**Primitive Variables:** - Primitive variables are byte, short, int, long, float, double, char etc.

**Reference Variables:** - Reference variables are used to refer or access an object.

**Byte** -> 1 byte -> -128 to 127

**Short** -> 2 byte -> -32768 to 32767

**Int** -> 4 byte -> -231 to 231-1

**Float** -> 4 byte -> Same as above

**Long** -> 8 byte -> -263 to 263-1

**Double** -> 8 byte -> Same as above

**Char** -> 2 byte -> \u0000 to \uffff i.e. 0 to 65,535

**Instance Variables:** - It is placed inside a class but outside a method. So, they are initialize when a class is instantiated. They have default values provided by JVM i.e. for **int** it is **0**, for **float** it is **0.0**, for **char** it is **‘\0’**, for **Boolean** it is **false**. We can mark it as **Public, Private, Protected, Default & Final** also.

**Local/Stack Variables:**  - These variables are used inside a method and such variables are creating only when a method is called and destroyed when method is terminated. They can be mark as Final but not Public, Private, Protected.

**Final Variables:** - Final is an access modifier applied to classes, instance variables, methods and local variables. Once a variable mark as final then its value cannot be change through the file.

**Static Variables:** - Static contents are loaded before any class instance is created. Only 1 copy of such variables is created and commonly share by all instances of the same class.

**Static {**

**//Static Variables**

**}**

This is an access modifier that can be applied to methods/variables.

**Abstract:** - It is an access modifier that can be applied to class/methods but not variables. If a class containing 1 or more declared methods, then they spoil entire class and make it abstract. We cannot create object of such class. It provides a foundation on which other classes are built.

**Wrapper Class:** -It is mainly provided for 2 reasons: -

1. To provide a mechanism that enables us to wrap primitive type values in an object so that all the activities reserved for the Object are also used for built in types.
2. For conversion facilities also.
3. Six Numeric Wrapper classes extends **“NUMBER”** class and they are **Byte, Short, Integer, Long, Float, Double** etc.

**Value of ()** is a **String** class method and it takes a string as a parameter and if that string is not properly formed to a specified type then it throws **java.lang.NumberFormatException**

**Integer a = Integer.valueOf(“15”)**

// It will convert “15” as a String to Integer 15 and then we can use all the method provided by **OBJECT** class.

**toString ()** is a method of “**OBJECT”** cosmic super class which can be further overloaded to construct wrapper class objects that return a string reference.

**String a = Integer.toString(125,16) // Returns Hex Conversion i.e. 7d**

**String a = Integer.toString(55) // Convert int to String and return “55” as String**

**String a = Integer.toString(78,2) // Return Binary Conversion i.e. 1001110**

**String a = Integer.toOctalString(125) // Return Octal Conversion i.e. 175**

**String a = Integer.toHexString(125) // Returns Hex Conversion i.e. 7d**

**String a = Integer.toBinaryString(125) // Return Binary Conversion i.e. 1111101**

**OBJECT CLASS: -** It is predefined in **java.lang** package. It defines basic state and behavior of all built in classes as well as user defined class objects. It means **Object class** is at top of hierarchy i.e. it is a super class of all classes. In java every Exception, Event extends it. It defines common functionality on which other classes are built.It has various methods defined **toString(), hashCode(), notify(), wait(), notifyAll()** etc.

**GARBAGE COLLECTION: -** It is an automatic solution to memory management. We can’t completely have controlled when it starts and when it doesn’t. **HEAP** part of a memory is involved in garbage collection, where object lives and dies. Since, a heap contained limited storage JVM make sure that heap can be utilized efficiently. So, when JVM sense that memory is running low it invokes garbage collector. **An object is eligible for garbage collection when no live thread can access it.** When an object reference is unreachable to any method/thread.

**Runtime rt = Runtime.getRuntime();**

**Rt.gc();**

**Runtime Class** provides an environment through which we can directly communicate with JVM.

**System.gc(); // Static method of System class to invoke garbage collector**

**Finalize: -** Java provides a mechanism to run some code just before object is deleted by GC. So, such code is placed inside a finalize method. This is great idea to release all resources that an object uses before it destroyed. To release all resources and perform cleanup operations we can use this. **Most importantly this method can be invoked at most 1 time only.**

**Inheritance: -** It is one of the striking feature of OOP languages that enables us to inherit existing classes so that newly created class can have all the features or properties of an existing or parent class. It enables us extensibility, reusability due to which development as well as compilation time is reduced. In java we have only **Single/Multilevel/Hierarchical** inheritance.

**Super** keyword in inheritance used to invoke super class constructor, invoking base class overriding methods, accessing data members of a super class.

**Abstract Class: -** It gives an additional flexibility and extensibility. It provides most common generic behavior to all of its child classes. It means it provides a foundation on which other classes are built.

**Interface: -** It can be used as a replacement of a **Multiple Inheritance.** It is a 100% abstract body. An interface informed other classes **What should they do? But not how it does it?** A class can extend only 1 class but it can implement more than 1 interfaces. However, an interface extends another interface.

1. All variables of interface are **Static and Final.**
2. All methods of interface are **Abstract.**

|  |  |
| --- | --- |
| **Abstract Class** | **Interface** |
| **Incomplete class**. So, can’t instantiate | Purely **abstract class** |
| Method **declaration and definition** | Method **declaration** |
| **Extend** keyword is used | **Implements** keyword is used |
| Can extend only **1 class** | Can implement **more than 1 interface** |
| Can have **constructor** | Can’t have **constructor** |

**When same method is present in Parent as well as Child class with same name and same number of arguments then it is called Method Overriding.**

**Inheritance Example: -**

Public class **Car**

{

Public void **start** ()

{

System.out.println(“Car-Start”);

}

Public void **stop** ()

{

System.out.println(“Car-Stop”);

}

Public void **refuels** ()

{

System.out.println(“Car- Refuels”);

}

}

Public class **BMW** extends **Car** //**HAS-A** relationship

{

Public void **start** ()

{

System.out.println(“BMW -Start”);

}

Public void **safety** ()

{

System.out.println(“BMW -Safety”);

}

}

Public class **TestCar**

{

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

{

// **Static Polymorphism (Compile Time Polymorphism)**

**BMW b = new BMW ();**

**b.start (); //BMW-Start**

**b.stop (); //Car-Stop**

**b.refuels (); //Car-Refuels**

**b.safety (); // BMW-Safety**

**//Dynamic Polymorphism (Run Time Polymorphism)**

**//Child object referred by Parent Class**

**Car c = new BMW (); //This is known as TOP CASTING**

**c.start (); //BMW-Start**

**c.stop (); //Car-Stop**

**c.refuels (); //Car-Refuels**

**It cannot access c.safety () because it’s a method of BMW class which cannot be referred by Parent class reference.**

**}**

**}**

**Interface Example: -**

Public interface **USBank**

{

Int min\_bal = 100; // Only method declaration 100% abstract body

Public void credit (); // Variables are Static and Final

Public void debit (); //No static method in interface

Public void transfer (); //No Main method as we cannot create object of interface

}

Public class **HSBCBank** implements **USBank** // **IS-A** relationship

{

Public void **credit** ()

{

System.out.println(“HSBC-Credit”);

}

Public void **debit** ()

{

System.out.println(“HSBC-Debit”);

}

Public void **transfer** ()

{

System.out.println(“HSBC-Transfer”);

}

Public void **educationLoan** ()

{

System.out.println(“HSBC-Education Loan”);

}

Public void **carLoan** ()

{

System.out.println(“HSBC-Car Loan”);

}

Public void **homeLoan** ()

{

System.out.println(“HSBC-Home Loan”);

}

}

Public class **TestBank**

**{**

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

**{**

//**Dynamic Polymorphism**

**USBank ub = new HSBCBank();**

**Ub.credit (); //HSBC-Credit**

**Ub. Debit (); //HSBC-Debit**

**Ub.transfer(); //HSBC-Transfer**

**Ub.carLoan (); and ub.homeLoan(); and ub.educationLoan (); //Not possible to call }}**

**Hiding the business logic is called Abstraction. Through abstract class we never achieve full abstraction. We can only achieve partial abstraction as we can define methods in Abstract Class also which are common to inherited classes.**

**Exception Handling: -** In JAVA all exceptions are of the java.lang.Exception class.

**Throwable Class**

**Error Class**

**Exception Class**

**ERROR represents serious abnormal conditions which an application should not try to catch. They are unchecked.**

**Checked Exception  
(Compile Time)**

**Unchecked Exception  
(Runtime Exception)**

**ArrayIndexOutOfBoundException, ArithmeticException, NullPointerException**

**ClassNotFoundException, IOException, FileNotFoundException**

**Throw:** - It is used to throw an exception explicitly. It must be used in try block.

**Throws: -** It is used when the function may throw any checked exception. It is used in method signature which might throw mentioned exception.

**Finally:** - It is used to cleanup resources allocated by **try block** such as closing files, database etc. The **Finally block** is used to execute given set of statements whether an exception has occurred or not. We can have only 1 **Finally block** for each **try block.**