**Java At A Glance**

* **String** is a final class from **java.lang** package.
* **System** is a final class from **java.lang** package. **out** is static member of System class of type **PrintStream**. **println** is a method of **PrintStream** class.
* Setting **path** variable and **classpath** variable

set path=I:\Java\jdk1.7.0\_25\bin

echo %path%

>javac  -d  (Specify the path where to save generated .class files)  FileName.java

>java  -classpath  (path of generated .class files)  ClassName

> set classpath=I:\Classes

> echo %classpath%

* **Public JRE** is a Standalone JRE. Any java application running on your system can use this JRE

**Using newInstance() Method**

Class c = Class.forName("packageName.MyClass");

MyClass object = (MyClass) c.newInstance();

**Using clone() method**

MyClass object1 = new MyClass();

MyClass object2 = object1.clone();

**Using object deserialization**

ObjectInputStream inStream = new ObjectInputStream(anInputStream );

MyClass object = (MyClass) inStream.readObject();

**Creating string and array objects**

String s = "string object";

int[] a = {1, 2, 3, 4};

**default**

* *default* keyword is used to define the default methods in an interface (From Java 8)

interface MyInterface

{

    public default void myDefaultMethod()

     {

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

    }

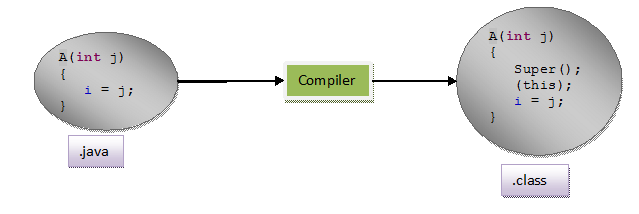
}

**SIB**

* Static Initialization Block(**SIB**) is used to initialize only **static** variables.

**IIB**

* IIB stands for **Instance Initialization Block**. As the name suggest this block is used to initialize **state of an object**. State of an object is indicated by instance variables or non-static variables. So, IIB is used to initialize instance variables or non-static variables.
* We all know that [first statement of constructor](http://javaconceptoftheday.com/constructors-in-java/) is super() or this(). After executing first statement, IIB blocks are called. After executing IIB blocks, remaining statements are executed.
* So, when the constructor is called while creating an object (Line 19), compiler will treat constructor code like this,



where (this) is a calling statement to IIB block.

You can keep any number of IIB blocks in a class. All blocks are called after super() in the constructor in the order they appear.

* **Note** : IIB blocks will **not** be called from the **constructor** in which **this()** statement is written as a first statement.
* IIBs can also be written as,

class A

{

     int i = 10;

}

This is same as,

class A

{

     int i;

     {

          i = 10;

     }

}

**Constructor**

* Only **public**, **protected** and **private** keywords are allowed before a **constructor** name. If you keep any other keyword before a constructor name, it gives **compile** time error.
* First statement in a constructor must be either **super**() or **this**(). If you put any other statements you will get compile time error.If you don’t include these statements, by default compiler will keep **super**() calling statement. **super**() – It is a calling statement to default constructor of super class. **this**()- it is a calling statement to constructor of the same class.
* **Private** constructors are used to restrict the instantiation of a class. When a class needs to prevent other classes from creating it’s objects then private constructors are suitable for that. Objects to the class which has only private constructors can be created within the class. A very good use of private constructor is in singleton pattern. This ensures only one instance of a class exist at any point of time. Here is an example of singleton pattern using private constructor.

class MyClass

{

    private static MyClass object = null;

    private MyClass()

     {

        //private constructor

    }

    public MyClass getObject()

    {

        if(object == null)

        {

            object = new MyClass();   //Creating object using private constructor

        }

        return object;

    }

}

**Super**

* super keyword is used to access super class members inside the sub class. Using super keyword, we can access super class methods, super class fields and super class constructors in the sub classes.
* super class constructor is called by **super()** calling statement.You can’t use super() calling statement outside the constructor. By default, super() calling statement is the first statement in any constructor.
* If you want same implementation as that of super class method in the sub class, but want to add some more extra statements to it, in such cases, super keyword will be very useful. First call the super class method using super keyword and after it add extra statements according to requirements in the sub class method.
* **Note** : You **can’t** use **super** and **this** keywords in a **static** **method** and in a **static initialization block** even though you are referring **static members**.

**This**

* **Note** : You **can’t** use **super** and **this** keywords in a **static** **method** and in a **static initialization block** even though you are referring **static members**.
* this keyword is used to access other members of the same class. Using this keyword, you can access methods, fields and constructors of the same class within the class. this refers to current instance of the class.
* this() is the calling statement to same class constructor. It must be used within constructor only. If it is used, it must be the first statement in the constructor.
* You can’t use super and this keywords in a static method and in a static initialization block even though you are referring static members.

**Final**

* **final keyword in java** can be used with a class, with a variable and with a method. final keyword restricts the further modification. When you use final keyword with an entity (class or variable or method), it gets the meaning that entity is complete and can not be modified further.
* We **can’t** create a **subclass** to the class or we **can’t** **extend** a class or we **can’t modify** a class which is declared as **final**.
* We **can’t override** a method or we **can’t modify** a method in the **sub class** which is declared as **final** in the **super class**.
* The value of a final variable can not be changed in the whole execution once it got initialized.
* Any class or any method can be either **abstract or final** but not both. abstract and final are totally opposite. Because, abstract class or abstract method must be implemented or modified in the sub classes but final does not allow this. This creates an ambiguity.
* final method can be overloaded and that overloaded method can be overridden in the sub class.
* final variable can not be re-initialized but final variable can be used to initialize other variables.
* When an array reference variable is declared as final, only variable itself is final but not the array elements.
* When a reference variable is declared as final, you can’t re-assign a new object to it once it is referring to an object. But, you can change the state of an object to which final reference variable is referring.
* Static variables, non-static variables and local variables all can be final. once the final variables are initialized, even you can’t re-assign the same value.
* If the **global variables** are not **initialized explicitly**, they get default value at the time of object creation. But **final global variables** don’t get default value and they must be **explicitly initialized** at the time of **object creation**. Uninitialized final field is called **Blank Final Field**.
* **final non-static global** variable must be **initialized** at the time of **declaration** or in all **constructors** or in any one of **IIBs – Instance Initialization Blocks.**
* **final static global variable** must be **initialized** at the time of **declaration** or in any one of **SIBs – Static Initialization Blocks.** (final static global variable **can’t** be initialized in constructors)

**Inheritance**

* Constructors, SIB – Static Initialization Block and IIB – Instance Initialization Block of super class will not be inheriting to its sub class. But they are executed while creating an object to sub class
* Static members of super class are inheriting to sub class as static members and non-static members are inheriting as non-static members only.
* By default, every class is a sub class of **java.lang.Object** class. So, every class in java has properties inherited from Object class

**private —> default or no access modifiers —> protected —> public**

**Private**

* **Private** members of a class whether it is a field or method or constructor **can not** be accessed **outside** the **class**.
* **Private** members will **not** be **inherited** to sub class.
* Class can not be a private except inner classes. Inner classes are nothing but again members of outer class. So members of a class (field, method, constructor and inner class) can be private but not the class itself.
* We can’t create sub classes to that class which has only private constructors.

**Default or Package or No-Access Modifiers**

* Default members or members with No-Access modifiers are accessed or visible within the package only. It applies to outer classes also.
* Default members can be inherited to sub classes within package.

**Protected**

* Protected member can be used within the package only.
* we can’t instantiate a class outside the package which has only protected constructors.
* **Note** : Outer class can not be protected.
* **Note** : We can create sub classes to a class which has only protected constructors but we can’t create objects to that class outside the package.

**Public**

* Public members can be used anywhere.
* Public members can be inherited to any sub class.

**Type Casting**

* **byte < short < int < long < float < double.**
* **Auto Widening -** When you are converting data from small sized data type to big sized data type, i.e when you are converting data from left-placed data type to right-placed data type in the above order, auto widening will be used. For example, when you are converting byte to short or short to int, auto widening will be used.
* **Explicit Narrowing -** When you are converting data from big sized data type to small sized data type, i.e when you are converting data from right-placed data type to left-placed data type in the above order, explicit narrowing will be used. For example, when you are converting double to float or float to int, explicit narrowing will be used.
* **Auto-Up Casting -** Auto-Up Casting is used to change the type of object from sub class type to super class type. i.e an object of sub class type is automatically converted to an object of super class type.
* **Explicit Down Casting -** Explicit down Casting is used to change the type of object from super class type to sub class type. i.e you have to explicitly convert an object of super class type to an object of sub class type.

**ClassCastException**

ClassCastException in java is a run time error it occurs when an object can not be casted to another type.

An object is automatically upcasted to its super class type. You need not to mention class type explicitly. But, when an object is supposed to be downcasted to its sub class type, then you have to mention class type explicitly. In such case, there is a possibility of occurring class cast exception. In most of time, it occurs when you are trying to downcast an object explicitly to its sub class type.

package com;

class A

{

    int i = 10;

}

class B extends A

{

    int j = 20;

}

class C extends B

{

    int k = 30;

}

public class ClassCastExceptionDemo

{

    public static void main(String[] args)

    {

        A a = new B();   //B type is auto up casted to A type

        B b = (B) a;     //A type is explicitly down casted to B type.

        C c = (C) b;    //Here, you will get class cast exception

        System.out.println(c.k);

    }

}

You will get ClassCastException. Below is the sample of the error.

**Exception in thread “main” java.lang.ClassCastException: com.B cannot be cast to com.C**  
**at com.ClassCastExceptionDemo.main(ClassCastExceptionDemo.java:23)**

In the above example, Class B extends Class A and Class C extends Class B. In the main method, Class B-type object is created (Line 21). It will be having two non-static fields. one field (int i) is inherited from class A and another one is its own field (int j). ‘a’ is Class A-type reference variable which will be pointing to this newly created object. In the next statement (Line 22), reference variable ‘a’ is assigned to ‘b’ which is Class B-type reference variable. After execution of this statement, ‘b’ will also be pointing to the same object to which ‘a’ is pointing. In the third statement, ‘b’ is assigned to ‘c’ which is Class C-type reference variable. So, ‘c’ will also be pointing to same object to which ‘a’ and ‘b’ are pointing. While executing this statement, you will get run time exception called Class Cast Exception.

**Why you got this exception?**

Every sub class extends its super class. i.e every child class will have some additional properties along with some inherited properties from its parent class. In the above example, Class A has one property (int i). Class B has two properties, one is it’s own and another one is inherited. Class C has three properties. one is it’s own and two are inherited. In this example, Class C-type reference variable is referring to Class B-type object. Class B-type object will be having only two properties. But, through Class C-type reference variable, you can access Class C’s own property (int k) like in the line 24. But, actually this property does not exist in Class B-type object. This creates the confusion. Class B-type can not be casted to Class C-type. That’s why, you will get class cast exception.

Put ClassCastException in simple terms. ClassCastException occurs when code has attempted to cast an object to a type of which it is not an object. In the above example, Class B is a Class A type but Class B is not a Class C type. Therefore, you are getting ClassCastException.

Consider one more case of ClassCastException.

public class ClassCastExceptionDemo

{

    public static void main(String[] args)

    {

        Object o = new String();

        Integer i = (Integer) o;

    }

}

We all know that every class in java is a sub class of java.lang.Object class. String is also a subclass of Obeject class and Integer is also a subclass of Object class. In the above example, String object is created and it is automatically up casted to Object type. Further, this object is explicitly downcasted to Integer type. This causes ClassCastException, because, String object is not an Integer type.

**Abstraction**

**Interface**

* Interface methods must be implemented as public. Because, interface methods are public by default and you should not reduce the visibility of any methods while overriding.
* interface fields are static and final by default and you can’t change their value once they are initialized
* Interfaces can’t have constructors.
* Interfaces can’t have initializers.
* Interfaces can’t be local members of a method.