java:

============================================

james gosling is developed by java programming language in the year of 1991.

it is object oriented language and it support all object oriented programming concepts.

on those days java is called as "oak" after it is renamed as java program in the year of 1995.

-----------java latest version is java 14-----------

jdk-->jre-->jvm

java development kit contains : jre+development tools(java, javac etc)

java runtime environment contains: jvm+ library classes(rt.jar)

java virtual machine contains: all functions (class loader+ memory management+execute engine)

versions of java:

1. j2se 1.0(1996)

2. j2se 1.1

3. j2se 1.2

4. j2se 1.3

5. j2se 1.4

6. jse 5.0 (from 2004)

7. jse 6.0(from 2006)

8.

9.

10.

11.jse 12.0(from 2019)

12.JSE 13.0 Sep(2019)

13.JSE 14.0 latest version

java is object oriented programming language. it support all object oriented programming concepts.

By using java we develop 4 types of applications those are

1. Standalone application or desktop application(calculator app,notepad++, eclipse editor,

anti virus s/w applications): these applications can be developed in his/her own system .

in this internet is not require

---------Core Java is required(JSE)--------

2. Web Applications : web applications are like facebook, gmail etc. internet connection is required.

(servlets, jsp,spring,structs,hybernate,jsf,EJB etc)

-------------jee is required---------------------

3.enterprise applications: it is also known as distributed applications .

enterprise applications are like irctc, amazon, ebay, banking related apllications etc.

--------------jee is required--------------------

Ex: irctc, icici bank , amazon etc

4. Mobile Applications: Micro edition(andriod ) ---mobile applications

------------jme is required-----------------------

simple java program:

-------------------------------

public class Sample

{

public static void main(String args[])

{

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

}

}

to compile java program in command prompt to write:

---------------------------------------------------

javac Sample.java ---> to generate bytecode

Sample.class

to run java program in command prompt to write:

-----------------------------------------------

java Sample---> to execute file

command Line Arguments:

================================

public class Test

{

public static void main(String args[])

{

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

{

System.out.println(args[i]);

}

}

}

d:\> javac Test.java

d:\>java Test 10 20 30

JVM: java virtual machine

1. class loader

2. memory management

3. Execution engine

--------------class Loader:(loading, linking, initialization)----------

1. loading:

----------

Bootstrap loader: rt.jar file is loaded

extension class loader: c:\ajay\java\ extension related file load

Application class loader: it loads the classpath: with related file

2.linking:

----------

verify(byte code), processing(to assign code), rectify

3.initialization:

---------------

to initialize the class data

2. Memory Management:(method area or class area, heap, stack, pc register,

native method stack)

---------------------

method area: it loads all class related file\

heap: object related information or string object

stack: it creates threads

native-method stack

3. execution engine:

--------------------------------------

garbage collector, JIT compiler etc..

public class Test

{

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

{

Test t1 = new Test();

Test t2 = new Test();

// Nullifying the reference variable

t1 = null;

// requesting JVM for running Garbage Collector

System.gc();

// Nullifying the reference variable

t2 = null;

// requesting JVM for running Garbage Collector

Runtime.getRuntime().gc();

}

@Override

// finalize method is called on object once

// before garbage collecting it

protected void finalize() throws Throwable

{

System.out.println("Garbage collector called");

System.out.println("Object garbage collected : " + this);

}

}

class in java:

====================

class is a combination of datamembers and methods.

in the following example name and eid are the data members and student and tostring are the methods.

class Student

{

private String name;

private int eid;

public Student(String name, int eid) {

super();

this.name = name;

this.eid = eid;

}

@Override

public String toString() {

return "Student [name=" + name + ", eid=" + eid + "]";

}

}

Object :

=======================

object is an instance of a class, it invoke methods and initilizes the data members.

java support th following built-in objects . They are

1. toString()

2. hashcode()

3. equals()

4. clone()

5. wait()

6. notify()

7. notifyAll()

equals() method:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.test;

public class ObjectDemo1

{

public static void main(String[] args) {

Student s1=new Student("ajay",1);

Student s2=s1;

System.out.println(s1);

System.out.println(s2);

System.out.println(s1.equals(s2));

}

}

class Student

{

private String name;

private int eid;

public Student(String name, int eid) {

super();

this.name = name;

this.eid = eid;

}

@Override

public String toString() {

return "Student [name=" + name + ", eid=" + eid + "]";

}

}

in the above example s1 is an instance and s2 is a reference.

so in that case equals method returns true .

it shows the given output:

Student [name=ajay, eid=1]

Student [name=ajay, eid=1]

true

object:

======

In the following example object method toString()

Student.java:

================

package com.test;

public class Student

{

private String name;

private int id;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

@Override

public String toString() {

return "Student [name=" + name + ", id=" + id + "]";

}

}

package com.test;

public class Demo

{

public static void main(String[] args) {

Student s=new Student();

s.setId(100);

s.setName("ajay");

System.out.println(s);

System.out.println(s.toString());

}}

output:

===========================

Student [name=ajay, id=100]

Student [name=ajay, id=100]

Without toString method in program:

================================================

Student.java:

----------

package com.test;

public class Student

{

private String name;

private int id;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

}

Demo.java

======================================

package com.test;

public class Demo

{

public static void main(String[] args) {

Student s=new Student();

s.setId(100);

s.setName("ajay");

System.out.println(s);

System.out.println(s.toString());

}}

output:

=================

com.test.Student@15db9742

com.test.Student@15db9742

toString() method returns the object code .................

because without any toString()

method it return hexadecimal values.

2) hashCode() method in java

===============================

in the following example it return different hashcode

for different object , it return diff integer related values

package com.test;

public class Demo

{

public static void main(String[] args) {

Student s=new Student();

Student s1=new Student();

s.setId(100);

s.setName("ajay");

System.out.println(s);

System.out.println(s.hashCode());

System.out.println(s1.hashCode());

}

}

output:

=============

com.test.Student@15db9742

366712642

1829164700

3) object equals() method

=================================

It compares both object hashcodes, if that hashcode is

same it return true otherwise it returns false.

the given example output is false.

package com.test;

public class Demo

{

public static void main(String[] args)

{

Student s=new Student();

Student s1=new Student();

System.out.println(s.hashCode());

System.out.println(s1.hashCode());

System.out.println(s.equals(s1));

}

}

output:

===========

366712642

1829164700

false

example2

=============

package com.test;

public class Demo

{

public static void main(String[] args)

{

Student s=new Student();

System.out.println(s.equals(s));

}

}

output:

======

true

Data Abstraction and Encapsulation:

========================================

Encapsulation: Binding (hiding) code(methods) and data is known as Encapsulation

Abstraction: hiding data and showing functionality is known as Abstraction

package com.test;

public class Sample

{

private int id;

private String name;

private String address;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

public void display()

{

System.out.println(id+" "+name+" "+address);

}

}

package com.test;

public class Test {

public static void main(String[] args) {

Sample s=new Sample();

s.setId(100);

s.setName("vijay");

s.setAddress("uppal");

s.display();

}}

In the above example Student is a class follows encapsulation(data&code hiding)

and it follows abstaction , it showing fucntionality it means it show display() method

but it internally it hides with data.

Variables in java:

===============================

three types of variables in java. they are

1. local variable

2. instance variable

3. static variable

1. local variable: local varible declared inside of the method:

package com.variable;

class Variable

{

void hello()

{

int a=10;// here 'a' is a local variable

System.out.println(a);

}

public static void main(String[] args)

{

Variable v=new Variable();

v.hello();

}

}

2. instance variable or gloable variable

----------------------------------------

instance varible is a variable declared in side of the class but not inside of the method.

package com.variable;

class Variable

{

int a=10;// here is 'a' is instance variable

void hello()

{

System.out.println(a);

}

public static void main(String[] args)

{

Variable v=new Variable();

v.hello();

}

}

3. static variable:

------------------------

static variabele is a variable it declared as a static , once it declared that variable

memory is allocated once. once it is updated variable value , then it stores updated value

only but not old value.

package com.variable;

class VariableDemo

{

static int count=0;// static varible

public void increment()

{

count++;

}

public static void main(String args[])

{

VariableDemo obj1=new VariableDemo();

VariableDemo obj2=new VariableDemo();

obj1.increment();

System.out.println("Obj1: count is="+obj1.count);

obj2.increment();

System.out.println("Obj2: count is="+obj2.count);

}

}

output: 1

2

Example2:

class Employee

{

int empid;

String name;

static String company ="Virtusa";

Employee(int empid, String name)

{

empid = r;

name = n;

}

void display ()

{

System.out.println(emppid+" "+name+" "+company);

}

}

public class TestStaticVariable1

{

public static void main(String args[])

{

Employee s1 = new Employee(111,"Karan");

Employee s2 = new Employee(222,"Aryan");

s1.display();

s2.display();

}

}

Datatypes in java:

---------------------

Data types specify the different sizes and values that can be stored in the variable.

There are two types of data types in Java:

Primitive data types: The primitive data types include boolean, char, byte,

byte, short, int, long, float and double.

Non-primitive data types: The non-primitive data types include Classes,

string, interfaces, and Arrays.

\*\*primitive data type:

1. boolean->boolean(true or false)

2. numeric->character->char 2 bytes

->integral->integer->byte(1 byte),short(2),int(4),long(8)

-> float-->float(4), double(8)

\*\*\*Non-primitive data type:String, array etc

Byte->short->int->long->float->double->

Typeconversion: type conversion is two types. they are

a) Narrow conversion OR implicit type casting

b)Widening conversion OR explicit type casting

a) Narrow conversion: conversion from maximum datatytpe into smaller datatype is known as a Narrow conversion

Ex: long double->double->float->long->int->short->byte

class Simple

{

public static void main(String[] args)

{

float f=10.5f;

//int a=f;//Compile time error

int a=(int)f;

System.out.println(f);

System.out.println(a);

}}

b)Widening conversion: conversion from min to maximum is known as Widening

Ex: long double->double->float->long->int->short->byte.

class Simple

{

public static void main(String[] args)

{

int a=10;

float f=a;

System.out.println(a);

System.out.println(f);

}

}

comments in java:

=============================

1. single line comment

2. multiline comment

3. documentation

1. single line: //this is single line comment

2. multiline comment: /\* this is

multiline

comment \*/

operator:

=======================

it is a symbol to perform action between the operands . they are like

1. arithemetic:- +,-,\*,/,%

2. logical:- && ,||,!

3. assignment: =, +=, -=,\*=,/=,%=

4. relational: <,>,<=,>=,==,!=

5. increment/decrement: ++,--

6. conditional or ternary: ?:

7. bitwise: &,|,<<,>>

1. arithemetic operator:

package com.variable;

public class Operatordemo

{

public static void main(String args[])

{

int a=10,b=5;

System.out.println(a+b);

System.out.println(a-b);

System.out.println(a\*b);

System.out.println(a/b);

System.out.println(a%b);

}

}

6. conditional operator:

------------------------

package com.variable;

public class Operatordemo

{

public static void main(String args[])

{

int a=10,b=20;

int max=(a>b)?a:b;

System.out.println(max);

}

}

control statements:

======================

1. if

2. if-else

3.nested-if

4. else-if

5.switch

6.for

7.while

8.do-while

package com.test;

import java.util.ArrayList;

class MyThread

{

public static void main(String[] args) {

ArrayList<Integer> l=new ArrayList<Integer>();

l.add(10);

l.add(20);

l.add(30);

/\*

\* for(int a:l) { System.out.println(a); }//enhanced for loop

\*/

l.forEach(a->System.out.println(a));//foreach method

}

}

Object oriented programming concepts:

====================================

1.Object

------

Java Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

2.Class

======

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

3. Inheritance

============

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

4. Polymorphism in Java

=====================

Polymorphism

If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

5.Abstraction

=============

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

6.Encapsulation

===============

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

objects and class:

====================

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

================================================

In this example, we have created a Student class which has two data members id and name.

We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

--------------------

//Java Program to illustrate how to define a class and fields

//Defining a Student class.

class Student

{

//defining fields

int id;//field or data member or instance variable

String name;

//creating main method inside the Student class

public static void main(String args[])

{

//Creating an object or instance

Student s1=new Student();//creating an object of Student

//Printing values of the object

System.out.println(s1.id);//accessing member through reference variable

System.out.println(s1.name);

}

}

Output:

0

null

Object and Class Example: main outside the class

================================================

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

class Student

{

int id;

String name;

}

class TestStudent1

{

public static void main(String args[])

{

Student s1=new Student();

System.out.println(s1.id);

System.out.println(s1.name);

}

}

Output:

0

null

------------------->3 Ways to initialize object

==========================================================================

There are 3 ways to initialize object in Java.

I.By reference variable

II.By method

III. By constructor

1) Object and Class Example: Initialization through reference

============================================================================

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

class Student

{

int id;

String name;

}

class TestStudent2

{

public static void main(String args[])

{

Student s1=new Student();

s1.id=101;

s1.name="vijay";

System.out.println(s1.id+" "+s1.name);//printing members with a white space

}

}

Output:

101 vijay

We can also create multiple objects and store information in it through reference variable.

File: TestStudent3.java

=========================================================================

class Student

{

int id;

String name;

}

class TestStudent3

{

public static void main(String args[])

{

//Creating objects

Student s1=new Student();

Student s2=new Student();

//Initializing objects

s1.id=101;

s1.name="rahul";

s2.id=102;

s2.name="Amit";

//Printing data

System.out.println(s1.id +" "+ s1.name);

System.out.println(s2.id +" "+ s2.name);

}

}

Output:

101 rahul

102 Amit

2) Object and Class Example: Initialization through method

===========================================================

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method.

Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

File: TestStudent4.java

class Student

{

int rollno;

String name;

void insertRecord(int r, String n)

{

rollno=r;

name=n;

}

void displayInformation()

{

System.out.println(rollno+" "+name);

}

}

class TestStudent4

{

public static void main(String args[])

{

Student s1=new Student();

Student s2=new Student();

s1.insertRecord(111,"Karan");

s2.insertRecord(222,"Aryan");

s1.displayInformation();

s2.displayInformation();

}

}

Output:

111 Karan

222 Aryan

As you can see in the above figure, object gets the memory in heap memory area.

The reference variable refers to the object allocated in the heap memory area.

Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through a constructor

==================================================================

class Student

{

int id;

String name;

//creating a parameterized constructor

Student(int i,String n)

{

id = i;

name = n;

}

void display()

{

System.out.println(id+" "+name);

}

public static void main(String args[])

{

//creating objects and passing values

Student s1 = new Student(111,"Karan");

Student s2 = new Student(222,"Aryan");

//calling method to display the values of object

s1.display();

s2.display();

}

}

constructor:

================================

Constructors in Java

Types of constructors

1. Default Constructor

2. Parameterized Constructor

In Java, a constructor is a block of codes similar to the method.

It is called when an instance of the class is created. At the time of calling constructor,

memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class.

In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java:

--------------------------------------------

no-arg constructor(default constructor), and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation.

It is not necessary to write a constructor for a class.

It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

-->Constructor name must be the same as its class name

-->A Constructor must have no explicit return type

--->A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use access modifiers while declaring a constructor. It controls the object creation.

In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

class <class\_name>()

{

}

Example of default constructor

-------------------------------

In this example, we are creating the no-arg constructor in the Bike class.

It will be invoked at the time of object creation.

class Bike1

{

Bike1()

{

System.out.println("Bike is created");

}

public static void main(String args[])

{

Bike1 b=new Bike1();

}

}

Output:

Bike is created

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.

Java default constructor

Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0,

null, etc., depending on the type.

Example of default constructor that displays the default values

//Let us see another example of default constructor

//which displays the default values

class Student3

{

int id;

String name;

void display()

{

System.out.println(id+" "+name);

}

public static void main(String args[])

{

//creating objects

Student3 s1=new Student3();

Student3 s2=new Student3();

//displaying values of the object

s1.display();

s2.display();

}

}

Output:

0 null

0 null

Explanation:In the above class,you are not creating any constructor so compiler provides

you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

---------------------------------

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects.

However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters.

We can have any number of parameters in the constructor.

//Java Program to demonstrate the use of the parameterized constructor.

class Student4

{

int id;

String name;

Student4(int i,String n)

{

id = i;

name = n;

}

void display()

{

System.out.println(id+" "+name);

}

public static void main(String args[])

{

//creating objects and passing values

Student4 s1 = new Student4(111,"Karan");

Student4 s2 = new Student4(222,"Aryan");

//calling method to display the values of object

s1.display();

s2.display();

}

}

+++++++=+++++++++++++++++++====================++++++++++++++++++++++++++++

java keywords:

=================================

List of all Java Keywords

Keywords or Reserved words are the words in a language that are used for

some internal process or represent some predefined actions. These words are therefore

not allowed to use as a variable names or objects. Doing this will result into a compile time error.

Java also contains a list of reserved words or keywords. These are:

1)abstract -Specifies that a class or method will be implemented later, in a subclass

2)assert -Assert describes a predicate (a truefalse statement) placed in a Java program to

indicate that the developer thinks that the predicate is always true at that place.

If an assertion evaluates to false at run-time, an assertion failure results,

which typically causes execution to abort.

3)boolean A data type that can hold True and False values only

4)break A control statement for breaking out of loops

5)byte A data type that can hold 8-bit data values

6)case Used in switch statements to mark blocks of text

7)catch Catches exceptions generated by try statements

8)char A data type that can hold unsigned 16-bit Unicode characters

9)class -Declares a new class

10)continue -Sends control back outside a loop

11)default -Specifies the default block of code in a switch statement

12)do -Starts a do-while loop

13)double A data type that can hold 64-bit floating-point numbers

14)else Indicates alternative branches in an if statement

15)enum A Java keyword used to declare an enumerated type. Enumerations extend the base class.

16)extends -Indicates that a class is derived from another class or interface

17)final -Indicates that a variable holds a constant value or that a method will not be overridden

18)finally -Indicates a block of code in a try-catch structure that will always be executed

19)float -A data type that holds a 32-bit floating-point number

20)for -Used to start a for loop

21)if -Tests a true/false expression and branches accordingly

22)implements -Specifies that a class implements an interface

23)import -References other classes

24)instanceof -Indicates whether an object is an instance of a specific class or implements an interface

25)int A data type that can hold a 32-bit signed integer

26)interface Declares an interface

27)long A data type that holds a 64-bit integer

28)native -Specifies that a method is implemented with native (platform-specific) code

29)new Creates new objects

30)null -Indicates that a reference does not refer to anything

31)package Declares a Java package

32)private -An access specifier indicating that a method or variable may be accessed

only in the class its declared in

33)protected An access specifier indicating that a method or variable may only

be accessed in the class its declared in (or a subclass of the class its

declared in or other classes in the same package)

34)public An access specifier used for classes, interfaces, methods, and variables

indicating that an item is accessible throughout the application (or where the

class that defines it is accessible)

35)return -Sends control and possibly a return value back from a called method

36)short A data type that can hold a 16-bit integer

37)static -Indicates that a variable or method is a class method

(rather than being limited to one particular object)

38)strictfp A Java keyword used to restrict the precision and rounding of floating

point calculations to ensure portability.

39)super Refers to a classs base class (used in a method or class constructor)

40)switch -A statement that executes code based on a test value

41)synchronized -Specifies critical sections or methods in multithreaded code

42)this -Refers to the current object in a method or constructor

43)throw Creates an exception

44)throws -Indicates what exceptions may be thrown by a method

45)transient -Specifies that a variable is not part of an objects persistent state

46)try -Starts a block of code that will be tested for exceptions

47)void -Specifies that a method does not have a return value

48)volatile -Indicates that a variable may change asynchronously

49)while -Starts a while loop

static keyword in java:

=========================================================

The static can be:

-->Variable (also known as a class variable)

-->Method (also known as a class method)

--->Block

-------------------------Static in Java--------------------

1) Java static variable

If you declare any variable as static, it is known as a static variable.

The static variable can be used to refer to the common property of all objects

(which is not unique for each object), for example, the company name of employees,

college name of students, etc.

The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable:

It makes your program memory efficient (i.e., it saves memory).

Understanding the problem without static variable

class Student

{

int rollno;

String name;

String college="ITS";

}

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Java static property is shared to all objects.

Example of static variable

//Java Program to demonstrate the use of static variable

class Student

{

int rollno;

String name;

static String college ="ITS";

Student(int r, String n)

{

rollno = r;

name = n;

}

void display ()

{

System.out.println(rollno+" "+name+" "+college);

}

}

public class TestStaticVariable1

{

public static void main(String args[])

{

Student s1 = new Student(111,"Karan");

Student s2 = new Student(222,"Aryan");

s1.display();

s2.display();

}

}

Output:

111 Karan ITS

222 Aryan ITS

Static Variable

Program of counter by static variable

----------------------------------------------------------------------------

As we have mentioned above, static variable will get the memory only once, if any object

changes the value of the static variable, it will retain its value.

class Counter2

{

static int count=0;

Counter2()

{

count++;

System.out.println(count);

}

public static void main(String args[])

{

Counter2 c1=new Counter2();

Counter2 c2=new Counter2();

Counter2 c3=new Counter2();

}

}

Output:

1

2

3

2) Java static method

============================================================================

If you apply static keyword with any method, it is known as static method.

A static method belongs to the class rather than the object of a class.

A static method can be invoked without the need for creating an instance of a class.

A static method can access static data member and can change the value of it.

Example of static method

//Java Program to demonstrate the use of a static method.

class Student

{

int rollno;

String name;

static String college = "ITS";

static void change()

{

college = "BBDIT";

}

Student(int r, String n)

{

rollno = r;

name = n;

}

void display()

{

System.out.println(rollno+" "+name+" "+college);

}

public class TestStaticMethod

{

public static void main(String args[])

{

Student.change();

Student s1 = new Student(111,"Karan");

Student s2 = new Student(222,"Aryan");

Student s3 = new Student(333,"mahesh");

s1.display();

s2.display();

s3.display();

}

}

Test it Now

Output:111 Karan BBDIT

222 Aryan BBDIT

333 mahesh BBDIT

The static method can not use non static data member or call non-static method directly.

this and super cannot be used in static context.

class A

{

int a=40;//non static

static void display()

{

System.out.println(a);

}

public static void main(String args[])

{

A a1;

A.display();

}

}

Output:Compile Time Error

3) Java static block

======================================================================

Is used to initialize the static data member.

It is executed before the main method at the time of class loading.

Example of static block

class A2

{

static

{

System.out.println("static block is invoked");

}

public static void main(String args[])

{

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

}

}

this keyword in java

===================================

In java, this is a reference variable that refers to the current object.

java 'this' keyword

Usage of java this keyword

-->'this' can be used to refer current class instance variable.

-->'this' can be used to invoke current class method (implicitly)

-->'this' can be used to invoke current class constructor.

Usage of Java this keyword

1) this: to refer current class instance variable

----------------------------------------------------

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

class Student

{

int rollno;

String name;

float fee;

Student(int rollno,String name,float fee)

{

rollno=rollno;

name=name;

fee=fee;

}

void display()

{

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

}

class TestThis1

{

public static void main(String args[]){

Student s1=new Student(111,"ankit",5000f);

Student s2=new Student(112,"sumit",6000f);

s1.display();

s2.display();

}

}

Output:

0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same.

So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

class Student

{

int rollno;

String name;

float fee;

Student(int rollno,String name,float fee)

{

this.rollno=rollno;

this.name=name;

this.fee=fee;

}

void display()

{

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

}

}

class TestThis2

{

public static void main(String args[])

{

Student s1=new Student(111,"ankit",5000f);

Student s2=new Student(112,"sumit",6000f);

s1.display();

s2.display();

}

}

Test it Now

Output:

111 ankit 5000

112 sumit 6000

If local variables(formal arguments) and instance variables are different,

there is no need to use this keyword like in the following program:

Program where this keyword is not required

class Student

{

int rollno;

String name;

float fee;

Student(int r,String n,float f)

{

rollno=r;

name=n;

fee=f;

}

void display()

{

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

}

}

class TestThis3

{

public static void main(String args[])

{

Student s1=new Student(111,"ankit",5000f);

Student s2=new Student(112,"sumit",6000f);

s1.display();

s2.display();

}

}

Test it Now

Output:

111 ankit 5000

112 sumit 6000

It is better approach to use meaningful names for variables. So we use same name for

instance variables and parameters in real time, and always use this keyword.

2) this: to invoke current class method

-----------------------------------------

You may invoke the method of the current class by using the this keyword.

If you don't use the this keyword, compiler automatically adds this keyword while

invoking the method. Let's see the example

this keyword:

class A

{

void m()

{

System.out.println("hello m");

}

void n()

{

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

//m( );

this.m();

}

}

class TestThis

{

public static void main(String args[])

{

A a=new A();

a.n();

}

}

Output:

hello n

hello m

3) this( ) : to invoke current class constructor

------------------------------------------------

The this( ) constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

Calling default constructor from parameterized constructor:

class A

{

A( )

{

System.out.println("hello a");

}

A(int x)

{

this( );

System.out.println(x);

}

}

class TestThis5

{

public static void main(String args[])

{

A a=new A(10);

}

}

Output:

hello a

10

-------------------------------------------------------------------------

Relationships :

===============

Relationship between two classes is known as inheritance.

for using inheritance we achive the following concepts:

-->Reducing redundency

-->reusability concept

Relationships are mainly two types: they are

1) inheritance

2) Association

1) inheritance(Is-A): Strong bonding between the two classes is known

as Inheritance. it is tightly coupled program.

Using "extends" keyword to acheive this relations ship.

class Superclassname

{

---

---------

}

class Subclass extends Superclass

{

---

---

}

2) Association(Has-A): it is loosely coupled program. without using any extends keyword.

Association is again classified into two types

class Superclassname

{

----

----

}

class Subclassname

{

Superclass objcet=new Superclassname;

----

-----

}

a) Aggregation(weak bonding) ; Example Car class and Music player class

withoout car class alsoMusicplayer class is exist

b) composition(strong bonding): Example car class and Engine class

Without car class Engine class doesn't exist

1)Inheritance: and types

===========================================================================

a. single level

b.multilevel

c.multiple inheritance

d.hie-rarchical

e. hybrid

a. single level inheritance: In single level

==========================================================

package com.Inheritance;

class Super//parent, base, super

{

void display()

{

System.out.println("super class");

}

}

class Sub extends Super

{

void display1()

{

System.out.println("sub class");

}

}

//Is-A relation ship between both classes

public class Single

{

public static void main(String args[])

{

Sub s=new Sub();

s.display1();

s.display();

}

}

b. multilevel - inheritance:

===========================================================

package com.Inheritance;

class Parent

{

void display()

{

System.out.println("parent class");

}

}

class child extends Parent

{

void display1()

{

System.out.println("child class");

}

}

class childchild extends child

{

void display2()

{

System.out.println("child child class");

}

}

public class Multilevel

{

public static void main(String args[])

{

childchild c=new childchild();

c.display2();

c.display1();

c.display();

}

}

c. multiple -inheritance:

====================================================

package com.Inheritance;

class Base

{

void display()

{

System.out.println("base class");

}

}

interface Base1

{

void display1( );

}

class Multiple extends Base implements Base1

{

public void display1()

{

System.out.println("Base1 calss");

}

void display2( )

{

System.out.println("derived class");

}

public static void main(String[] args)

{

Multiple d=new Multiple();

d.display();

d.display1();

d.display2();

}

}

d. Hie-rarchical inheritance:

=================================================

package com.Inheritance;

class Base11

{

void display()

{

System.out.println("base class");

}

}

class Derive1 extends Base11

{

void getdata()

{

System.out.println("derive class2 ");

}

}

class Derive2 extends Base11

{

void showdata()

{

System.out.println("Derive class 2");

}

}

public class Demo\_Hierarchical

{

public static void main(String args[])

{

Derive1 d1=new Derive1();

Derive2 d2=new Derive2();

d1.display();

d2.display();

d1.getdata();

d2.showdata();

}

}

===============================================================

class A

{

--

---

}

class B extends A//tightly coulped example inheritance

{

----

---

}

=============================================================

class A

{

----

---

}

class B

{

---

A a=new A( );//association and loosly coupled

---

}

Polymorphism :

============================================================

poly means many morphism means mechanisms

many forms, in java polymorphism are 2 types they are

1)Static polymorphism

2) Dynamic polymorphism

\*\*\*static polymorphism is also known as compile time polymorphism or early binding polymorphism

Ex: Method overloading

\*\*\*Dynamic polymorphism is also known as run time polymorphism or late binding polymorphism

Ex: Method Overriding

Method Overloading in Java

==================================================================

Different ways to overload the method

-->By changing the no. of arguments or paramters

-->By changing the datatype

If a class has multiple methods having same name but different in parameters,

it is known as Method Overloading.

Suppose you have to perform addition of the given numbers but there can be any number of arguments,

if you write the method such as a(int,int) for two parameters, and b(int,int,int) for

three parameters then it may be difficult for you as well as other programmers to understand

the behavior of the method because its name differs.

In java, Method Overloading is not possible by changing the return type of the method only.

1) Method Overloading: changing no. of arguments

========================================================================

In this example, we have created two methods, first add() method performs addition of

two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

class Adder

{

static int add(int a,int b)

{

return a+b;

}

static int add(int a,int b,int c)

{

return a+b+c;

}

}

class TestOverloading1

{

public static void main(String[] args)

{

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(11,11,11));

}

}

Output:

22

33

2) Method Overloading: changing data type of arguments

=================================================

In this example, we have created two methods that differs in data type.

The first add method receives two integer arguments and second add method receives two double arguments.

class Adder

{

static int add(int a, int b)

{

return a+b;

}

static double add(double a, double b)

{

return a+b;

}

}

class TestOverloading2

{

public static void main(String[] args)

{

System.out.println(Adder.add(11,11));

System.out.println(Adder.add(12.3,12.6));

}

}

Test it Now

Output:

22

24.9

Method Overriding in Java

===================================================================

If subclass (child class) has the same method as declared in the parent class,

known as method overriding in Java.

In other words, If a subclass provides the specific implementation of the method that has been declared

by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding:

---------------------------------------------------------

Method overriding is used to provide the specific implementation of a method which is

already provided by its superclass.

Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

-->The method must have the same name as in the parent class

-->The method must have the same parameter as in the parent class.

-->There must be an IS-A relationship (inheritance).

Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process

in which a call to an overridden method is resolved at runtime,

thats why it is called runtime polymorphism. I have already discussed method overriding

in detail in a separate tutorial, refer it: Method Overriding in Java.

Example

In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class.

The child class is overriding the method myMethod() of parent class.

In this example we have child class object assigned to the parent class reference

so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called (not the type of reference).

To understand the concept of overriding, you should have the basic knowledge of inheritance in Java.

class ABC{

public void myMethod(){

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

}

}

public class XYZ extends ABC{

public void myMethod(){

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

}

public static void main(String args[]){

ABC obj = new XYZ();//example of upcasting

obj.myMethod();

}

}

Output:

Overriding Method

When an overridden method is called through a reference of parent class,

then type of the object determines which method is to be executed.

Thus, this determination is made at run time.

Since both the classes, child class and parent class have the same method animalSound.

Which version of the method(child class or parent class) will be called is determined at runtime by JVM.

overriding static methods in java:

-------------------------------------

class Parent {

// Static method in base class

// which will be hidden in subclass

static void m1()

{

System.out.println("From parent "

+ "static m1()");

}

// Non-static method which will

// be overridden in derived class

void m2()

{

System.out.println("From parent "

+ "non-static(instance) m2()");

}

}

class Child extends Parent {

// This method hides m1() in Parent

static void m1()

{

System.out.println("From child static m1()");

}

// This method overrides m2() in Parent

@Override

public void m2()

{

System.out.println("From child "

+ "non-static(instance) m2()");

}

}

// Driver class

class Main {

public static void main(String[] args)

{

Parent obj1 = new Child();

// As per overriding rules this

// should call to class Child static

// overridden method. Since static

// method can not be overridden, it

// calls Parent's m1()

obj1.m1();

// Here overriding works

// and Child's m2() is called

obj1.m2();

}

}

super keyword in java:

======================================================

The super keyword in Java is a reference variable which is used to refer immediate parent class object.

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

Usage of Java Super keyword

---------------------------------------

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

We can use super keyword to access the data member or field of parent class.

It is used if parent class and child class have same fields.

class Animal

{

String color="white";

}

class Dog extends Animal

{

String color="black";

void printColor()

{

System.out.println(color);//black

System.out.println(super.color);//white

}

}

class TestSuper1

{

public static void main(String args[])

{

Dog d=new Dog();

d.printColor();

}}

Output:

black

white

In the above example, Animal and Dog both classes have a common property color.

If we print color property, it will print the color of current class by default.

To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method;

==============================================================================================

The super keyword can also be used to invoke parent class method.

It should be used if subclass contains the same method as parent class.

In other words, it is used if method is overridden.

class Base1

{

void display()

{

System.out.println("base1...");

}

}

class Derive extends Base1

{

void getdata()

{

System.out.println("derive method1...");

}

void hello()

{

System.out.println("derive method2...");

}

void show()

{

super.display();

getdata();

hello();

}

}

class TestSuper2

{

public static void main(String args[]){

Derive d=new Derive();

d.show();

}

}

To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

class Animal

{

Animal( )

{

System.out.println("animal is created");

}

}

class Dog extends Animal

{

Dog( )

{

super( );

System.out.println("dog is created");

}

}

class TestSuper3

{

public static void main(String args[])

{

Dog d=new Dog();

}

}

Note: super( ) is added in each class constructor automatically by compiler if there is no super( ) or this( ).

java super:

================================================================

As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Another example of super keyword where super() is provided by the compiler implicitly.

class Animal

{

Animal( )

{

System.out.println("animal is created");}

}

class Dog extends Animal

{

Dog( )

{

System.out.println("dog is created");

}

}

class TestSuper4

{

public static void main(String args[]){

Dog d=new Dog();

}

}

final Keyword In Java

======================================================

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

-->variable

-->method

-->class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

final keyword in java

======================================================================

1) Java final variable

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

Example of final variable

------------------------------

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

class Bike9

{

final int speedlimit=90;//final variable

void run()

{

speedlimit=400;

}

public static void main(String args[])

{

Bike9 obj=new Bike9();

obj.run();

}

}

Output:Compile Time Error

2) Java final method

============================================================================

If you make any method as final, you cannot override it.

Example of final method

class Bike

{

final void run()

{

System.out.println("running");}

}

class Honda extends Bike

{

void run()

{

System.out.println("running safely with 100kmph");

}

public static void main(String args[])

{

Honda honda= new Honda();

honda.run();

}

}

Error:

Output:Compile Time Error

3) Java final class

============================================================================

If you make any class as final, you cannot extend it.

Example of final class

final class Bike12

}

class Honda1 extends Bike12

{

void run()

{

System.out.println("running safely with 100kmph");

}

public static void main(String args[])

{

Honda1 honda= new Honda1();

honda.run();

}

}

Output:Compile Time Error

-----------Polymorphism in Java----------------

Polymorphism in Java is a concept by which we can perform a single action in different ways.

Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs"

means forms. So polymorphism means many forms.

There are two types of polymorphism in Java:

compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method

overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism.

Here, we will focus on runtime polymorphism in java.

Runtime Polymorphism in Java

-------------------------------

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.

Let's first understand the upcasting before Runtime Polymorphism.

--------------------------------------------------------------------

Upcasting:

===========

If the reference variable of Parent class, refers to the object of Child class,

it is known as "upcasting". For example:

Upcasting in Java

class A

{

}

class B extends A

{

}

A a=new B();//upcasting

For upcasting, we can use the reference variable of class type or an interface type. For Example:

interface I

{

}

class A

{

}

class B extends A implements I

{

}

Here, the relationship of B class would be:

B IS-A A

B IS-A I

B IS-A Object

Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Example of Java Runtime Polymorphism

-------------------------------------

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

class Bike

{

void run()

{

System.out.println("running");

}

}

class Splendor extends Bike

{

void run()

{

System.out.println("running safely with 60km");

}

public static void main(String args[])

{

Bike b = new Splendor();//upcasting

b.run();

}

}

Test it NowOutput:

running safely with 60km.

Static Binding and Dynamic Binding

---------------------------------------

Connecting a method call to the method body is known as binding.

There are two types of binding

Static Binding (also known as Early Binding).

Dynamic Binding (also known as Late Binding).

Static vs. Dynamic Binding in java Understanding Type

Let's understand the type of instance.

1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

int data=30;

Here data variable is a type of int.

2) References have a type

class sample

{

public static void main(String args[])

{

sample d1;//Here d1 is a type of sample

}

}

3) Objects have a type

An object is an instance of particular java class,but it is also an instance of its superclass.

class Animal

{

}

class Dog extends Animal

{

public static void main(String args[])

{

Dog d1=new Dog();

}

}

Here d1 is an instance of Dog class, but it is also an instance of Animal.

static binding:

------------------

When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

------- Example of static binding-------------------------

-----------------------------------------------------------------------------------

class sample

{

private void display()

{

System.out.println("static binding...");

}

public static void main(String args[])

{

sample d1=new sample();

d1.display();

}

}

----------Dynamic binding-----------------------------------------------------------

-----------------------------------------------------------------------------------------

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding:

class Test

{

void getRead()

{

System.out.println("super class method...");

}

}

class Test\_Sub extends Test

{

void getRead()

{

System.out.println("sub class nethod...");

System.out.println("implementation is possible in this method");

}

public static void main(String args[])

{

Test\_Sub a=new Test();

a.getRead();

}

}

Test it Now

Abstract class in Java

======================================================================

A class which is declared with the abstract keyword is known as an abstract class in Java.

It can have abstract and non-abstract methods (method with the body).

it is also known as concrete method(implemented method), non-concrete method(non-implemented method).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

-----------------------------------

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

to the user.

Another way, it shows only essential things to the user and hides the internal details,

for example, sending SMS where you type the text and send the message.

You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

Abstract class (0 to 100%)

Interface (100%)

Abstract class in Java

----------------------------------------------------

A class which is declared as abstract is known as an abstract class.

It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

->An abstract class must be declared with an abstract keyword.

->It can have abstract and non-abstract methods.

->It cannot be instantiated.

->It can have constructors and static methods also.

->It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

--------------------------------------------------------------------------------------------

abstract class A

{

}

Abstract Method in Java:

-----------------------

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

abstract void printStatus();//no method body and abstract

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run.

Its implementation is provided by the Honda class.

abstract class Bike

{

abstract void run();

}

class Honda4 extends Bike

{

void run()

{

System.out.println("running safely");

}

public static void main(String args[])

{

Bike obj = new Honda4();

obj.run();

}

}

Example2:

------------------------------------------------------

abstract class Bank

{

abstract int getRateOfInterest();

}

class SBI extends Bank

{

int getRateOfInterest()

{

return 7;

}

}

class PNB extends Bank

{

int getRateOfInterest()

{

return 8;

}

}

class TestBank

{

public static void main(String args[])

{

Bank b;

b=new SBI();

System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");

b=new PNB();

System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");

}

}

Abstract class having constructor, data member and methods

===============================================================================================

An abstract class can have a data member, abstract method, method body (non-abstract method),

constructor, and even main() method.

//Example of an abstract class that has abstract and non-abstract methods

abstract class Bike

{

Bike()

{

System.out.println("bike is created");

}

abstract void run();

void changeGear()

{

System.out.println("gear changed");

}

}

//Creating a Child class which inherits Abstract class

class Honda extends Bike

{

void run()

{

System.out.println("running safely..");

}

}

//Creating a Test class which calls abstract and non-abstract methods

class TestAbstraction2

{

public static void main(String args[])

{

Bike obj = new Honda();

obj.run();

obj.changeGear();

}

}

Another real scenario of abstract class:

================================================================================

The abstract class can also be used to provide some implementation of the interface.

In such case, the end user may not be forced to override all the methods of the interface.

interface A

{

void a();

void b();

void c();

void d();

}

abstract class B implements A

{

public void c()

{

System.out.println("I am c");

}

}

class M extends B

{

public void a()

{

System.out.println("I am a");

}

public void b()

{

System.out.println("I am b");

}

public void d()

{

System.out.println("I am d");

}

}

class Test5

{

public static void main(String args[])

{

A a=new M();

a.a();

a.b();

a.c();

a.d();

}

}

Interface in Java

=================================================================================================

Multiple inheritance by Interface:

An interface in java is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction.

There can be only abstract methods in the Java interface, not method body.

It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables.

It cannot have a method body.

Java Interface also represents the IS-A relationship.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have default and static methods in an interface.

Since Java 9, we can have private methods in an interface.

-----------------------------------------------------------------

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

It is used to achieve abstraction.

By interface, we can support the functionality of multiple inheritance.

It can be used to achieve loose coupling.

Why use Java Interface

How to declare an interface?

An interface is declared by using the interface keyword.

It provides total abstraction; means all the methods in an interface are declared

with the empty body, and all the fields are public, static and final by default.

A class that implements an interface must implement all the methods declared in the interface.

Syntax:

interface <interface\_name>

{

// declare constant fields

// declare methods that abstract

// by default.

}

Internal addition by the compiler

----------------------------------------------------------------------------

The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.

interface in java

===================

The relationship between classes and interfaces, a class extends another class, an interface extends another interface, but a class implements an interface.

The relationship between class and interface

------------------------------------------------------------------

Java Interface Example

In this example, the Printable interface has only one method, and its

implementation is provided in the A6 class.

interface printable

{

void print();

}

class A6 implements printable

{

public void print()

{

System.out.println("Hello");

}

public static void main(String args[])

{

A6 obj = new A6();

obj.print();

}

}

Hello

Java Interface Example: hello

In this example, the Drawable interface has only one method.

Its implementation is provided by Rectangle and Circle classes.

In a real scenario, an interface is defined by someone else, but its implementation is

provided by different implementation providers. Moreover, it is used by someone else.

The implementation part is hidden by the user who uses the interface.

File: TestInterface1.java

interface Drawable

{

void draw();

}

//Implementation: by second user

class Rectangle implements Drawable

{

public void draw()

{

System.out.println("drawing rectangle");

}

}

class Circle implements Drawable

{

public void draw()

{

System.out.println("drawing circle");

}

}

//Using interface: by third user

class TestInterface1

{

public static void main(String args[])

{

Drawable d=new Circle();

d.draw();

}

}

Output:

drawing circle

Java Interface Example: Bank

============================================================================================

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

interface Bank

{

float rateOfInterest();

}

class SBI implements Bank

{

public float rateOfInterest()

{

return 9.15f;

}

}

class PNB implements Bank

{

public float rateOfInterest()

{

return 9.7f;

}

}

class TestInterface2

{

public static void main(String[] args)

{

Bank b=new SBI();

System.out.println("ROI: "+b.rateOfInterest());

}

}

Output:

ROI: 9.15

Multiple inheritance in Java by interface

=========================================================================================

If a class implements multiple interfaces, or an interface extends multiple interfaces,

it is known as multiple inheritance.

multiple inheritance in java

interface Printable

{

void print();

}

interface Showable

{

void show();

}

class A7 implements Printable,Showable

{

public void print()

{

System.out.println("Hello");

}

public void show()

{

System.out.println("Welcome");

}

public static void main(String args[]){

A7 obj = new A7();

obj.print();

obj.show();

}

}

Output:Hello

Welcome

Q) Multiple inheritance is not supported through class in java,

but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported

in the case of class because of ambiguity. However, it is supported in case of an

interface because there is no ambiguity. It is because its implementation is provided by

the implementation class. For example:

interface Printable

{

void print();

}

interface Showable

{

void print();

}

class TestInterface3 implements Printable, Showable

{

public void print()

{

System.out.println("Hello");

}

public static void main(String args[])

{

TestInterface3 obj = new TestInterface3();

obj.print();

}

}

Test it Now

Output:

Hello

As you can see in the above example, Printable and Showable interface have

same methods but its implementation is provided by class TestTnterface1, so there is no ambiguity.

Interface inheritance

A class implements an interface, but one interface extends another interface.

interface Printable

{

void print();

}

interface Showable extends Printable

{

void show();

}

class TestInterface4 implements Showable

{

public void print()

{

System.out.println("Hello");

}

public void show()

{

System.out.println("Welcome");

}

public static void main(String args[])

{

TestInterface4 obj = new TestInterface4();

obj.print();

obj.show();

}

}

Test it Now

Output:

Hello

Wrapper Class:

==========

The wrapper class in Java provides the mechanism to convert primitive into object

and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature convert primitives

into objects and objects into primitives automatically.

The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Autoboxing: primitive values are converted to Object type is known as auto-boxing.

Ex: Integer i=10;

-->in the above example i value 10 is primitive type

-->'i' is a varible of Integer class.

package com.test;

public class DemoAuto

{

public static void main(String[] args) {

Integer i=10;

System.out.println(i);

}

}

Unboxing in java:

====================

conversion from object type into Integer (primitive)type.

package com.test;

public class DemoUnbox

{

public static void main(String[] args)

{

Integer i=new Integer(10);

int x=i;

System.out.println(x);

}

}

Java Default Method Example(java 8 feature)

==================================================================================================

In the following example, Sayable is a functional interface that contains a

default and an abstract method. The concept of default method is used to define a method

with default implementation. You can override default method also to provide more specific

implementation for the method.

Let's see a simple

interface Sayable

{

// Default method

default void say()

{

System.out.println("Hello, this is default method");

}

// Abstract method

void sayMore(String msg);

}

public class DefaultMethods implements Sayable

{

public void sayMore(String msg)

{ // implementing abstract method

System.out.println(msg);

}

public static void main(String[] args)

{

DefaultMethods dm = new DefaultMethods();

dm.say(); // calling default method

dm.sayMore("Work is worship"); // calling abstract method

}

}

Marker interface in Java:

=================================================================================================

It is an empty interface (no field or methods). Examples of marker interface are

Serializable,

Clonnable and

Remote interface.

All these interfaces are empty interfaces.

public interface Serializable

{

// nothing here

}

Examples of Marker Interface which are used in real-time applications :

--------------------------------------------------------------------------------------------

Cloneable interface : Cloneable interface is present in java.lang package.

There is a method clone( ) in Object class. A class that implements the Cloneable interface

indicates that it is legal for clone( ) method to make a field-for-field copy of instances of that class.

Invoking Objects clone method on an instance of the class that does not implement the Cloneable

interface results in an exception CloneNotSupportedException being thrown.

By convention, classes that implement this interface should override Object.clone() method.

by using to command prompt window to check that Cloneable interface is a marker interface or not.

like javap java.io.Seriallizable;

javap java.lang.Cloneable;

// Java program to illustrate Cloneable interface

import java.lang.Cloneable;

// By implementing Cloneable interface

// we make sure that instances of class A

// can be cloned.

class A implements Cloneable

{

int i;

String s;

// A class constructor

public A(int i,String s)

{

this.i = i;

this.s = s;

}

// Overriding clone() method

// by simply calling Object class

// clone() method.

protected Object clone()

throws CloneNotSupportedException

{

return super.clone();

}

}

public class Test

{

public static void main(String[] args)

throws CloneNotSupportedException

{

A a = new A(20, "virtusa");

// cloning 'a' and holding

// new cloned object reference in b

// down-casting as clone() return type is Object

A b = (A)a.clone();

System.out.println(b.i);

System.out.println(b.s);

}

}

Serializable interface :

====================================================================================================

Serializable interface is present in java.io package. It is used to make an object eligible

for saving its state into a file. This is called Serialization.

Classes that do not implement this interface will not have any of

their state serialized or deserialized.

All subtypes of a serializable class are themselves serializable.

// Java program to illustrate Serializable interface

package com.Marker;

import java.io.\*;

class A1 implements Serializable

{

int i;

String s;

public A1(int i,String s)

{

this.i = i;

this.s = s;

}

}

public class Demo\_Serializable1

{

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

{

A1 a = new A1(20,"Virtusa");

// Serializing 'a'

FileOutputStream fos = new FileOutputStream("E:\\xyz.txt");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(a);

FileInputStream fis = new FileInputStream("E:\\xyz.txt");

ObjectInputStream ois = new ObjectInputStream(fis);

A1 b = (A1)ois.readObject();//down-casting object

System.out.println(b.i+" "+b.s);

// closing streams

/\*

\* oos.close(); ois.close();

\*/

}

}

Output:

Remote interface : Remote interface is present in java.rmi package.

A remote object is an object which is stored at one machine and accessed from another machine.

So, to make an object a remote object, we need to flag it with Remote interface.

Here, Remote interface serves to identify interfaces whose methods

may be invoked from a non-local virtual machine.

Any object that is a remote object must directly or indirectly implement

this interface. RMI (Remote Method Invocation) provides some convenience

classes that remote object implementations can extend which facilitate remote object creation.

Java String

====================================================

In Java, string is basically an object that represents sequence of char values.

An array of characters works same as Java string. For example:

char[] ch={'j','a','v','a','t','p','o','i','n','t'};

String s=new String(ch);

is same as:

String s="javatpoint";

Java String class provides a lot of methods to perform operations

on string such as compare(), concat(), equals(), split(), length(),

replace(), compareTo(), intern(), substring() etc.

String in Java

=========================================

CharSequence Interface

The CharSequence interface is used to represent the sequence of characters.

String, StringBuffer and StringBuilder classes implement it.

It means, we can create strings in java by using these three classes.

We will discuss immutable string later. Let's first understand what is String in Java and

how to create the String object.

What is String in java

-------------------------

Generally, String is a sequence of characters. But in Java, string is an object that represents a

sequence of characters. The java.lang.String class is used to create a string object.

How to create a string object?

There are two ways to create String object:

By string literal

By new keyword

1) String Literal

-----------------------------

Java String literal is created by using double quotes. For Example:

String s="welcome";

Each time you create a string literal, the JVM checks the "string constant pool" first.

If the string already exists in the pool, a reference to the pooled instance is returned.

If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

For example:

String s1="Welcome";

String s2="Welcome";//It doesn't create a new instance

Java string literal

In the above example, only one object will be created. Firstly,

JVM will not find any string object with the value "Welcome" in string constant pool,

that is why it will create a new object. After that it will find the string with the value "Welcome"

in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword:

String s=new String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory,

and the literal "Welcome" will be placed in the string constant pool.

The variable s will refer to the object in a heap (non-pool).

Java String Example

public class StringExample

{

public static void main(String args[])

{

String s1="java";//creating string by java string literal

char ch[]={'s','t','r','i','n','g','s'};

String s2=new String(ch);//converting char array to string

String s3=new String("example");//creating java string by new keyword

System.out.println(s1);

System.out.println(s2);

System.out.println(s3);

}}

Test it Now

java

strings

example

Java StringBuffer class

===============================

Java StringBuffer class is used to create mutable (modifiable) string.

The StringBuffer class in java is

same as String class except it is mutable i.e. it can be changed.

Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously.

So it is safe and will result in an order.

Important Constructors of StringBuffer class

Constructor Description

StringBuffer() creates an empty string buffer with the initial capacity of 16.

StringBuffer(String str) creates a string buffer with the specified string.

StringBuffer(int capacity) creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class

Modifier and Type Method Description

-->public synchronized StringBuffer append(String s) is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.

-->public synchronized StringBuffer insert(int offset, String s) is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.

-->public synchronized StringBuffer replace(int startIndex, int endIndex, String str) is used to replace the string from specified startIndex and endIndex.

-->public synchronized StringBuffer delete(int startIndex, int endIndex) is used to delete the string from specified startIndex and endIndex.

public synchronized StringBuffer reverse() is used to reverse the string.

public int capacity() is used to return the current capacity.

public void ensureCapacity(int minimumCapacity) is used to ensure the capacity at least equal to the given minimum.

public char charAt(int index) is used to return the character at the specified position.

public int length() is used to return the length of the string i.e. total number of characters.

public String substring(int beginIndex) is used to return the substring from the specified beginIndex.

public String substring(int beginIndex, int endIndex) is used to return the substring from the specified beginIndex and endIndex.

What is mutable string

----------------------

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

class StringBufferExample{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello ");

sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java

}

}

2) StringBuffer insert( ) method

The insert( ) method inserts the given string with this string at the given position.

class StringBufferExample2

{

public static void main(String args[])

{

StringBuffer sb=new StringBuffer("Hello ");

sb.insert(1,"Java");//now original string is changed

System.out.println(sb);//prints HJavaello

}

}

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

class StringBufferExample3{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.replace(1,3,"Java");

System.out.println(sb);//prints HJavalo

}

}

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

class StringBufferExample4{

public static void main(String args[]){

StringBuffer sb=new StringBuffer("Hello");

sb.delete(1,3);

System.out.println(sb);//prints Hlo

}

}

5) StringBuffer reverse() method

The reverse() method of StringBuilder class reverses the current string.

class StringBufferExample5

{

public static void main(String args[])

{

StringBuffer sb=new StringBuffer("Hello");

sb.reverse();

System.out.println(sb);//prints olleH

}

}

Exception Handling in Java

=========================================================

The Exception Handling in Java is one of the powerful mechanism to handle the runtime

errors so that normal flow of the application can be maintained.

In this page, we will learn about Java exceptions, its type and the difference between

checked and unchecked exceptions.

What is Exception in Java:

=====================

Dictionary Meaning: Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program.

It is an object which is thrown at runtime.

What is Exception Handling

Exception Handling is a mechanism to handle runtime errors such as

ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

-----------------------------------------

The core advantage of exception handling is to maintain the normal flow of the application.

An exception normally disrupts the normal flow of the application that is why we use exception handling.

Let's take a scenario:

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

statement 8;

statement 9;

statement 10;

Suppose there are 10 statements in your program and there occurs an exception at statement 5,

the rest of the code will not be executed i.e. statement 6 to 10 will not be executed.

If we perform exception handling, the rest of the statement will be executed.

That is why we use exception handling in Java.

Do You Know?

The java.lang.Throwable class is the root class of Java Exception hierarchy

which is inherited by two subclasses: Exception and Error.

A hierarchy of Java Exception classes are given below:

hierarchy of exception handling

Types of Java Exceptions

--------------------------------------

There are mainly two types of exceptions: checked and unchecked. Here,

an error is considered as the unchecked exception.

According to Oracle, there are three types of exceptions:

-->Checked Exception

-->Unchecked Exception

-->Error

Types of Java Exceptions:

Difference between Checked and Unchecked Exceptions

=================================================

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as

checked exceptions e.g. IOException, SQLException,ClassNotFoundException etc.

Checked exceptions are checked at compile-time.

All checked exceptions are depends on another java applications.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g.

ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

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

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword Description

=================================================================================

1. try The "try" keyword is used to specify a block where we should place exception code.

The try block must be followed by either catch or finally. It means, we can't use try block alone.

2.catch The "catch" block is used to handle the exception.

It must be preceded by try block which means we can't use catch block alone.

It can be followed by finally block later.

3.finally The "finally" block is used to execute the important code of the program.

It is executed whether an exception is handled or not.

4.throw The "throw" keyword is used to throw an exception.

5.throws The "throws" keyword is used to declare exceptions.

It doesn't throw an exception. It specifies that there may occur an exception in the method.

It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

public class JavaExceptionExample{

public static void main(String args[]){

try{

//code that may raise exception

int data=100/0;

}catch(ArithmeticException e){System.out.println(e);}

//rest code of the program

System.out.println("rest of the code...");

}

}

Test it Now

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

int a=50/0;//ArithmeticException

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a

NullPointerException.

String s=null;

System.out.println(s.length());//NullPointerException

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException.

Suppose I have a string variable that has characters, converting this variable into digit

will occur NumberFormatException.

String s="abc";

int i=Integer.parseInt(s);//NumberFormatException

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

int a[]=new int[5];

a[10]=50; //ArrayIndexOutOfBoundsException

Java Inner classes

==========================================================

Java inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

class Java\_Outer\_class

{

//code

class Java\_Inner\_class

{

//code

}

}

Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of outer class including private.

2) Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.

3) Code Optimization: It requires less code to write.

Types of Nested classes

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

----->Non-static nested class (inner class)

1. Member inner class

2. Anonymous inner class

3. Local inner class

---->Static nested class

Type Description

Member Inner Class A class created within class and outside method.

Anonymous Inner Class A class created for implementing interface or extending class. Its name is decided by the java compiler.

Local Inner Class A class created within method.

Static Nested Class A static class created within class.

Nested Interface An interface created within class or interface.

multitasking:

Definition:executing multiple task simultaniously

Multithreading:

========================================================

Multithreading in java is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing.

Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading:

---------------------------------------------------

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.

2) You can perform many operations together, so it saves time.

3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking is a process of executing multiple tasks simultaniously. We use multitasking to utilize the CPU.

Multitasking can be achieved in two ways:

=========================================

----->Process-based Multitasking (Multiprocessing) os level is best (Ex: writing program, listening music, downloading music etc)

---->Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

Each process has an address in memory. In other words, each process allocates a separate memory area.

A process is heavyweight.

Cost of communication between the process is high.

Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

Threads share the same address space.

A thread is lightweight.

Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight subprocess, the 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 uses a shared memory area.

Java Multithreading

--------------------------

As shown in the above figure, a thread is executed inside the process.

There is context-switching between the threads. There can be multiple processes inside the OS,

and one process can have multiple threads.

Life cycle of a Thread (Thread States)

=========================================

1.New

2.Runnable

3.Running

4.Non-Runnable (Blocked)

5.Terminated

A thread can be in one of the five states. According to sun, there is only 4 states in thread

life cycle in java new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

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

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of

start() method.

2) Runnable

The thread is in runnable state after invocation of start() method,

but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

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

How to create thread

==========================

There are two ways to create a thread:

--->By extending Thread class

--->By implementing Runnable interface.

Thread class:

--------------------------

Commonly used methods of Thread class:

--->public void run( ): is used to perform action for a thread.

--->public void start( ): starts the execution of the thread.JVM calls the run() method on the thread.

---->public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

------->public void join(): waits for a thread to die.

----->public void join(long miliseconds): waits for a thread to die for the specified miliseconds.

----->public int getPriority(): returns the priority of the thread.

----->public int setPriority(int priority): changes the priority of the thread.

---->public String getName(): returns the name of the thread.

--->public void setName(String name): changes the name of the thread.

--->public Thread currentThread(): returns the reference of currently executing thread.

public int getId(): returns the id of the thread.

public Thread.State getState(): returns the state of the thread.

----->public boolean isAlive(): tests if the thread is alive.

----->public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.

----->public void suspend(): is used to suspend the thread(depricated).

----->public void resume(): is used to resume the suspended thread(depricated).

---->public void stop(): is used to stop the thread(depricated).

public boolean isDaemon(): tests if the thread is a daemon thread.

public void setDaemon(boolean b): marks the thread as daemon or user thread.

public void interrupt(): interrupts the thread.

public boolean isInterrupted(): tests if the thread has been interrupted.

public static boolean interrupted(): tests if the current thread has been interrupted.

Runnable interface:

=====================================

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run( ).

public void run( ): is used to perform action for a thread.

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

A new thread starts(with new callstack).

The thread moves from New state to the Runnable state.

When the thread gets a chance to execute, its target run( ) method will run.

1) Java Thread Example by extending Thread class

-------------------------------

class Multi extends Thread

{

public void run()

{

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

}

public static void main(String args[])

{

Multi t1=new Multi();

t1.start();

}

}

Output:thread is running...

2) Java Thread Example by implementing Runnable interface

class Multi3 implements Runnable

{

public void run()

{

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

}

public static void main(String args[])

{

Multi3 m1=new Multi3();

Thread t1 =new Thread(m1);

t1.start();

}

}

Output:thread is running...

\*\*\*\*wait a thread

1.yield()

2.join()

3.sleep()

1. yield()->this method is useful to pass the current thread,if other thread is maximum priority of the paused thread then it will executed the other thread, if there is no maximum thread priority then it will execute it self.

--> if both thread a same priority then it will execution depends on thread scheduler:

2. join(); if u complete the a perticular thread then we cann goo for join() method afetr it execution of the thread it stop to execute the statement, then automatically die in that thrread.

\*\*\*\*\*to find that thread isallive or not by using isAlive() method

3. sleep();milliseconds

File i/o concept in java:

======================================================

import java.io.\*-->package all related io package.

1.character stream(FileReader, FileWriter, BufferedReader, BufferedWriter, printwriter)

2.Bytestream(outstream,inputstream,FileInputStream,FileOutputStream)

3.Data stream

4.BufferdStream

5.Object Stream

Serialization in java

======================================================

Serialization in Java is a mechanism of writing the state of an object into a byte stream.

It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called deserialization.

Advantages of Java Serialization

It is mainly used to travel object's state on the network (which is known as marshaling).

java serialization

java.io.Serializable interface

Serializable is a marker interface (has no data member and method).

It is used to "mark" Java classes so that objects of these classes may get the certain capability.

The Cloneable and Remote are also marker interfaces.

It must be implemented by the class whose object you want to persist.

The String class and all the wrapper classes implement the java.io.Serializable interface by default.

Let's see the example given below:

import java.io.Serializable;

public class Student implements Serializable//marker interface

{

int id;

String name;

public Student(int id, String name)

{

this.id = id;

this.name = name;

}

}

In the above example, Student class implements Serializable interface. Now its objects can be converted into stream.

\*\*\*\*\*\*ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor

1) public ObjectOutputStream(OutputStream out) throws IOException {} creates an ObjectOutputStream that writes to the specified OutputStream.

Important Methods

Method Description

1) public final void writeObject(Object obj) throws IOException {} writes the specified object to the ObjectOutputStream.

2) public void flush() throws IOException {} flushes the current output stream.

3) public void close() throws IOException {} closes the current output stream.

Example of Java Serialization

------------------------------

In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

import java.io.\*;

class Persist

{

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

{

Student s1 =new Student(211,"ravi");

FileOutputStream fout=new FileOutputStream("f.txt");

ObjectOutputStream out=new ObjectOutputStream(fout);

out.writeObject(s1);

out.flush();

System.out.println("success");

}

}

success

Deserialization in java

----------------------------

Deserialization is the process of reconstructing the object from the serialized state.

It is the reverse operation of serialization.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

1) public ObjectInputStream(InputStream in) throws IOException {} creates an ObjectInputStream that reads from the specified InputStream.

Important Methods

Method Description

1) public final Object readObject() throws IOException, ClassNotFoundException{} reads an object from the input stream.

2) public void close() throws IOException {} closes ObjectInputStream.

Example of Java Deserialization

import java.io.\*;

class Depersist{

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

ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));

Student s=(Student)in.readObject();

System.out.println(s.id+" "+s.name);

in.close();

}

}

211 ravi

Java Transient Keyword

================================

Java transient keyword is used in serialization. If you define any data member as transient,

it will not be serialized.

Let's take an example, I have declared a class as Student, it has three data members id, name and age.

If you serialize the object, all the values will be serialized but I don't want to serialize one value,

e.g. age then we can declare the age data member as transient.

Example of Java Transient Keyword

--------------------------------------

In this example, we have created the two classes Student and PersistExample.

The age data member of the Student class is declared as transient, its value will not be serialized.

If you deserialize the object, you will get the default value for transient variable.

Let's create a class with transient variable.

import java.io.Serializable;

public class Student implements Serializable

{

int id;

String name;

transient int age;//Now it will not be serialized

public Student(int id, String name,int age) {

this.id = id;

this.name = name;

this.age=age;

}

}

Now write the code to serialize the object.

import java.io.\*;

class PersistExample{

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

{

Student s1 =new Student(211,"ravi",22);//creating object

//writing object into file

FileOutputStream f=new FileOutputStream("f.txt");

ObjectOutputStream out=new ObjectOutputStream(f);

out.writeObject(s1);

out.flush();

out.close();

f.close();

System.out.println("success");

}

}

Output:

success

Now write the code for deserialization.

import java.io.\*;

class DePersist

{

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

{

ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));

Student s=(Student)in.readObject();

System.out.println(s.id+" "+s.name+" "+s.age);

in.close();

}

}

211 ravi 0

As you can see, printing age of the student returns 0 because value of age was not serialized.

garbage collection:

====================

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.

How can an object be unreferenced?

There are many ways:

By nulling the reference

By assigning a reference to another

By anonymous object etc.

1) By nulling a reference:

Employee e=new Employee();

e=null;

2) By assigning a reference to another:

Employee e1=new Employee();

Employee e2=new Employee();

e1=e2;//now the first object referred by e1 is available for garbage collection

3) By anonymous object:

new Employee();

finalize() method

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

protected void finalize()

{

}

CollectionFramework

===========================================

Collection is an interface and it is parent to all the interfaces in java.

Collection(I)

1. List(I)-->ArrayList(c),//by default array size is 10

LinkedList(c),

Vector(C)

-->stack(C)

-->it is growable by default in nature.

-->it can store homogenious and heterogenious elements

-->it contain some buildin methods

\*\*\* duplicates are allowed

\*\*\*\*insertion order is preserved(in which way u inserted elemenets that order only it is displayed)

2.Set(I):-----

HashSet(C)

LilnkedHashSet(C)

SoretedSet(I)

NavigableSet(I)

--TreeSet(C)

3.Queue(I)

PriorityQueue

BlockingQueue

PriorityBlockingQueue

4.Map(I) special interface in java.

HashedMap(c)

LinkedHashMap(c)

SortedMap(I)

NavigableMap(I)

Java IDE: integrated development environment:

Intellij , eclipse, myeclipse, netbeans

application server: enterprise, servletes, jsp, ejb ,jms(in built it contains web server)

web server: servlet, jsp,