

System.out.println("How are you");

}

protected void finalize()

{

System.out.println("Garbage Collected");

}

}

class fina

{

public static void main(String args[])

{

b obj=new b();

obj.show();

System.gc();

}

}

**Super keyword:**  The super keyword refers to the base class of the class in which the keyword is used. It is used to refer to member values or method of the super class.

OR

The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

## **Usage of java super Keyword**

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

class base

{

int a;

base()

{

a=10;

}

void baseMethod()

{

System.out.println("Value Base:"+a);

}

}

class derived extends base

{

int b;

derived()

{

b=20;

}

void derivedMethod()

{

super.baseMethod();

System.out.println("Value Derived:"+b);

}

}

class inheritence

{

public static void main(String args[])

{

derived obj=new derived();

/\* obj.baseMethod();\*/

obj.derivedMethod();

}

}

# **Java instanceof operator**

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

# **Abstract class in Java**

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

abstract class a

{

abstract void show();//abstract method

void disp() //non abstract method

{

System.out.println("We are in class a");

}

}

class b extends a

{

void show()

{

super.disp();

System.out.println("We are in class b");

}

}

class c extends a

{

void show()

{

System.out.println("We are in class c");

}

}

class abst

{

public static void main(String args[])

{

a obj;

b obj1=new b();

obj=obj1;

obj.show();

c obj2=new c();

obj=obj2;

obj.show();

}

}

Difference between Abstract Class and Interface

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract**methods. | Interface can have **only abstract** methods. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can have static methods, main method and constructor**. | Interface **can't have static methods, main method or constructor**. |
| 5) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 6) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 7) **Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

**STRINGS**:- String manipulation is the most common part of many java programs. Strings represent a sequence of characters. The easiest way to represent a sequence of characters in java is by using a character array.

Example:- char charArray[ ] = new char[4];

charArray[0] = `j`;

charArray[1] = `a`;

charArray[2] = `v`;

charArray[3] = `a`;

**Some Most Commonly Used String Methods**

Method Call Taskperformed

s2 = s1.toLowerCase; Converts the string s1 to all lowercase

s2 = s1.toUpperCase; Converts the string s1 to all Uppercase

s1.equals(s2) Returns 'true' if s1 is equal to s2

s1.length( ) Gives the length of s1

s1.concat(s2) Concatenates s1 and s2

s1.substring(n,m) Gives substring starting from nth character

**WRAPPER CLASSES**:- Primitive data types may be converted into object types by using the wrapper classes contained in the java.lang package.

**autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known and autoboxing and vice-versa unboxing.

Wrapper Classes for Converting Simple Types

Simple Type Wrapper Class

boolean Boolean

char Character

double Double

float Float

int Integer

long Long

eg:

public class Wrap

{

public static void main(String args[])

{

//Converting int into Integer

int a=20;

Integer i=Integer.valueOf(a);//converting int into Integer

Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);

}

}

**INTERFACES**:- An interface is basically a kind of class. Like classes, interfaces contain methods and variables but with a major differences. The difference is that interfaces define only abstract methods and final fields. This means that interfaces do not specify any code to implement these methods and data fields contain only constants.

OR

**Interface:**

An interface is a collection of method names, without definitions, that can be added to classes to provide additional behavior not included with those methods the class defined itself or inherited from its superclasses.

interface In

{

void show ();

}

class P1 implements In

{

public void show ()

{

System.out.println("Hello Inter1");

}

}

class P2 implements In

{

public void show()

{

System.out.println("Hello Inter2");

}

}

class j11

{

public static void main (String args[])

{

P1 obj=new P1();

P2 obj1=new P2();

In obj3;

obj3= obj;

obj3.show();

obj3=obj1;

obj3.show();

}

}

**Packages: Putting Classes Together**:- Packages are java's way of grouping a variety of classes and/or interfaces together. By organizing our classes into packages we achieve the following benefits:-

1. The classes contained ijn the packages of other programs can be easily reused.

2. In packages, classes can be unique compared with classes in other packages. That is, two classes in two different packages can have the same name. They may be reffered by their fully qualified name, comprising the package name and the class name.

3. Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.

4. Package also provide a way for separating "design" from "coding". First we can design classes and decide their relationships, and then we can implement the java code needed for the methods. It is possible to change the implementation of any method without affecting the rest of the design.

**Java System Packages and Their Classes**

package name contents

java.awt Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

java.applet Classes for creating and implementing applets.

**Creating Package:**

###### First thing to do:

Step1:

package pkg; /\* pkg is the name of directory created in bin\*/

public class package1

{

public void show()

{

System.out.println("Welcome to package");

}

}

Step2: save it as package1.java

Step3: Compile

Step4: Select java and class file from bin and cut it then paste it in pkg directory

**Second thing to do:**

**Step1:**

**i**mport pkg.\*;

class pckg

{

public static void main(String args[])

{

package1 obj=new package1();

obj.show();

}

}

**Step2:** Save file as pckg.java

**Step3:**  Compile and run

### Thread in java:

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.

**LIFE CYCLE OF A THREAD**:- During the life time of a thread, there has many states it can enter. They include:-

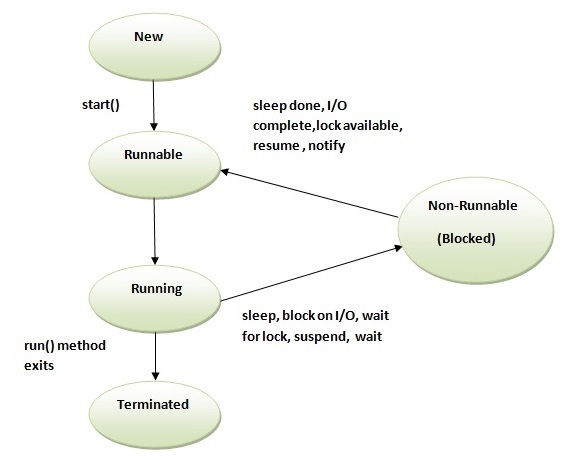
1. Newborn state

2. Runnable state

3. Running state

4. Blocked state

5. Dead state/Terminated



**Newborn State**:- When we create a thread object, the thread is born and is said to be in *newborn state.* At this state, we can do only one of the following things with it:-

* Schedule it for running using start( ) method.
* Kill it using stop( ) method.

**Runnable State**:- The *runnable* state means that the thread is ready for execution and is waiting for the availability of the processor.

**Running State**:- Running means that the processor has given its time to the thread for its execution. A running thread may relinquish its control in one of the following situations.

1. It has been suspended using **suspend( )** method. A suspended thread can be revived by using the **resume( )** method.

2. It has been made to sleep. We can put a thread to sleep for a specified time period using the method **sleep**(time) where *time* is in milliseconds.

3. It has been told to wait untill some event occurs. This is done using the **wait( )** method. The thread can be scheduled to run again using the **notify( )** method.

**Blocked State**:- A thread is said to be *blocked* when it is prevented from entering into the runnable state and subsequently the running state.

**Dead State**:- Every state has a life cycle. A running thread ends its life when it has completed executing its **run( )** method.

There are two ways to create a thread:

-By extending Thread class

-By implementing Runnable interface.

1.

class thd **extends Thread**

{

public void run()

{

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

}

public static void main(String args[])

{

thd obj=new thd();

obj.start();

}

}

2.

class thd **implements Runnable**

{

public void run()

{

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

}

public static void main(String args[])

{

thd obj=new thd();

Thread t=new Thread(obj);

t.start();

}

}

# **Thread Priority:**

|  |  |  |
| --- | --- | --- |
| Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. **3 constants defiend in Thread class:**  |  | | --- | | 1. public static int MIN\_PRIORITY 2. public static int NORM\_PRIORITY 3. public static int MAX\_PRIORITY |  |  | | --- | | Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10. |   **Eg:**  class ThdPr implements Runnable  {  public void run()  {  System.out.println("thread name"+Thread.currentThread().getName());  try  {  Thread.sleep(2000);  }  catch(InterruptedException e)  {  System.out.println(e);  }  Thread.yield();  System.out.println("Running thread priority is:"+Thread.currentThread().getPriority());    }  public static void main(String args[])  {  ThdPr m1=new ThdPr ();  ThdPr m2=new ThdPr ();  Thread t1=new Thread(m1,"MCR");  Thread t2=new Thread(m2,"BCA");  t1.setPriority(1);  t2.setPriority(10);  t1.start();  t2.start();    }  } **Daemon Thread in Java:** **Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.  There are many java daemon threads running automatically e.g. gc, finalizer etc.  **Java Garbage Collection**  In java, garbage means unreferenced objects.  Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.  To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management. 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. |

**Multithreaded Programming**:- Multithreading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (processes) which can be implemented at the same time in parallel.

**Program to demo Multithreading:**

class A extends Thread

{

public void run()

{

for(int i=1;i<=5;i++)

{

System.out.println("from thread A:i="+i);

}

System.out.println(" Exit from A");

}

}

class B extends Thread

{

public void run()

{

for(int j=1;j<=5;j++)

{

System.out.println("from thread B:j="+j);

}

System.out.println(" Exit from B");

}

}

class C extends Thread

{

public void run()

{

for(int k=1;k<=5;k++)

{

System.out.println("from thread C:k="+k);

}

System.out.println(" Exit from C");

}

}

class ThreadTest

{

public static void main(String args[])

{

new A().start();

new B().start();

new C().start();

}

}

**Difference between multithreading and multitasking**:-

**Multithreading**

1.It is a programming concept in which a program or a process is divided into ntwo or more subprogramsor threads that are executed at the same time in parallel.

2.It supports execution of multiple parts of a single program simultaneously.

3. The processor has to switch between different parts or threads of a program.

4. It is highly efficient.

5. A thread is the smallest unit in multithreading.

6. It helps in developing efficient programs.

7. It is cost-effective in case of context switching.

**Multitasking**

1. It is an operating system concept in which multiple tasks are performed simultaneously.

2. It supports execution of multiple programs simultaneously.

3. The processor has to switch between different programs or processes.

4. It is less efficient in comparision to multithreading .

5. A program or process is the smallest unit in a multiutasking environment.

6. It helps in developing efficient operating system.

7. It is expensive in case of context switching.

**SYNCHRONIZATION**:- One thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we may get strange results. Java enables us to overcome this problem using a technique known as *synchronization*.

**Program to Implement Synchronization:**

class Table

{

synchronized void printTable(int n) //synchronized method

{

for(int i=1;i<=5;i++)

{

System.out.println(n\*i);

try

{

Thread.sleep(400);

}

catch(Exception e)

{

System.out.println(e);

}

}

}

}

class MyThread1 extends Thread

{

Table t;

MyThread1(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(5);

}

}

class MyThread2 extends Thread

{

Table t;

MyThread2(Table t)

{

this.t=t;

}

public void run()

{

t.printTable(100);

}

}

public class Sync

{

public static void main(String args[])

{

Table obj = new Table();//only one object

MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}

}

**TYPES OF ERRORS**:- Errors may broadly be classified into two categories:-

* Compile-time errors
* Run-time errors

**Compile-Time Errors**:- All syntax errors will be detected and dispalyed by the java compiler and therefore these errors are known as compiule-time errors. Whenever the compiler dispalys an error, it will not create the **.class** file. It is therefore necessary that we fix all the errors before we can successfully compile and run the program.

*Illustration of compile-time errors*

/\* This program contains an error \*/

class Error1

{

public static void main (string args[ ] )

{

system.out.println ("Hello Java!") // Missing;

}

}

The most common problems are:-

* Missing semicolons
* Missing (or mismatch of) brackets in classes and methods
* Misspelling of identifiers and keywords
* Missing double quotes in strings
* Use of undeclared variables
* Incompatible types in assignments/ initialization

**Run-Time Errors**:- Sometimes, a program may compile successfully creating the **.class** file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. Most common run-time errors are:-

* Dividing an integer by zero
* Accessing an element that is out of the bounds of an array
* Trying to store a value into an array of an incompatible class or type
* Trying to cast an instance of a class to one of its subclasses
* Passing a parameter that is not in a valid range or value for a method
* Trying to illegally change the state of a thread

**EXCEPTIONS**:- An exception is a condition that is caused by a run-time error in the program. When the java interpreter encounters an error such as dividing an integer by zero, it creates an exception object and throws it. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as *exception handling*

### Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The Sun Microsystems says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

### 1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

### 2) Unchecked Exception

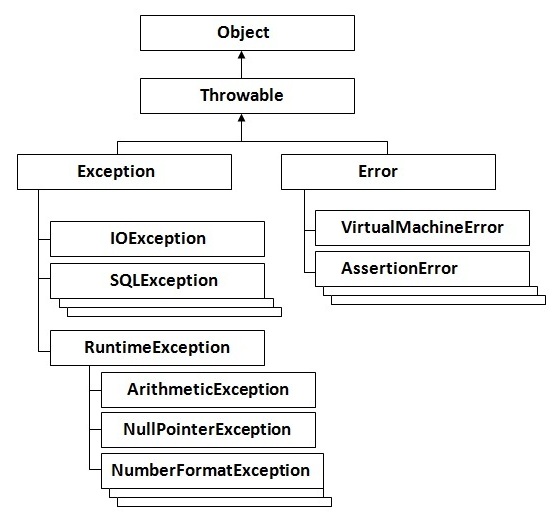
The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

### 3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc

There are 5 keywords used in java exception handling.

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



**try**: Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

**try**

**{**

**statement; //** generates an exception

**}**

**catch:** Java catch block is used to handle the Exception. It must be used after the try block only.

**catch (Exception-type** e**)**

**{**

statement; **//**  process the exception

**}**

.........................

.........................

**FINALLY STATEMENT:-** Java supports another statement known as **finally** statement that can be used to handle an exception that is not caught by any of the previous catch statements**, finally** block can be used to handle any exception generated within a try block.

**1.Program to demonstrate Exception Handling and Finally Statement:**

class excp

{

public static void main(String args[])

{

try

{

int i=10;

int j=0;

int k=i/j;

}

catch(ArithmeticException e)

{

System.out.println("Error1234: Contact vendor");

}

finally

{

System.out.println("End in Finally block ");

}

}

}

**2. Program to demonstrate Exception Handling using Throw, Throws, Throwable, try,catch,finally:**

import java.io.\*;

class exp4 extends Throwable

{

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

{

BufferedReader obj1=new BufferedReader(new InputStreamReader(System.in));

int z,i=5,j;

System.out.println("Enter value of j:");

j=Integer.parseInt(obj1.readLine());

ArithmeticException obj=new ArithmeticException();

try

{

if (j==0)

{

throw obj;

}

z=i/j;

System.out.print("Value="+z);

}

catch(ArithmeticException e)

{

System.out.print("You are trying to divide a number by zero!");

}

finally

{

System.out.println("\nWe are in Finally block");

}

}

}

## **throw** keyword: The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exceptions in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

throw exception;

Example of throw IOException.

throw new IOException("sorry device error);

# **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.

**Syntax**:

return\_type method\_name() throws exception\_class\_name

{

//method code

}

**Difference between throw and throws**

|  |  |  |
| --- | --- | --- |
| **No.** | **throw** | **throws** |
| 1) | Java throw keyword is used to explicitly throw an exception. | Java throws keyword is used to declare an exception. |
| 2) | Checked exception cannot be propagated using throw only. | Checked exception can be propagated with throws. |
| 3) | Throw is followed by an instance. | Throws is followed by class. |
| 4) | Throw is used within the method. | Throws is used with the method signature. |
| 5) | You cannot throw multiple exceptions. | You can declare multiple exceptions e.g. public void method()throws IOException,SQLException. |

**Difference between final ,finally and finalize**

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **final** | **finally** | **finalize** |
| 1) | Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. | Finally is used to place important code, it will be executed whether exception is handled or not. | Finalize is used to perform clean up processing just before object is garbage collected. |
| 2) | Final is a keyword. | Finally is a block. | Finalize is a method. |

**HOW APPLETS DIFFER FROM APPLICATIONS**:-

* Applets do not use the **main( )** method for initializing the execution of the code. Applets, when loaded, automatically call certain methods of Applet class to start and execute the applet code.
* Unlike stand-alone applications, applets cannot be run independently. They are run from inside a Web page using a special feature known as HTML tag.
* Applets cannot read from or write to the files in the local computer
* Applets cannot communicate with other servers on the network.
* Applets cannot run any program from the local computer
* Applets are restricted from using libraries from other languages such as C or C++. (Remember, Java language supports this feature through **native** methods. )

**APPLET :**An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are many advantages of applet. They are as follows:

* It works at client side so less response time.
* Secured
* It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

### Drawback of Applet

* Plugin is required at client browser to execute applet.

**APPLET LIFE CYCLE**:- Every java applet inherits a set of default behaviours from the **Applet** class. As a result, when an applet is loaded , it undergoes a series of changes in its state. The applet states include:-

* Born on initializing state
* Running state
* Idle state
* Dead or destroyed state

**Initializing State:-** Applet enters the*initialization* state when it is first loaded. This is achieved by calling the **init( )** method of **Applet** Class. At this stage, we may do the following, if required.

* Create objects needed by the applet
* Set up initial values
* Load images or fonts
* Set up colors

public void init ( )

{

..........................

.......................... (Action)

}

**Running State***:-*  Applet enters the ***running* state** when the system calls the **start( )** method of **Applet** Class. This occurs automatically after the applet is initialized. Starting can also occur if the applet is already in "stopped" (idle) state.

public voids start ( )

{

...........................

...........................

........................... (Action)

}

**Idle or Stopped State**:- An applet becomes *idle* when it is stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. We can also do by calling the **stop ( )** method explicity.

public void stop ( )

{

.........................

......................... (Action)

.........................

}

**Dead State**:- An applet is said to be *dead*  when it is removed from memory. This occurs automatically by invoking the **destroy ( )** method when we quit the browser.

public void destroy ( )

{

....................

.................... (Action)

....................

}

**Display State**:- Applet moves to the *display*  state whenever it has to perform some output operations on the screen. This happens immediately after the applet enters into the running state. The **paint( )** method is called to accomplish this task.

public void paint (Graphics g)

{

....................

.................... (Display statements)

....................

}

**Applet Programming:**

**.java file**

import java.awt.\*;

import java.applet.\*;

public class aplet extends java.applet.Applet

{

public void paint(Graphics obj)

{

obj.drawString("Welcome to Applet",200,200);

}

}

**.htm file**

<html>

<body bgcolor="red">

<Applet code="aplet.class" width=400 height=400>

</Applet>

</body>

</html>

**Passing Parameter to Applet using param in HTML**

**.java file**

import java.awt.\*;

import java.applet.\*;

public class param extends java.applet.Applet

{

String name;

public void init()

{

name=getParameter("**p**");

}

public void paint(Graphics obj)

{

obj.drawString(name,10,30);

}

}

**.htm file**

<html>

<applet code="param.class" width=500 height=500>

<param name=**p** value="Marwari College">

</applet>

</html>

**INTRODUCTION TO AWT PACKAGE**:- The Abstract Window Toolkit (AWT) package in java enables the programmers to create GUI-based applications. AWT provides support for both standard and applet windows.

**Component**:- Component class is the super class to all the other classes from which various GUI elements are realized. It is primarily responsible for effecting the display of a graphic object on the screen. It also handles the various keyboard and mouse events of the GUI application.

**Container**:- As the name suggests, the **Container** object contains the other awt components. It manages the layout and placement of the various awt components within the container. A container object can contain other containers objects as well; thus allowing nesting of containers.

**Window**:- The **Window** object realizes a top-level window but without any border or menu bar. It just specifies the layout of the window. A typical window that you would want to create in your application is not normally derived from the **Window** class but from its subclass, i.e., **Frame**.

**Panel**:- The super class of applet, **Panel** represents a window space on which the application's output’s displayed. It is just like a normal window having no border, title bar, menu bar, etc. A panel can contain within itself other panels as well.

**Frame**:- The **Frame** object realizes a top-level window complete with border and menu bar. It supports common window-related events such as close, open, activate, deactivate, etc. Almost all the programs that we created while discussing applets and graphics programming used one or more classes of the awt package.

###### Graphical Interface

import java.awt.\*;

import java.applet.Applet;

public class Buton extends Applet

{

Button b1 = new Button ("play");

Button b2 = new Button ("stop");

Checkbox c1=new Checkbox("prakash");

Choice c2=new Choice();

public void init()

{

add(b1);

add(b2);

add(c1);

c2.addItem("MBA");

add(c2);

}

public void paint(Graphics g)

{

g.drawString("My name is Prakash",100,100);

g.drawOval(195,100,90,55);

g.setColor(Color.YELLOW);

g.fillOval(200,20,30,45);

}

}

Buton.htm

<html>

<bodybgcolor="red">

<Applet code="Buton.class" width=400 height=400>

</Applet>

</body>

</html>

##### Passing Parameters to Applets

With Java applications, you pass parameters to your main() routine by using arguments on the command line.

Applet parameters come in two parts:

A parameter name, which is simply a name you pick, and a value, which is the actual value of that particular parameter.

In the HTML file that contains the embedded applet, you indicate each parameter using the <PARAM> tag, which has two attributes for the name and the value, called (surprisingly enough) NAME and VALUE. The <PARAM> tag goes inside the opening

and closing <APPLET> tags:

<APPLET CODE="MyApplet.class" WIDTH=100 HEIGHT=100>

<PARAM NAME=font VALUE="TimesRoman">

<PARAM NAME=size VALUE="36">

A Java applet appears here.</APPLET>

Parameters are passed to your applet when it is loaded. In the init() method for your applet, you can then get hold of those parameters by using the getParameter() method. getParameter() takes one argument-a string representing the name of the parameter you're looking for-and returns a string containing the corresponding value of that parameter.

To get the value of the font parameter from the HTML file, you might have a line such as this in your init() method:

String theFontName = getParameter("font");

Note

The names of the parameters as specified in <PARAM> and the names of the parameters in

getParameter() must match identically, including having the same case. In other words,

<PARAM NAME="name"> is different from <PARAM NAME="Name">. If your parameters are not being properly passed to your applet, make sure the parameter cases match.

import java.awt.Graphics;

import java.awt.Font;

import java.awt.Color;

public class MoreHelloApplet extends java.applet.Applet

{

Font f = new Font("TimesRoman", Font.BOLD, 36);

public void paint(Graphics g)

{

g.setFont(f);

g.setColor(Color.red);

g.drawString("Hello Again!", 5, 40);

}

}

import java.awt.Graphics;

import java.awt.Font;

import java.awt.Color;

public class MoreHelloApplet extends java.applet.Applet

{

Font f = new Font("TimesRoman", Font.BOLD, 36);

String name;

public void init()

{

name = getParameter("name");

if (name == null)

name = "ABC";

name = "Hello " + name + "!";

}

public void paint(Graphics g)

{

g.setFont(f);

g.setColor(Color.red);

g.drawString(name, 5, 40);

}

}

<HTML>

<HEAD>

<TITLE>Hello!</TITLE>

</HEAD>

<BODY>

<P>

<APPLET CODE="MoreHelloApplet.class" WIDTH=200 HEIGHT=50>

<PARAM NAME=name VALUE="Bonzo">

Hello to whoever you are!

</APPLET>

</BODY>

</HTML>

**Input Stream Methods**:-

***Method*** ***Description***

1. read( ) Reads a byte from the input stream

2. read (byte b[ ]) Reads an array of bytes into b

3. read (byte b[ ], int n, int m) Reads m bytes into b starting from nth byte

4. close Close the input stream

**Output Stream Methods**:-

***Method*** ***Description***

1. write ( ) Writes a byte to the output stream

2. write (byte b[ ]) Writes all bytes in the array b to the output stream

3. write (byte b[ ], int n, int m) Writes m bytes from array b starting from nth byte

4. close( ) Closes the output stream

**Copying characters**:-

// Copying characters from one file into another

import java**.**io**.**\***;**

class CopyCharacters

{

public static void main (string args [ ] )

{

// Declare and create input and output files

File inFile = new File ("input**.**dat")**;**

File outFile = new File ("output**.**dat")**;**

FileReader ins = null**;** // Creates file stream ins

FileWriter outs = null**;** // Creates file stream outs

try

{

ins = new FileReader (inFile)**;** // Opens inFile

outs = new FileWriter (outFile)**;** // Opens outFile

// Read and write till the end

int ch**;**

while ( (ch = ins**.**read ( ) ) ! = -1)

{

outs**.**write (ch)**;**

}

}

catch (IOException e)

{

System**.**out**.**println (e) **;**

System**.**exit (-1) **;**

}

finally // Close files

{

try

{

ins.close ( )**;**

outs**.**close ( )**;**

}

catch (IOExecution e) { }}}}