To **remove** those **unused imports** automatically, just click on the class and press the

shortcut “Ctrl + Shift + O” to initilize the “Organize **imports**”

To see all shortcuts inside eclipse itself:

1. Open Eclipse

2. Click on Preferences under window

3. Click on General

4. Click on Keys

To print system.out.prtintln – type syso then press ctrl+space

To get template for function – write public then ctrl+space / write name of fun alone then ctrl+space

To get template for main function- write main then ctrl+space

I­­n layman terms: JDK is grandfather JRE is father and JVM is their son. **[i.e. JDK > JRE > JVM ]**

**JDK = JRE + Development/debugging tools**

**JRE = JVM + Java Packages Classes(like util, math, lang, awt,swing etc)+runtime libraries.**

**JVM = Class loader system + runtime data area + Execution Engine.**

In other words if you are a Java programmer you will need JDK in your system and this package will include JRE and JVM as well but if you are normal user who like to play online games then you will only need JRE and this package will not have JDK in it.

**JVM :**

The Java Virtual machine (JVM) is the virtual machine that run the Java bytecodes. The JVM doesn't understand Java typo, that's why you compile your \*.java files to obtain \*.class files that contain the bytecodes understandable by the JVM. It's also the entity that allows Java to be a "portable language" (write once, run anywhere). Indeed there are specific implementations of the JVM for different systems (Windows, Linux, MacOS, see the wikipedia list..), the aim is that with the same bytecodes they all give the same results.

**JDK and JRE**

To explain the difference between JDK and JRE, the best is to read the Oracle documentation and consult the diagram :

**Java Runtime Environment (JRE)**

The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language. In addition, two key deployment technologies are part of the JRE: Java Plug-in, which enables applets to run in popular browsers; and Java Web Start, which deploys standalone applications over a network. It is also the foundation for the technologies in the Java 2 Platform, Enterprise Edition (J2EE) for enterprise software development and deployment. The JRE does not contain tools and utilities such as compilers or debuggers for developing applets and applications.

**Java Development Kit (JDK)**

The JDK is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.

Note that Oracle is not the only one to provide JDK.

[JIT Compile Process (Courtesy: Oracle documentation)](https://i.stack.imgur.com/pQRpc.gif)

[](https://i.stack.imgur.com/AaveN.png)

a class is loaded:

* when the new bytecode is executed. For example, SomeClass f = new SomeClass();
* when the bytecodes make a static reference to a class. For example, System.out.

**When Class is loaded in Java**

Class loading is done by ClassLoaders in Java which can be implemented to eagerly load a class as soon as another class references it or [lazy load](http://javarevisited.blogspot.sg/2012/07/why-enum-singleton-are-better-in-java.html) the class until a need of class initialization occurs. If Class is loaded before its actually being used it can sit inside before being initialized.

|  |  |
| --- | --- |
|  | Classes are loaded at runtime to execute their code.  Classes are loaded at compile time to check code using the class for type safety. Whenever you write code that uses a class (eg, calling a method on it), the compiler needs to load that class to make sense of your code (eg, to check methods or base type) |

When a class is loaded it is not initialized.

**When and how is a Java class initialized?**

A class is initialized when a symbol in the class is first used.

1) an [Instance](http://javarevisited.blogspot.sg/2012/02/difference-between-instance-class-and.html) of class is created using either new() keyword or using [reflection](http://javarevisited.blogspot.sg/2012/05/how-to-access-private-field-and-method.html) using class.forName(), which may throw [ClassNotFoundException](http://javarevisited.blogspot.sg/2011/08/classnotfoundexception-in-java-example.html) in Java.

2) an static method of Class is invoked.

3) an static field of Class is assigned.

4) an static field of class is used which is not a constant variable.

5) if Class is a top level class and an [assert statement](http://javarevisited.blogspot.sg/2012/01/what-is-assertion-in-java-java.html) lexically nested within class is executed.

public class SomethingCaller {

public static Something something = null; // (1) does not cause class loading

public static Class<?> somethingClass = Something.class; // (2) causes class loading

public void doSomething() {

new Something(); // (3) causes class loading

}

}

JVM will initialize superclass and fields in textual order, initialize static, final fields first, and give every field a default value before initialization.

**The compiler optimizes inlineable static final fields by embedding the value in the bytecode instead of computing the value at runtime.**

**When you fire up a JVM and load a class for the first time (this is done by the classloader when the class is first *referenced* in any way) any static blocks or fields are 'loaded' into the JVM and become accessible.**

there is only one copy of static variable will be present in [Java Heap memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html), which can be accessed or altered by any object. Mostly utility methods are declared as static method,   
**loading and unloading of static fields.** Static fields or variables are initialized when Class is first loaded by ClassLoader while they are unloaded from memory if there is no live reference of field from any Thread static is present and they are eligible for Garbage Collection.

Static fields are initialised when a class is loaded and and are discarded when the classloader for that class is unloaded. They can be cleaned up, even duplicated in another class loader.

For applications like those this use OSGi, static variables don't live for the life of the application can be reloaded many times.

How this is implement may be JVM dependant but the Sun/Oracle JVM creates an "object" to hold the static fields for a class. This object is accessible via the Unsafe class which can also be used to examine this "objects" fields.

We have 3 segments in our memory:

1. Stack Segment — contains local variables and Reference variables (variables that hold the address of an object in the heap).
2. Heap Segment — contains all created objects in runtime, objects only plus their object attributes (instance variables).
3. **Code Segment — the segment where the actual compiled Java bytecodes resides when loaded. Static members (variables or methods) are called class members, meaning they reside where the class (bytecode) resides, which is in the Code Segment.**

**Storage Area of static member in JVM memory?**  
static member variables are kept on the Permanent Genration.

**System.out.println:**

System is a **class** in the java.lang package.

out is a **static member** of the System class, and is an instance of java.io.PrintStream.

println is a **method** of java.io.PrintStream. This method is overloaded to print message to output destination, which is typically a console or file.

the System class belongs to java.lang package

e.g. TestClass.StringStr.length();

<<ClassName.StaticVariableofString.StringFunciton>>

class System {

public static final PrintStream out;

//...

}

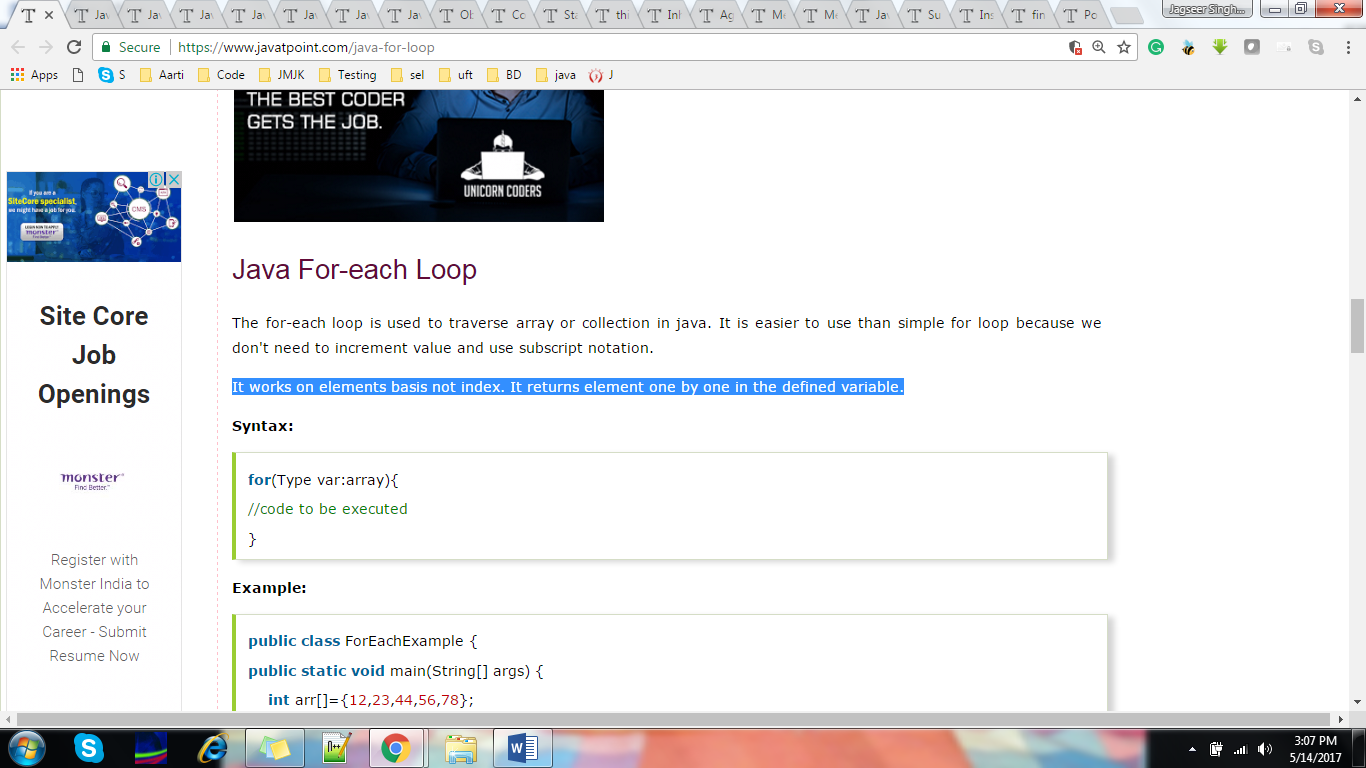
the Prinstream class belongs to java.io package

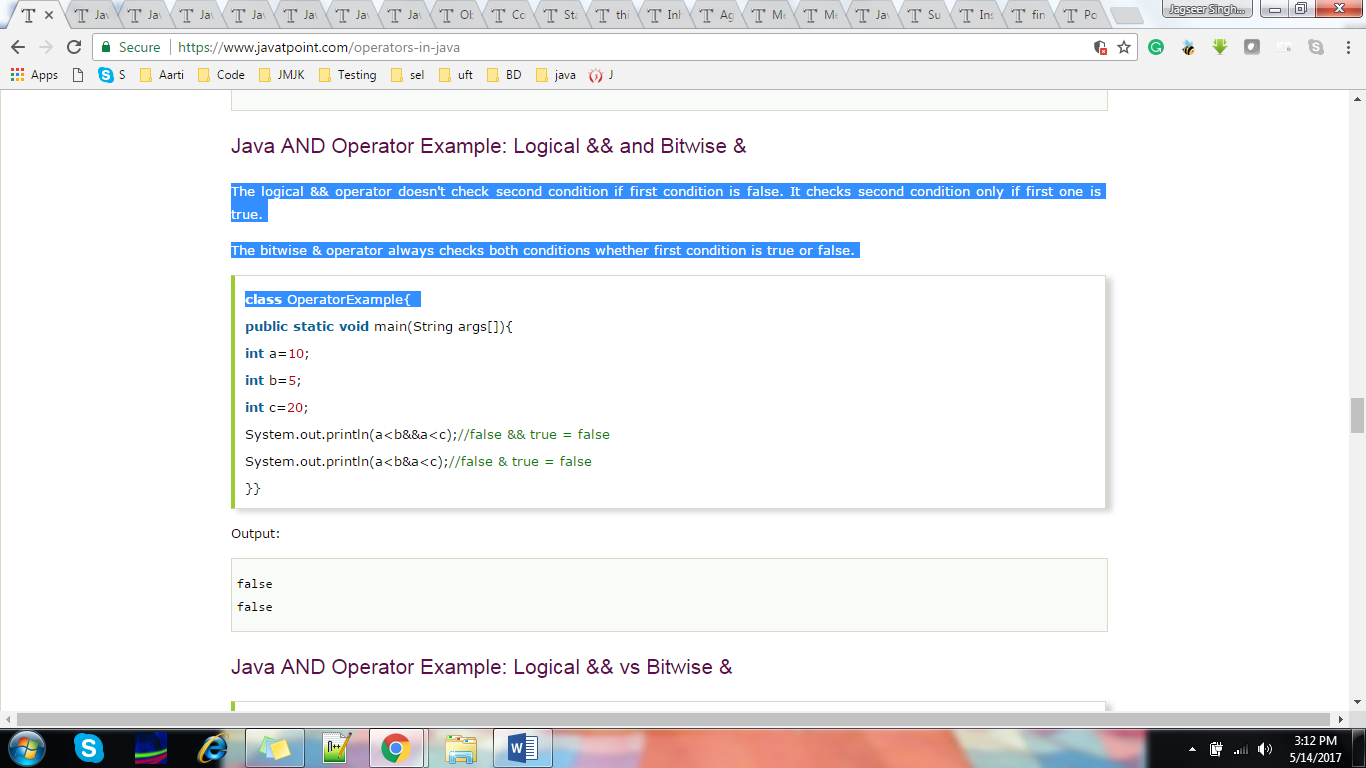
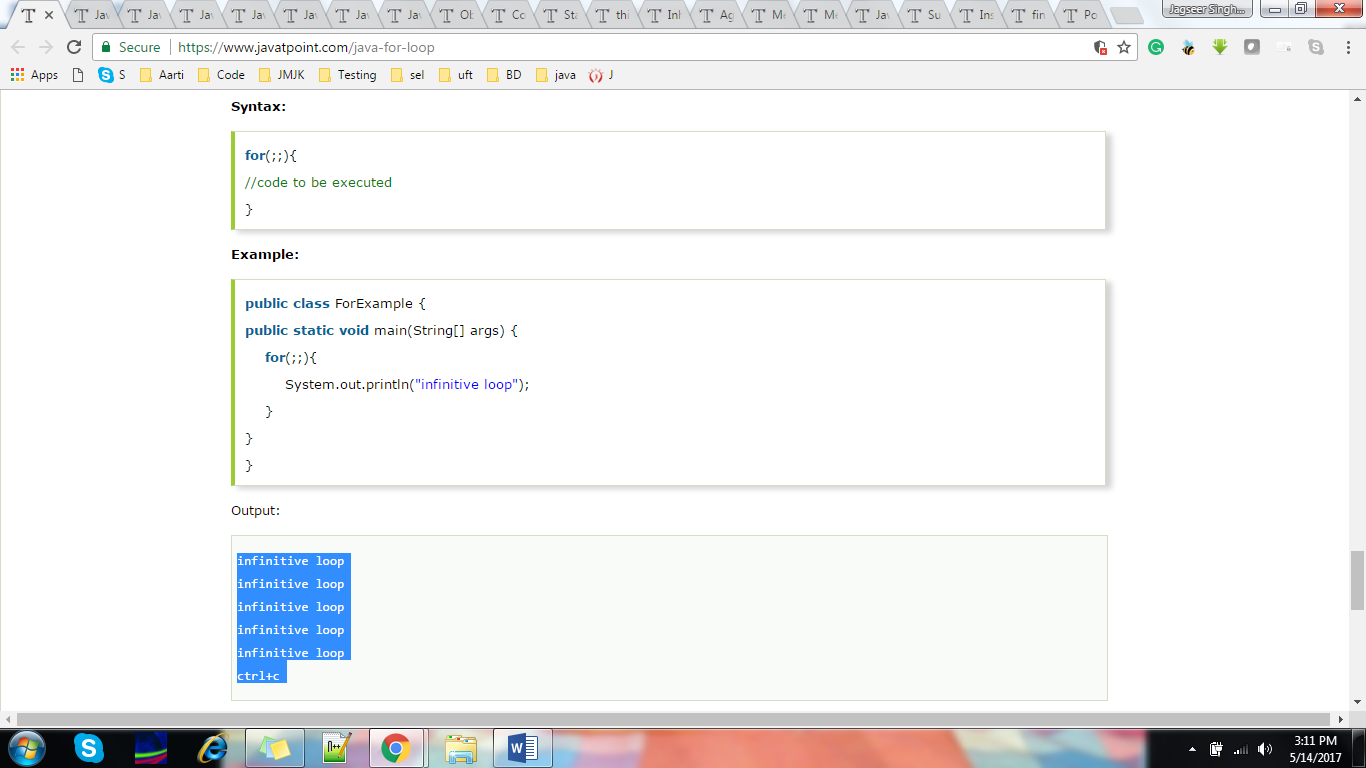
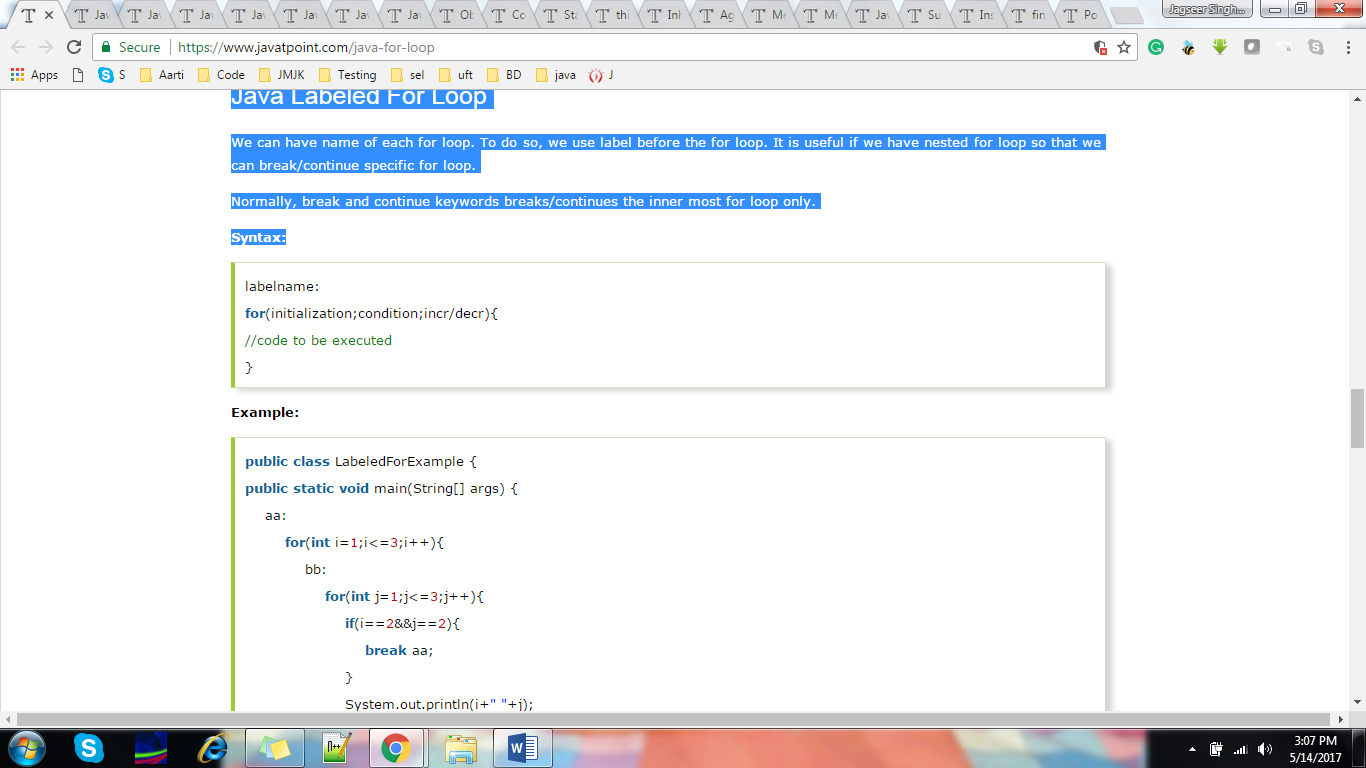
class PrintStream{

public void println();

//...

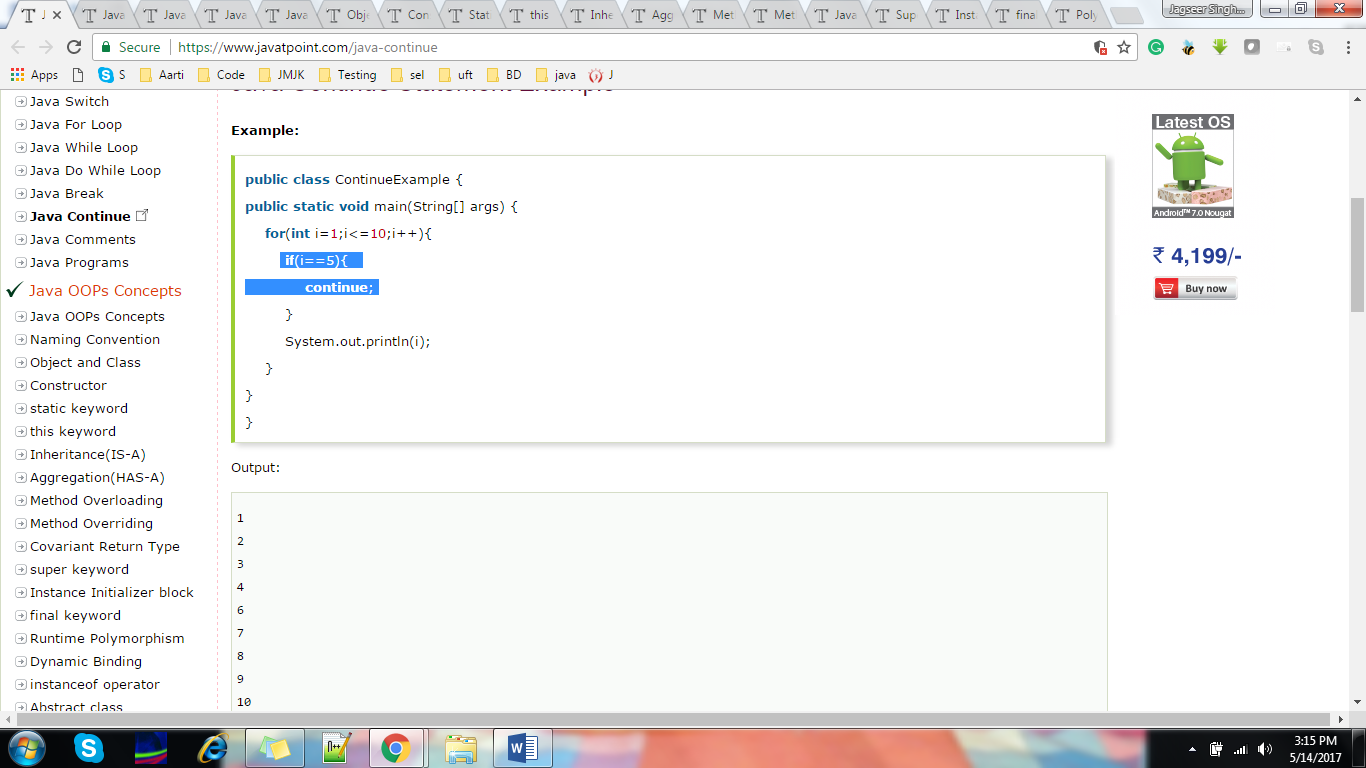
}

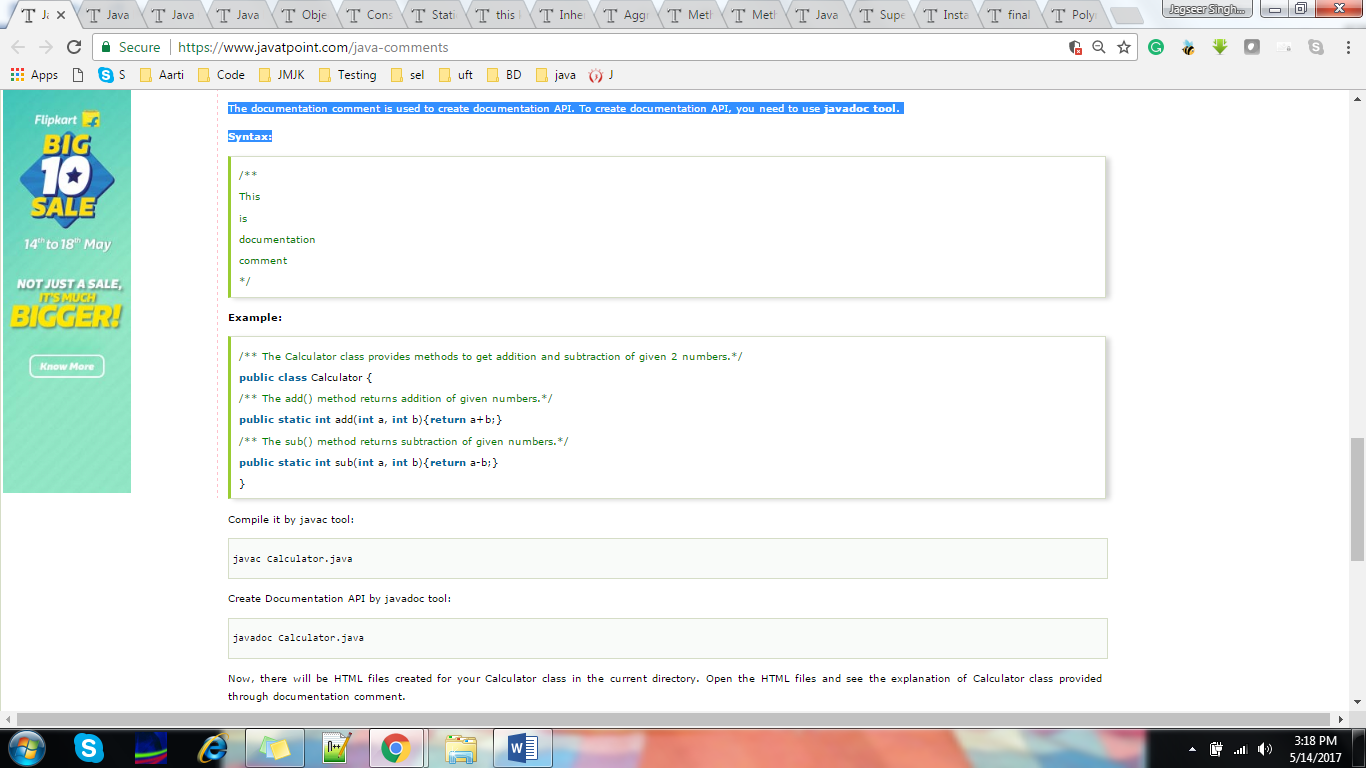




The Java continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

**Syntax:**





Object

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

Class

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

Inheritance

**When one object acquires all the properties and behaviours of parent object** i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.



Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

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

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

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.



Encapsulation

**Binding (or wrapping) code and data together into a single unit is known as encapsulation**. For example: 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.

S

## What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

* By new keyword
* By newInstance() method
* By clone() method
* By deserialization
* By factory method etc.

## Anonymous object

Anonymous simply means nameless. An object which has no reference is known as anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, anonymous object is a good approach. For example:

1. **new** Calculation();//anonymous object

## Java Copy Constructor

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

* By constructor
* By assigning the values of one object into another
* By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

### **Restrictions for static method**

|  |
| --- |
| There are two main restrictions for the static method. They are: |

|  |
| --- |
| 1. The static method can not use non static data member or call non-static method directly. 2. this and super cannot be used in static context. |

## Java static block

* Is used to initialize the static data member.
* It is executed before main method at the time of classloading.

#### 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.

### **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.

Inheritance in Java

where child object acquires all the properties and behaviors of parent object. The idea behind inheritance in java is the Reusability.

Multiple Inheritance and Hybrid Inheritance are not supported in Java using class. Why Multiple Inheritance is not supported in Java?

Dr James Gosling words on this

“JAVA omits many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than beneﬁt. This primarily consists of operator overloading (although it does have method overloading), multiple inheritance, and extensive automatic coercions.”

He has clearly explained the reason behind not supporting because of

1)To Retain Simplicity rather than making it complex and confusing

2)Very rare situation to get the need of it.

3) to avoid confusion of using which method from multiple parent classes.

### **Why use inheritance in java**

* For Method Overriding (so runtime polymorphism can be achieved).
* For Code Reusability.

#### Inheritance represents the **IS-A relationship**, also known as parent-child relationship. Multiple inheritance is not supported in java through class.

Relationship between two classes is **Programmer IS-A Employee**.It means that Programmer is a type of Employee.

1. **class** Employee{
2. **float** salary=40000;
3. }
4. **class** Programmer **extends** Employee{

#### Multiple inheritance is not supported in java through class.

# **Aggregation in Java**

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc.

### **When use Aggregation?**

* Code reuse is also best achieved by aggregation when there is no is-a relationship.
* Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

#### Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters

### **Can we overload java main() method?**

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only.

### **Usage of Java Method Overriding**

* Method overriding is used to provide specific implementation of a method that is already provided by its super class.
* Method overriding is used for runtime polymorphism

R**ules for Java Method Overriding**

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

## Advantage of method overloading

Method overloading increases the readability of the program.

### **Different ways to overload the method**

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

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

## Usage of java super Keyword

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

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.

**Instance Initializer block** is used to initialize the instance data member. It run each time when object of the class is created

Why use instance initializer block?

|  |
| --- |
| Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc. |

#### Note: The java compiler copies the code of instance initializer block in every constructor.

## Rules for instance initializer block :

|  |
| --- |
| There are mainly three rules for the instance initializer block. They are as follows: |

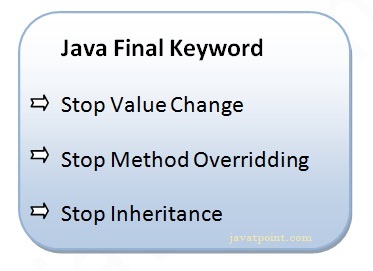
1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

### **Que) Can we initialize blank final variable?**

Yes, but only in constructor.

### **static blank final variable**

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.



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

#### Rule: Runtime polymorphism can't be achieved by data members

### **What is Runtime Polymorphism?**

Runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time.

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

### **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

An interface in java is a blueprint of a class. It has static constants and abstract methods. Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

**An interface is a contract: The guy writing the interface says, "hey, I accept things looking that way", and the guy using the interface says "OK, the class I write looks that way".**

## 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.

**If we don’t implement the all methods of interface then it shows error.**

### **What is abstraction?**

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

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

### **Can there be any abstract method without abstract class?**

No, if there is any abstract method in a class, that class must be abstract.

**methods of an interface is abstract by default, and static and abstract keywords can't be used together.**

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) **Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

## Marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

1. //How Serializable interface is written?
2. **public** **interface** Serializable{
3. }

### **Can we define private and protected modifiers for variables in interfaces?**

No, they are implicitly public.

## How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

If you want to keep the package within the same directory, you can use . (dot).

1. How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java mypack.Simple |

#### Note: If you import a package, subpackages will not be imported.

#### Note: Sequence of the program must be package then import then class.

#### The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

### **Ways to load the class files or jar files**

|  |
| --- |
| There are two ways to load the class files temporary and permanent. |

* Temporary
  + By setting the classpath in the command prompt
  + By -classpath switch
* Permanent
  + By setting the classpath in the environment variables
  + By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

#### Rule: There can be only one public class in a java source file and it must be saved by the public class name

### **How to put two public classes in a package?**

|  |
| --- |
| If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. |

# **Java Static Import**

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

# **Package class**

The package class provides methods to get information about the specification and implementation of a package. It provides methods such as getName(), getImplementationTitle(), getImplementationVendor(), getImplementationVersion() etc.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

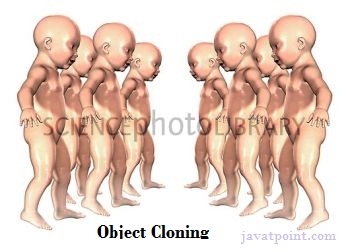
There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

### **Advantage of Encapsulation in java**

By providing only setter or getter method, you can make the class **read-only or write-only**.

# **Object Cloning in Java**

The **object cloning** is a way to create exact copy of an object. For this purpose, clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class. Syntax of the clone() method is as follows:

1. **protected** Object clone() **throws** CloneNotSupportedException

### **What is the class name of java array?**

In java, array is an object. For array object, an proxy class is created whose name can be obtained by getClass().getName() method on the object.

1. **int** arr[]={4,4,5};
2. Class c=arr.getClass();
3. String name=c.getName();

### **Copying a java array**

We can copy an array to another by the arraycopy method of System class.

 System.arraycopy(copyFrom, 2, copyTo, 0, 7);

**Java strictfp** keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable

### **Legal code for strictfp keyword**

The strictfp keyword can be applied on methods, classes and interfaces.

### **Illegal code for strictfp keyword**

The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.

1. **strictfp** **class** A{}//strictfp applied on class
2. **strictfp** **interface** M{}//strictfp applied on interface
3. **class** A{
4. **strictfp** **void** m(){}//strictfp applied on method
5. }

**To create the document API,** you need to use the javadoc tool followed by java file name. There is no need to compile the javafile.

On the command prompt, you need to write:

javadoc M.java

|  |
| --- |
| **Comment & Description** |
| 1 | **/\* text \*/**  The compiler ignores everything from /\* to \*/.  Ctrl+Shift+/ or \ |
| 2 | **//text**  The compiler ignores everything from // to the end of the line.  Ctrl+M |
| 3 | **/\*\* documentation \*/**  This is a documentation comment and in general its called **doc comment**. The **JDK javadoc** tool uses *doc comments* when preparing automatically generated documentation.  To quickly add a Javadoc use following shortcut:  **Windows**: alt + shift + J  **Or Simply type /\*\* then start typing it will gets auto extended to next lines** |

**The java String is immutable** i.e. it cannot be changed. Whenever we change any string, a new instance is created. **For mutable string,** you can use StringBuffer and StringBuilder classes.

### **String Literal**

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.

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

### **Why string objects are immutable in java?**

|  |
| --- |
| Because java uses the concept of string literal.Suppose there are 5 reference variables,all referes to one object "sachin".If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java. |

## String compare by == operator

The = = operator compares references not values

1. **class** Teststringcomparison3{
2. **public** **static** **void** main(String args[]){
3. String s1="Sachin";
4. String s2="Sachin";
5. String s3=**new** String("Sachin");
6. System.out.println(s1==s2);//true (because both refer to same instance)
7. System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
8. }
9. }

# **Java String format**

The **java string format()** method returns the formatted string by given locale, format and arguments.

If you don't specify the locale in String.format() method, it uses default locale by calling *Locale.getDefault()* method.

The format() method of java language is like *sprintf()* function in c language and *printf()* method of java language.

In java, **String concatenation is** implemented through the StringBuilder (or StringBuffer) class and its append method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For Example:

 String s="Sachin"+" Tendulkar";

String s=(**new** StringBuilder()).append("Sachin").append(" Tendulkar”).toString();

# **Java StringBuffer class**

Java StringBuffer class is used to created 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. multip le threads cannot access it simultaneously. So it is safe and will result in an order.

# **Java StringBuilder class**

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

## Important Constructors of StringBuilder class

### **Do I need to import java.lang package any time? Why ?**

No. It is by default loaded internally by the JVM.

### **Can I import same package/class twice? Will the JVM load the package twice at runtime?**

One can import the same package or same class multiple times. Neither compiler nor JVM complains about it.But the JVM will internally load the class only once no matter how many times you import the same class.

# **Exception –**

Checked exceptions are those exceptions which can be detected by compiler, and forces us to handle them. If we can’t handle them then compile time error will come.

Unchecked exceptions are those exceptions which can occur at runtime and compiler will not be having any clue of it. So compile time error will not come even if we don’t handle un checked exception(also called as RunTimeException)

**Checked Exception:** They are used to represent problems that can be recovered during the execution of the program. They usually are **not the programmer's fault**. For example, a file specified by user is not readable, or no network connection available, etc., In all these cases, our program doesn't need to exit, instead it can take actions like alerting the user, or go into a fallback mechanism(like offline working when network not available), etc.

**Unchecked Exceptions:** They again can be divided into two: **Errors and RuntimeExceptions**. One reason for them to be unchecked is that they are numerous in number, and required to handle all of them will clutter our programm and reduce its clarity. The other reason is:

* **Runtime Exceptions:** They are usually **happened due to programmers fault**. For example, if an ArithmeticException of division by zero occurs or an ArrayIndexOutOfBoundsExceptionoccurs, it is because we are not careful enough in our coding. They happen usually because some errors in our program logic. So, they must be cleared before our program enters into production mode. They are unchecked in the sense that, our program must fail when it occurs, so that we programmers can resolve it at the time of development and testing itself.
* **Errors:** Errors are **situations from which usually the program cannot recover**. For example, if a StackOverflowError occurs, our program cannot do much, such as increase the size of program's function calling stack. Or if an OutOfMemoryError occurs, we cannot do much to increase the amount of RAM available to our program. In such cases, it is better to exit the program. That is why they are made unchecked.

**Unchecked Exceptions:** need not be included in any method’s throws list because compiler does not check to see if a method handles or throws these exceptions.

**Unchecked**   
[ArrayIndexOutOfBoundsException](http://docs.oracle.com/javase/8/docs/api/java/lang/ArrayIndexOutOfBoundsException.html)   
[ClassCastException](http://docs.oracle.com/javase/8/docs/api/java/lang/ClassCastException.html)   
[IllegalArgumentException](http://docs.oracle.com/javase/8/docs/api/java/lang/IllegalArgumentException.html)   
[IllegalStateException](http://www.coderanch.com/how-to/java/illegalstateexception)   
[NullPointerException](http://docs.oracle.com/javase/8/docs/api/java/lang/NullPointerException.html)   
[NumberFormatException](http://docs.oracle.com/javase/8/docs/api/java/lang/NumberFormatException.html)   
[AssertionError](http://docs.oracle.com/javase/8/docs/api/java/lang/AssertionError.html)   
[ExceptionInInitializerError](http://docs.oracle.com/javase/8/docs/api/java/lang/ExceptionInInitializerError.html)   
[StackOverflowError](http://docs.oracle.com/javase/8/docs/api/java/lang/StackOverflowError.html)   
[NoClassDefFoundError](http://docs.oracle.com/javase/8/docs/api/java/lang/NoClassDefFoundError.html)   
  
**Checked**   
Exception   
[IOException](http://docs.oracle.com/javase/8/docs/api/java/io/IOException.html)   
[FileNotFoundException](http://docs.oracle.com/javase/8/docs/api/java/io/FileNotFoundException.html)   
[ParseException](http://docs.oracle.com/javase/8/docs/api/java/text/ParseException.html)   
[ClassNotFoundException](http://docs.oracle.com/javase/8/docs/api/java/lang/ClassNotFoundException.html)   
[CloneNotSupportedException](http://docs.oracle.com/javase/8/docs/api/java/lang/CloneNotSupportedException.html)   
[InstantiationException](http://docs.oracle.com/javase/8/docs/api/java/lang/InstantiationException.html)   
[InterruptedException](http://docs.oracle.com/javase/8/docs/api/java/lang/InterruptedException.html)   
[NoSuchMethodException](http://docs.oracle.com/javase/8/docs/api/java/lang/NoSuchMethodException.html)   
[NoSuchFieldException](http://docs.oracle.com/javase/8/docs/api/java/lang/NoSuchFieldException.html)

## Use of throws keyword in Java

1. The [**throws keyword**](http://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/) is used in method declaration, in order to explicitly specify the exceptions that a particular method might throw. When a method declaration has one or more exceptions defined using throws clause then the method-call must handle all the defined exceptions.  
2. When defining a method you must include a throws clause to [**declare those exceptions**](http://beginnersbook.com/2013/04/java-exception-handling/) that might be thrown but doesn’t get caught in the method.  
3. If a method is using throws clause along with few exceptions then this implicitly tells other methods that – “ If you call me, you must handle these exceptions that I throw”.

**Complete Example of Java throws Clause**

class Demo

{

static void throwMethod() throws NullPointerException

{

System.out.println ("Inside throwMethod");

throw new NullPointerException ("Demo");

}

public static void main(String args[])

{

try

{

throwMethod();

}

catch (NullPointerException exp)

{

System.out.println ("The exception get caught" +exp);

}

}

}

**The output of the above program is:**

Inside throwMethod

The exception get caught java.lang.IllegalAccessException: **Demo**

### **Which exception should be declared**

**Ans)** checked exception only, because:

* **unchecked Exception:** under your control so correct your code.
* **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

### **Advantage of Java throws keyword**

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

### Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

In order to **throw user defined exceptions**, [**throw keyword**](http://beginnersbook.com/2013/12/throw-keyword-example-in-java/) is being used.

try{ p.msg(); b=50/0; }catch(Exception e){

//throw new ArithmeticException ("Jagseer");

System.out.println(e+"value"+b); }

finally{System.out.println(b);System.out.println("JMJK");}

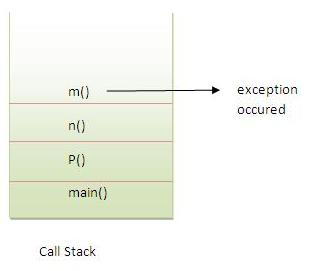
System.out.println("JMJK\_bye"); }

Output:

java.lang.ArithmeticException: / by zerovalue0  
0  
JMJK  
JMJK\_bye ((When handled so normal flow,if thrown then only finally block is executed)

#### Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

#### Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).



In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

void msg() **throws IOException**{ //if I Use this then exception is getting handled by try catch in calling function otherwise getting thrown here only.

b= 50/0;

throw new IOException (" JAi SHU JMJK");

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

}

Try to understand the difference between throws and throw keywords, ***throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.**

If a method does not handle a checked exception, the method must declare it using the **throws** keyword.

**If method doesn't want to catch it but want to give this responsibility to caller whoever is calling, then it throws the exception. I**

-when added throws then it got propegated to Main though throw was present

- when throws was not used then it was thrown using throw.

- when nothing was used throws or throw it got handled by main only.

- when in parent funtion didnot declare the excption it didnot allow the exceptin throw in child also

-child fun either same exception or specific than parent. Means child or overridden fun should have either no or less than parent in exceptions handling.

there are two methods to handle Exceptions.

1. Put the Exception causing code in try and catch block.
2. Declare the method to be throwing an Exception

If either of the above two is not done, compiler gives an error.

If you are catching an exception type, you do not need to throw it, unless you are going to rethrow it. In the example you post, the developer should have done one or another, not both.

* **catch an exception only if you can handle it in a meaningful way**
* **declare throwing the exception upward if it is to be handled by the consumer of the current method**
* **throw exceptions if they are caused by the input parameters (but these are more often unchecked)**

### **Case1: You handle the exception**

* In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

### **Case2: You declare the exception**

* A)In case you declare the exception, if exception does not occur, the code will be executed fine.
* B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

# **Exception tricky concepts.**

* Ground rule : try execution starts
* Case 1) : if exception does not occur then try continues the execution til the last line before the return statement in try block and then control goes to finally block, if finally does not have any return then control comes back to try’s return statement and hence completes. If finally has return statement then flow returns from finally it never reaches back to try’s return statement.
* Case 2): if exception occurs in try block then control jumps to catch block and if catch has return statement then flow excecutes until last line before return in catch , then control goes to finally, if finally has return then control returns to caller and if finally does not have return then control goes back to return statement of catch block.

# **Exception in inheritance**

* Ground rule 1(**parent** is throwing checked/unchecked)
  + **Child class’s** overridden method **is not forced to through** the exception.  
    However it can through the exception if it wants (rules will apply).
* Ground rule 2 : (**child** is throwing)
  + **Checked exception** then **parent must also through** the exception (same as parent or subclass exception) otherwise compilation fails.
  + **Unchecked exception** then **parent need not to throw** exception.

# **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.

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(){}

**public** **void** finalize(){System.out.println("object is garbage collected");}

**Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).**

#### Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

**public** **static** **void** gc(){}

System.gc();

#### Note: Neither finalization nor garbage collection is guaranteed.

# **Reflection:**

**Java Reflection** is a *process of examining or modifying the run time behavior of a class at run time*.

The **java.lang.Class** class provides many methods that can be used to get metadata, examine and change the run time behavior of a class.

The java.lang and java.lang.reflect packages provide classes for java reflection.

### **java.lang.Class class**

The java.lang.Class class performs mainly two tasks:

* provides methods to get the metadata of a class at run time.
* provides methods to examine and change the run time behavior of a class

### **How to get the object of Class class?**

There are 3 ways to get the instance of Class class. They are as follows:

* forName() method of Class class
* getClass() method of Object class
* the .class syntax

### **Commonly used methods of Class class:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public String getName() | returns the class name |
| 2) public static Class forName(String className)throws ClassNotFoundException   1. Class c=Class.forName("Simple"); 2. System.out.println(c.getName());   **OR using Object Class**   1. **void** printName(Object obj){ 2. Class c=obj.getClass(); 3. System.out.println(c.getName());   **OR using .Class**   1. Class c = **boolean**.**class**; 2. System.out.println(c.getName()); | loads the class and returns the reference of Class class. |
| 3) public Object newInstance()throws InstantiationException,IllegalAccessException | creates new instance. |
| 4) public boolean isInterface()   1. Class c=Class.forName("Simple"); 2. System.out.println(c.isInterface()); | checks if it is interface. |
| 5) public boolean isArray() | checks if it is array. |
| 6) public boolean isPrimitive() | checks if it is primitive. |
| 7) public Class getSuperclass() | returns the superclass class reference. |
| 8) public Field[] getDeclaredFields()throws SecurityException | returns the total number of fields of this class. |
| 9) public Method[] getDeclaredMethods()throws SecurityException | returns the total number of methods of this class. |
| 10) public Constructor[] getDeclaredConstructors()throws SecurityException | returns the total number of constructors of this class. |
| 11) public Method getDeclaredMethod(String name,Class[] parameterTypes)throws NoSuchMethodException,SecurityException | returns the method class instance. |

# **How to call private method from another class in java**

You can call the private method from outside the class by changing the runtime behaviour of the class.

By the help of **java.lang.Class** class and **java.lang.reflect.Method** class, we can call private method from any other class.

*File: A.java*

1. **class** A{
2. **private** **void** cube(**int** n){System.out.println(n\*n\*n);}
3. }

*File: M.java*

1. **import** java.lang.reflect.\*;
2. **class** M{
3. **public** **static** **void** main(String args[])**throws** Exception{
4. Class c=A.**class**;
5. Object obj=c.newInstance();
7. Method m=c.getDeclaredMethod("cube",**new** Class[]{**int**.**class**});
8. m.setAccessible(**true**);
9. m.invoke(obj,4);
10. }}

Output:64

# **Understanding javap tool**

The **javap command** disassembles a class file. **The javap command displays information about the fields,constructors and methods present in a class file.**

**C:\Users\Jagseer\Downloads\JMJK\src\SAM>javap SAM.Nad**

javap SAM.Nad

Compiled from "Nad.java"

public class SAM.Nad {

public SAM.Nad();

public void finalize();

public static void main(java.lang.String[]) throws java.lang.Exception;

}

# **Java JDBC Tutorial**

Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.

JDBC (Java Database Connectivity) 

# **Java List Interface**

List Interface is the subinterface of Collection.It contains methods to insert and delete elements in index basis.It is a factory of ListIterator interface.

### **List Interface declaration**

1. **public** **interface** List<E> **extends** Collection<E>

## Java ListIterator Interface

ListIterator Interface is used to traverse the element in backward and forward direction.

### **ListIterator Interface declaration**

1. **public** **interface** ListIterator<E> **extends** Iterator<E>
2. ListIterator<String> itr=al.listIterator();
3. System.out.println("traversing elements in forward direction...");  //do reverese for backward
4. **while**(itr.hasNext()){   //hasPrevious
5. System.out.println(itr.next());  //.previous()
6. }

### **Hierarchy of Collection Framework**

Let us see the hierarchy of collection framework.The **java.util** package contains all the classes and interfaces for Collection framework.

hierarchy of collection framework

### **Methods of Collection interface**

There are many methods declared in the Collection interface. They are as follows:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(Object element) | is used to insert an element in this collection. |
| 2 | **public boolean addAll(Collection c)** | is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | is used to delete an element from this collection. |
| 4 | **public boolean removeAll(Collection c)** | is used to delete all the elements of specified collection from the invoking collection. |
| 5 | **public boolean retainAll(Collection c)** | is used to delete all the elements of invoking collection except the specified collection. |
| 6 | public int size() | return the total number of elements in the collection. |
| 7 | public void clear() | removes the total no of element from the collection. |
| 8 | public boolean contains(Object element) | is used to search an element. |
| 9 | **public boolean containsAll(Collection c)** | is used to search the specified collection in this collection. |
| 10 | public Iterator iterator() | returns an iterator. |
| 11 | public Object[] toArray() | converts collection into array. |
| 12 | public boolean isEmpty() | checks if collection is empty. |
| 13 | public boolean equals(Object element) | matches two collection. |
| 14 | public int hashCode() | returns the hashcode number for collection. |

**Wherever \*All is used in method name, that deals with collection.**

# **Java ArrayList class**

Java ArrayList class hierarchy

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

**public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

The important points about Java ArrayList class are:

* Java ArrayList class can contain **duplicate elements.**
* **Java ArrayList class maintains insertion order.**
* Java ArrayList class is **non synchronized.**
* Java ArrayList allows **random access because array works at the index basis.**
* In Java ArrayList class**, manipulation is slow** because a lot of shifting needs to be occurred if any element is removed from the array list.

**Linked List (Rest of above are same)**

java.util.LinkedList class.

**public** **class** LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Cloneable, Serializable

* **Linked list** **manipulation is fast**
* **Java LinkedList class can be used as list, stack or queue.**

### **ArrayList class declaration**

Let's see the declaration for java.util.ArrayList class.

1. **public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable

### **Java Non-generic Vs Generic Collection**

**Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.**

Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.

Let's see the old non-generic example of creating java collection.

1. ArrayList al=**new** ArrayList();//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

1. ArrayList<String> al=**new** ArrayList<String>();//creating new generic arraylist

### **Two ways to iterate the elements of collection in java**

There are two ways to traverse collection elements:

1. By Iterator interface.
2. By for-each loop.

 ArrayList<String> al=**new** ArrayList<String>();

**for**(String obj:al)

    System.out.println(obj)

Iterator itr=al.iterator();

**while**(itr.hasNext()){

    Student st=(Student)itr.next();

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

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

# **ava HashSet class**

Java HashSet class hierarchy

Java HashSet class is used to create a collection **that uses a hash table for storage**.

* HashSet stores the elements by using a mechanism called **hashing.**

## Difference between List and Set

List can contain duplicate elements whereas **Set contains unique elements only**.

1. **public** **class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable

# **Java LinkedHashSet class**

Java HashSet class hierarchy

Java LinkedHashSet class is a Hash table and Linked list implementation of the set interface. **It inherits HashSet class and implements Set interface.**

* Contains unique elements only like HashSet.
* Provides all optional set operations, and permits null elements.
* **Maintains insertion order.**

# **Java TreeSet class**

TreeSet class hierarchy

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Contains **unique** elements only like HashSet.
* **Access and retrieval times are quiet fast.**
* **Maintains ascending order.**

The elements in TreeSet must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement Comparable interface.

1. **import** java.util.\*;
2. **class** Book **implements** Comparable<Book>{

### **Methods of Java Queue Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(object) | It is used to insert the specified element into this queue and return true upon success. |
| boolean offer(object) | It is used to insert the specified element into this queue. |
| Object remove() | It is used to retrieves and **removes the head of this queue**. |
| Object poll() | It is used to retrieves and removes the head of this queue, or returns null if this queue is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this queue. |
| Object peek() | It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |

## PriorityQueue class

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.

1. **public** **class** PriorityQueue<E> **extends** AbstractQueue<E> **implements** Serializable

 Queue<Book> queue=**new** PriorityQueue<Book>();

PriorityQueue<String> queue=**new** PriorityQueue<String>();

**The elements in PriorityQueue must be of Comparable type.**

# **Java Deque Interface**

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

## Deque Interface declaration

1. **public** **interface** Deque<E> **extends** Queue<E>

## ArrayDeque class

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

* Unlike Queue, we can add or remove elements from both sides.
* Null elements are not allowed in the ArrayDeque.
* ArrayDeque is not thread safe, in the absence of external synchronization.
* ArrayDeque has no capacity restrictions.
* ArrayDeque is faster than LinkedList and Stack.

java arraydeque hierarchy

  Deque<String> deque=**new** ArrayDeque<String>();

deque.add("mukul");

deque.offerFirst("jai");

//deque.poll();

    //deque.pollFirst();//it is same as poll()

    deque.pollLast();  //removes the last

# **Java Map Interface**

A map **contains values on the basis of key i.e. key and value pair**. Each key and value pair is known as an entry. **Map contains only unique keys.**

Map is useful if you have **to search, update or delete elements on the basis of key**.

### **Useful methods of Map interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object put(Object key, Object value) | It is used to insert an entry in this map. |
| void putAll(Map map) | It is used to insert the specified map in this map. |
| Object remove(Object key) | It is used to delete an entry for the specified key. |
| Object get(Object key) | It is used to return the value for the specified key. |
| boolean containsKey(Object key) | It is used to search the specified key from this map. |
| Set keySet() | It is used to return the Set view containing all the keys. |
| **Set entrySet()** | **It is used to return the Set view containing all the keys and values.** |

## Map.Entry Interface

Entry is the **sub interface of Map**. So we will be accessed it by **Map.Entry name**. It provides methods to get key and value.

### **Methods of Map.Entry interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object getKey() | It is used to obtain key. |
| Object getValue() | It is used to obtain value. |

1. Map<Integer,String> map=**new** HashMap<Integer,String>();
2. map.put(100,"Amit");
3. **for**(Map.Entry m:map.entrySet()){
4. System.out.println(m.getKey()+" "+m.getValue());
5. }

### **Difference between HashSet and HashMap**

HashSet contains only values whereas HashMap contains entry(key and value).

# **Java HashMap class**

Java HashMap class hierarchy

Java HashMap class implements the map interface by using a hashtable.

* A HashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It maintains no order.

# **Java LinkedHashMap class**

* It is same as HashMap instead maintains insertion order.

# **Java TreeMap class**

It is same as HashMap instead maintains ascending order

### **What is difference between HashMap and TreeMap?**

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| 1) HashMap can contain one null key. | TreeMap can not contain any null key. |
| 2) HashMap maintains no order. | TreeMap maintains ascending order. |

# **Java Hashtable class**

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

The important points about Java Hashtable class are:

* A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* It contains only unique elements.
* It may have not have any null key or value.
* It is synchronized.

### **Hashtable class declaration**

Let's see the declaration for java.util.Hashtable class.

1. **public** **class** Hashtable<K,V> **extends** Dictionary<K,V> **implements** Map<K,V>, Cloneable, Serializable

# **Difference between HashMap and Hashtable**

HashMap and Hashtable both are used to store data in key and value form. Both are using hashing technique to store unique keys.

But there are many differences between HashMap and Hashtable classes that are given below.

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

Java EnumSet class

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

**public** **abstract** **class** EnumSet<E **extends** Enum<E>> **extends** AbstractSet<E> **implements** Cloneable, Serializable

1. **enum** days {
2. SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
3. }
4. Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
5. Set<days> set1 = EnumSet.allOf(days.**class**);
6. System.out.println("Week Days:"+set1);
7. Set<days> set2 = EnumSet.noneOf(days.**class**);
8. System.out.println("Week Days:"+set2);

# **Java EnumMap class**

Java EnumMap class is the specialized Map implementation for enum keys. It inherits Enum and AbstractMap classes.

**public** **class** EnumMap<K **extends** Enum<K>,V> **extends** AbstractMap<K,V> **implements** Serializable, Cloneable

 EnumMap<Days, String> map = **new** EnumMap<Days, String>(Days.**class**);

map.put(Days.Monday, "1");

1. **for**(Map.Entry m:map.entrySet()){
2. System.out.println(m.getKey()+" "+m.getValue());
3. }

# **Java Collections class**

Java collection class is used exclusively with static methods that operate on or return collections. It inherits Object class.

The important points about Java Collections class are:

* Java Collection class supports the **polymorphic algorithms** that operate on collections.
* Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.

Java.util.Collections class.

**public** **class** Collections **extends** Object

1. List<String> list = **new** ArrayList<String>();
2. list.add("C");
3. Collections.addAll(list, "Servlet","JSP");
4. String[] strArr = {"C#", ".Net"};
5. Collections.addAll(list, strArr);

**Other func:**

Collections.max(list));

Collections.min(list));

# **Sorting in Collection**

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

|  |
| --- |
| **Collections** class provides static methods for sorting the elements of collection.If collection elements are of Set type, we can use TreeSet.But We cannot sort the elements of List.Collections class provides methods for sorting the elements of List type elements. |
| **public void sort(List list):** is used to sort the elements of List.List elements must be of Comparable type. |

#### Note: String class and Wrapper classes implements Comparable interface by default. So if you store the objects of string or wrapper classes in list, set or map, it will be Comparable by default.

# **Java Comparable interface**

Java Comparable interface is used to order the objects of user-defined class.This interface is found in java.lang package and contains only one method named compareTo(Object). It provide single sorting sequence only i.e. you can sort the elements on **based on single data member only**. For example it may be rollno, name, age or anything else.

### **compareTo(Object obj) method**

**public int compareTo(Object obj):** is used to compare the current object with the specified object.

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

EX.

1. **public** **int** compareTo(Student st){
2. **if**(age==st.age)
3. **return** 0;
4. **else** **if**(age>st.age)
5. **return** 1;
6. **else**
7. **return** -1;
8. }

# **Difference between Comparable and Comparator**

Comparable and Comparator both are interfaces and can be used to sort collection elements.

But there are many differences between Comparable and Comparator interfaces that are given below.

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc. | Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc. |
| 2) Comparable **affects the original class** i.e. actual class is modified. | Comparator **doesn't affect the original class** i.e. actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is found in **java.lang** package. | Comparator is found in **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method. |

# **Java Comparator interface**

It **provides multiple sorting sequence i**.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.

#### compare() method

**public int compare(Object obj1,Object obj2):** compares the first object with second object.

## Collections class

**Collections** class provides static methods for sorting the elements of collection.

**public void sort(List list, Comparator c):**

1. **class** AgeComparator **implements** **Comparator**{
2. **public** **int compare(Object o1,Object o2){**
3. Student s1=(Student)o1;
4. Student s2=(Student)o2;
6. **if**(s1.age==s2.age)
7. **return** 0;
8. **else** **if**(s1.age>s2.age)
9. **return** 1;
10. **else**
11. **return** -1;
12. }
13. }
14. **Collections.sort(al,new AgeComparator());**

# **Difference between ArrayList and Vector**

ArrayList and Vector both implements List interface and maintains insertion order.

But there are many differences between ArrayList and Vector classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity. | Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object. |
| 5) ArrayList uses **Iterator** interface to traverse the elements. | Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also. |

# **Properties class in Java**

The **properties** object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

It can be used to get property value based on the property key. The Properties class provides methods to get data from properties file and store data into properties file. Moreover, it can be used to get properties of system.

### **Advantage of properties file**

**Recompilation is not required, if information is changed from properties file:** If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

**b.properties**

1. user=system
2. password=oracle
3. FileReader reader=**new** FileReader("db.properties");
4. Properties p=**new** Properties();
5. p.load(reader);
6. System.out.println(p.getProperty("user"));
7. System.out.println(p.getProperty("password"));

### **get all the system properties**

1. Properties p=System.getProperties();
2. Set set=p.entrySet();
3. Iterator itr=set.iterator();
4. **while**(itr.hasNext()){
5. Map.Entry entry=(Map.Entry)itr.next();
6. System.out.println(entry.getKey()+" = "+entry.getValue());

### **to create properties file**

1. Properties p=**new** Properties();
2. p.setProperty("name","Sonoo Jaiswal");
3. p.setProperty("email","sonoojaiswal@javatpoint.com");
5. p.store(**new** FileWriter("info.properties"),"Javatpoint Properties Example");

meaning of synchronized

i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.

thread safe

1. In Java **synchronized**is the modifier applicable for blocks and methods. We **can’t**declare classes and variables as **synchronized**.
2. If a method or block declared as **synchronized**then at a time only one thread can allowed to operate on the given object.
3. The main **advantage**of synchronized keyword is we can **resolve data inconsistency problems.** But the main **disadvantage**of synchronized keyword is it**increases waiting time of thread and effects performance of the system.**Hence, if there is no specific requirement it is never recommended to use synchronized keyword.

# **Java I/O Tutorial**

Read data from file Line by line

**import** java.io.BufferedReader;

**import** java.io.File;

**import** java.io.FileReader;

**import** java.io.IOException;

**public** **class** readFlatFile {

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

{

File fle = **new** File("E:\\My Data\\JMJK\\selenium\\Java Workspace\\misc\\Sample.txt");

FileReader fr = **new** FileReader(fle);

BufferedReader br = **new** BufferedReader(fr);

String strLine;

**while** ((strLine = br.readLine())!= **null**)

{

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

}

br.close();

fr.close();

}

}

**File**

Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion, etc.

The File object represents the actual file/directory on the disk.

File f = null;

String[] strs = {"test1.txt", "test2.txt"};

try {

for(String s:strs ) {

f = new File(s);

boolean bool = f.canExecute();

String a = f.getAbsolutePath(); }

## Reading and Writing Files

Java FileOutputStream is an output stream used for writing data to a file.

If you have to **write primitive values** into a file, use **FileOutputStream class**. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for **character-oriented data**, it is preferred to **use FileWriter** than FileOutStream.

### **Byte Streams (**8-bit bytes)

**FileInputStream, FileOutputStream**.

InputStream f = new FileInputStream("C:/java/hello"); OR

InputStream f = new FileInputStream(new File("C:/java/hello")); OR

FileInputStream in =new FileInputStream("input.txt");

FileInputStream in =new FileInputStream("input.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

### **Character Streams (**for 16-bit unicode.)

**FileReader** and **FileWriter**

🡪 Internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

FileReader in = null;

FileWriter out = null;

try {

in = new FileReader("input.txt");

out = new FileWriter("output.txt");

int c;

while ((c = in.read()) != -1) {

out.write(c);

other methods

/ Creates a FileWriter Object

File file = new File("Hello1.txt");

file.createNewFile();

FileWriter writer = new FileWriter(file);

writer.write("This\n is\n an\n example\n");

writer.flush();

writer.close();

/ Creates a FileReader Object

FileReader fr = new FileReader(file);

char [] a = new char[50];

fr.read(a); // reads the content to the array

for(char c : a)

System.out.print(c); // prints the characters one by one

Java provides the following three standard streams −

* **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
* **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
* **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.
* InputStreamReader cin = new InputStreamReader(**System.in**);
* char c;
* do {
* c = (char) cin.read();
* System.out.print(c);
* } while(c != 'q');

# **BufferedReader Class**

read the text from a character-based input stream. It can be used **to read data line by line by readLine() method**. It makes the performance fast. It inherits Reader class.

1. FileReader fr=**new** FileReader("D:\\testout.txt");
2. BufferedReader br=**new** BufferedReader(fr);
3. **int** i;
4. **while**((i=br.read())!=-1){           System.out.print((**char**)i);            }

## InputStreamReader and BufferedReader

1. InputStreamReader r=**new** InputStreamReader(System.in);
2. BufferedReader br=**new** BufferedReader(r);
3. String name=br.readLine();
4. System.out.println("Welcome "+name);

# **BufferedWriter Class**

1. FileWriter writer = **new** FileWriter("D:\\testout.txt");
2. BufferedWriter buffer = **new** BufferedWriter(writer);
3. buffer.write("Welcome to javaTpoint.");

# **ObjectOutputStream / ObjectInputStream**

writes the specified object to the ObjectOutputStream.

FileOutputStream out = new FileOutputStream("test.txt");

ObjectOutputStream oout = new ObjectOutputStream(out);

oout.writeObject(s);

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("test.txt"));

System.out.println("" + (String) ois.readObject());

# How to read file in Java – BufferedInputStream

1) Created a File instance by providing the full path of the file(which we will read) during File Object creation.  
2) Passed the file instance to the [FileInputStream](http://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html#FileInputStream(java.io.File)) which opens a connection to the actual file, the file named by the File object file in the file system.  
3) Passed the FileInputStream instance to [BufferedInputStream](http://docs.oracle.com/javase/7/docs/api/java/io/BufferedInputStream.html#BufferedInputStream(java.io.InputStream)) which creates a BufferedInputStream and saves its argument, the input stream in, for later use. An internal buffer array is created and stored in buf using which the read operation gives good performance as the content is readily available in the buffer.  
4) Used while loop to read the file. Method [available()](http://docs.oracle.com/javase/7/docs/api/java/io/BufferedInputStream.html#available()) is used for checking the end of the file as it returns 0 when the pointer reaches to the end of the file. Read the file content using [read()](http://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html#read()) method of FileInputStream.

package beginnersbook.com;

import java.io.\*;

public class ReadFileDemo {

public static void main(String[] args) {

//Specify the path of the file here

File file = new File("C://myfile.txt");

BufferedInputStream bis = null;

FileInputStream fis= null;

try

{

//FileInputStream to read the file

fis = new FileInputStream(file);

/\*Passed the FileInputStream to BufferedInputStream

\*For Fast read using the buffer array.\*/

bis = new BufferedInputStream(fis);

/\*available() method of BufferedInputStream

\* returns 0 when there are no more bytes

\* present in the file to be read\*/

while( bis.available() > 0 ){

System.out.print((char)bis.read());

}

}catch(FileNotFoundException fnfe)

{

System.out.println("The specified file not found" + fnfe);

}

catch(IOException ioe)

{

System.out.println("I/O Exception: " + ioe);

}

finally

{

try{

if(bis != null && fis!=null)

{

fis.close();

bis.close();

}

}catch(IOException ioe)

{

System.out.println("Error in InputStream close(): " + ioe);

}

}

}

}

### **Creating Directories**

**mkdir( )** method creates a directory,

String dirname = "/tmp/user/java/bin";

File d = new File(dirname);

d.mkdirs();

**Mkdirs()** method creates both a directory and all the parents of the directory.

Java Regex

#### java.util.regex package

1. MatchResult interface
2. Matcher class
3. Pattern class
4. PatternSyntaxException class

## Matcher class

It implements **MatchResult** interface. **It is a *regex engine*** i.e. used to perform match operations on a character sequence.

## Pattern class

It is the *compiled version of a regular expression*. It is used to **define a pattern for the regex engine.**

The . (dot) represents a single character.

1. //1st way
2. Pattern p = Pattern.compile(".s");//. represents single character
3. Matcher m = p.matcher("as");
4. **boolean** b = m.matches();
6. //2nd way
7. **boolean** b2=Pattern.compile(".s").matcher("as").matches();
9. //3rd way  eASY
10. **boolean b3 = Pattern.matches(".s", "as");**

**See more examples at** [**https://www.javatpoint.com/java-regex**](https://www.javatpoint.com/java-regex)

# Java Inner Class

**Java inner class** or nested class is a class i.e. declared inside the class or interface.

### **Advantage**

1. it can access all the members (data members and methods) of outer class including private.
2. to develop more readable and maintainable code
3. Code Optimization:

### **Types of Nested classes**

1. Non-static nested class(**inner class)**
   * a)Member inner class
   * b)Annomynous inner class
   * c)Local inner class
2. Static **nested class**

# Java Member inner class

A non-static class that is created inside a class but outside a method is called member inner class.

# Java Anonymous inner class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

How many ways we can create object in java

Using new keyword  
This is the most common way to create an object in java. Almost 99% of objects are created in this way.

MyObject object = new MyObject();

Using Class.forName()  
If we know the name of the class & if it has a public default constructor we can create an object in this way.

MyObject object = (MyObject) Class.forName("subin.rnd.MyObject").newInstance()

Using clone()  
The clone() can be used to create a copy of an existing object.

MyObject anotherObject = new MyObject();

MyObject object = (MyObject) anotherObject.clone();

Using object deserialization  
Object deserialization is nothing but creating an object from its serialized form.

ObjectInputStream inStream = new ObjectInputStream(anInputStream );

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

# **Upcasting and downcasting in Java**

Casting : taking an object of one type and assigning it to reference variable of another type.

Upcasting : Object of child class is assigned to reference varibale of parent type.

Ex: class A{}  
class B extend A{}

A a1 = new B();  
this operation is a upcasting. and it happens automatically.no need to do anything explicitly.

here we can call the methods defined/declared in A but during runtime it will call class B’s overridden methods.  
if the method is not overridden in child’s class then only parent’s method which will be inherited to child will be called.

but same is not applicable to variables because variables decision happens at a compile time so always class A’s variables (not child’s inherited variables) will be accessed.

🡪x is 10 in parent and 20 in child, even though child has overridden ‘x’ but variable of parent is choosen.

**Downcasting : is not directly possible in Java.**

|  |
| --- |
| Child c = **new** Parent(); |

it is compile time error. because all the members of child class are not available in parent,

|  |  |
| --- | --- |
| 2 | Parent p = **new** Child();  Child c =(hild) p; |

downcasting will not happen automatically like upcasting,we have to cast it explicitly like we did above (Child)p.

it is putting parent’s reference variable holding child’s object to child’s reference variable.

it is almost equivalent to

|  |  |
| --- | --- |
| 1 | Child c = **new** Child(); |

but indirectly using parent’s reference variable as temp.

**Whenever upcasting happens always remember**

**1)parent’s variables will be accessed**

**2)child’s methods(overridden methods if overriding happened else inherited methods as it is from parent) will be called.**

**Reason - Variable binding happens at compile time unlike method binding which happens at run time.**

# **Why is Downcasting?**

Downcasting is used more frequently than upcasting. Use downcasting when we want to access specific behaviors of a subtype.

**QUICK CONCEPTS-**

[**http://javainsimpleway.com/**](http://javainsimpleway.com/)

**Variable hiding** happens when there is another variable with the **same name** and it has **nearest scope**.

Ex. When local and global variables have the same name, local variables will take precedence than

global variables.

variables in the parent class gets inherited to the child class but just that they will be **hidden** as we declared the variables with the same name in the child class.

## ****Key points to remember****

Variable hiding happens when there is another variable with the **same name**

By default Java compiler adds **this** to the variable name (only when there is no local variable with the same name)

When the instance variable is hidden by the local variable, use **this** explicitly to access the instance variable

# **Compiler code**

When we write a program in Java, we know that it gets converted to byte code, but before that compiler adds some **intelligent code** into our code.

**Lets see what it can add**

1) Extends **Object** to our class

**public** **class** CompilerCodeDemo **extends** Object {

**int** x;

2) **Default constructor** if there are no constructor defined for a class

**public** CompilerCodeDemo(CompilerCodeDemo **this**) {

**super**(**this**);

    }

3)Adds **this** argument to the constructor

**public** Example2(Example2 **this**,**int** x,**float** y) {

**super**(**this**);

4)Adds **super** statement inside constructor as first statement (since there is no **super** or **this** exist already)

# **this keyword in Java**

**1) this** always points to the **current object**

System.out.println("Object1 calling method1 "+obj1);

obj1.method1();

System.out.println("this keyword points to "+**this**);

**2) this** can be used with instance variables to overcome variable hiding

**3) this** can be used to call the **overloaded constructor** of the same class

**public** Person() {

**this**("javainsimpleway",20);

    }

Remember **this** is not allowed to make **recursive** constructor call.

**4) this** can be used inside the instance methods to call another instance method of the same class

**5) this** can be passed as a method parameter

**public** **void** m1(){

        System.out.println("m1");

        m2(**this**);

    }

# **SIB,IIB and Examples on SIB and IIB**

SIB –Static Initialization Block  
IIB – Instance Initialization Block

**public** **class** SIB1 {

    // SIB : It executes when the class gets loaded and executes only once in entire execution

**static** {

        System.out.println("SIB");

    }

    // IIB : It executes when the constructor is called but before the execution of constructor. So it executes as many times as constructor gets executed.

    {

        System.out.println("IIB");

    }

**public** SIB1() {

        System.out.println("Constructor");

    }

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

SIB1 sib = **new** SIB1();

    }

}

What if more than one SIB and IIB blocks are in the same class ?

All SIB blocks gets executes first and then all IIB blocks executes and finally constructor executes.

**Notable points about constructor in Java**

1) Default constructor is added automatically for any class in java  
2) If there is any parameterized constructor exist then default constructor is not added automatically.  
3)super() call to super class default constructor **is the default first statement of any constructor** if there is no this or super call explicitely.  
4) Both ‘super’ and ‘this’ can never be used for single constructor, any one of them is allowed.

5) ‘super’ and ‘this’ should be the first statement inside a constructor.

Note:  
If the subclass constructor does not specify which super class constructor to invoke then the compiler will automatically call the accessible no-args constructor in the super class.

No matter whether subclass constructor has ‘this’ call or not, super call will always be there internally.

Constructor Ground Rules

* 1. Recursive constructor call is not allowed.

1. **public** Example3() {
2. **this**(); //this point to same default constructor so it is calling the same constructor
3. System.out.println("inside Example3 constructor");
4. }

2) Always constructor first statement should be ‘this’ or ‘super’ but never be both.

# **Singleton Class**

The Singleton's purpose is to control object creation, limiting the number of objects to only one. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields. Singletons often control access to resources, such as database connections or sockets.

For example, if you have a license for only one connection for your database or your JDBC driver has trouble with multithreading, the Singleton makes sure that only one connection is made or that only one thread can access the connection at a time.

## Implementing Singletons

The easiest implementation consists of a private constructor and a field to hold its result, and a static accessor method with a name like getInstance().

Singleton tmp = Singleton.getInstance( );

tmp.demoMethod( );

# **Access Modifiers**

**1) Private Access Modifier**

It specifies the accessibility within a class. And when we try to access outside the class, it gives compile time error.

So ensure to use private members only within a class.

**Can we make Constrictor as private ? what happens if we make it private ?**

Yes absolutely fine to make constructor as private.  
If we make constructor as private, we can’t create an object of a class outside the class.

This feature of making constructor as private helps **in achieving Singleton pattern**.

(constructor ABC() is not visible->(fix by making it default or public) otherwise at runtime-Unresolved compilation problem)

**class** ABC

{

**private** ABC(){

System.***out***.println("prvt ABC");

}

ABC (**int** q){

**this**(); /// accessing the private default contructor

System.***out***.println("constrtr"+q);

}

}

**public** **class** JPrac {

**private** JPrac(){

System.***out***.println("private contr");

}

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

{

// **TODO** Auto-generated method stub

JPrac jp = **new** JPrac();

ABC abc= **new** ABC(2);

System.***out***.println("sdf");

}

}

**2) Default Access Modifier**

If we don’t specify any modifier, then java adds a default modifier to it.  
And its scope is the package where the class resides. Hence not accessible outside the package.

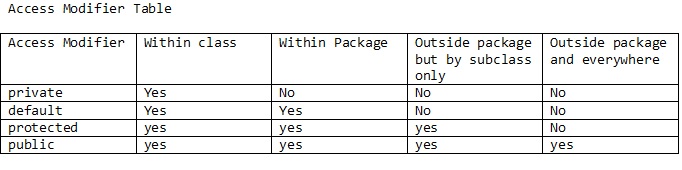
**3) Protected Access Modifier**

It is accessible within and outside the package only through Inheritance

This modifier can be applied on class attributes and methods and cannot be applied on a class (because we inherit class features not class itself). it gives compile time error when we try to access protected member ‘

**4) Public Access Modifier**

This can be accessible anywhere in the application and has no restriction to access.

Note:  
Whenever we override a method in the sub class, the sub class method should not narrow the access modifier specified in the super class however it can widen it.

This means if method is protected in super class then it cannot be private/default in subclass however it can be public in subclass.

# **Interfaces**

It is one of the ways to achieve abstraction in Java.

It will have only method declaration(abstract methods) and constant attributes in it.

It cannot be instantiated like how we can’t instantiate abstract class.

They are used to achieve multiple inheritance and polymorphism.

|  |
| --- |
| **public** **interface** Hello{  String str = ”hello”;  **void** sayHello();  } |

Note : All the variables inside Interface are **public , static and final** even if we don’t specify anything. we can access it directly using interface name.

All methods are **public and abstract** even if we don’t specify anything.

Also we **can’t change** these default access modifiers.

# [**How to rename a class and its corresponding file in Eclipse**](https://stackoverflow.com/questions/5178411/how-to-rename-a-class-and-its-corresponding-file-in-eclipse)

Shift + alt + r (Right click file ->refactor ->rename) when cursor is on class name. The file and constructors will be also changed.

# **Multithreading**

It says multiple tasks can be done at a time but actually it is not exactly like that.  
It will keep the cpu idle time less. Because cpu will have a responsibility to execute many threads if we Use multithreading.

**Normal Class** **:** A Java class

**Java Beans** **:**

- All properties private (use getters/setters)

- A public no-argument constructor

- Implements Serializable.

**Pojo :**

Plain Old Java Object is a Java object not bound by any restriction other than those forced by the Java Language Specification. I.e., a POJO should not have to

Extend prespecified classes

Implement prespecified interface

Contain prespecified annotations

POJO stands for Plain Old Java Object, and would be used to describe the same things as

a "Normal Class"

POJO is usually used to describe a class that doesn't need to be a subclass of anything, or implement specific interfaces, or follow a specific pattern.It may also override Object.toString() and Object.equals().

a JavaBean follows a set of rules. Most commonly Beans use getters and setters to protect their member variables, which are typically set to private and have a no-argument public constructor.

Collections Sorting

http://javainsimpleway.com/collections-sorting-in-java/

**For arrays** sorting we use the method **Arrays.sort();**

**int**[] arrayOfIntegers = {3,1,0,7};

    Arrays.sort(arrayOfIntegers);

    System.out.println(Arrays.toString(arrayOfIntegers))

  String[] arrayOfStrings = {"hai","how","are","you"};

Arrays.sort(arrayOfStrings); //same way

**For collection sorting**

we use the method **Collections.sort()** for arraylist containing same type of data**;**

List stringList = **new** ArrayList();

stringList.add("java");

Collections.sort (stringList);

**for** (Object element : stringList) {

        System.out.println(element);

    }

**Collection Custom type Sort:**

* **all elements of collection should be of same type otherwise**

**Exception in thread “main” java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String**

* **custom type cannot be sorted as it is e.g. class objects otherwise**

**java.lang.ClassCastException: com.kb.collections\_sorting.Person cannot be cast to java.lang.Comparable**

**comparable mechanism**

* + We need to tell the mechanism how it should be sorted. And that mechanism is **comparable** sort based on one attribute
    - **class** Person **implements** **Comparable<Person>  {**

  @Override

**public** **int** **compareTo**(Person p)

{

**return** **this**.name.compareTo(p.name);

    } )

Client function:

Collections.sort(people);

**for** (Object person : people) {

    System.out.println(person);

}

**comparator mechanism**

* + We should use **comparator mechanism** whenever we need to sort custom objects based on different atttibutes.
  + Collections.sort(collection , comparator);

Case 1:

In Person Class:

**public** **static** Comparator<Person> nameComparator = **new** Comparator<Person>() {

        @Override

**public** **int** compare(Person p1, Person p2) {

**return** p1.name.compareTo(p2.name);

        }

    };

**public** **static** Comparator<Person> ageComparator = **new** Comparator<Person>() {

        @Override

**public** **int** compare(Person p1, Person p2) {

**return** p1.age - p2.age;

        }

Client func:

List people = **new** ArrayList();

Person p1 = **new** Person();

p1.setName("kb");

p1.setAge(26);

people.add(p1);

Collections.sort(people, Person.nameComparator);

**for** (Object person : people) {

            System.out.println(person);

        }

//same way for age based sorting

 Case 2: we want to sort based on age and name both in a single comparator

**public** **static** Comparator<Person> ageVoterIdComparator = **new** Comparator<Person>() {

        @Override

**public** **int** compare(Person p1, Person p2) {

**int** retVal =  p1.age - p2.age;;

**if**(retVal == 0){

**return** p1.voterId.compareTo(p2.voterId);

            }

**return** retVal;

        }

# **Serialization and externalization**

# Serialization **is the process of writing an object state (**atrributes and its values )**into the file.**

So to achieve serialization in java, any class must implement one marker interface. ( “serializable”.)

**public** **class** Employee **implements** Serializable{

**public** **class** Serialize {

Employee emp = **new** Employee();

  ObjectOutputStream oout = **null**;

**try** {

   oout = **new** ObjectOutputStream( **new** FileOutputStream (**new** File("employee.ser") ) );

   oout.writeObject(emp);

  }

**catch** (IOException e) { }

**catch** (Exception e) { }

**finally** {

**if** (oout != **null**) {

**try** {

     oout.flush();

     oout.close();

     oout = **null**;

    } **catch** (IOException e) {

     e.printStackTrace();

    }

**Deserialization** is if we need that object In future we can rebuild(read) it from the file

ObjectInputStream oin = ObjectInputStream(fileInputStream(**new** File("employee.ser"));

Employee emp = (Employee) oin.readObject();

System.out.println(emp.getAge());

**Remember in serialization**

1)If we dont want to save any of the attribute from the serializable class then we can make such attributes as “transient”.

2)all static fields can not be serializable sine its a class attribute not a object attribute.

transient String designation;

# **Externalization**

Externalization is nothing but serialization but an alternative for it and Externalizable interface extends Serializable interface.

public class ExternalizationBean implements Externalizable{

@Override

public void readExternal(ObjectInput out) throws IOException,

ClassNotFoundException {

name = (String) out.readObject();

id=(int)out.readInt();

temp=(String) out.readObject();

}

@Override

public void writeExternal(ObjectOutput in) throws IOException {

in.writeObject(name);

in.writeInt(id);

in.writeObject(temp);

**externalize the object into file:**

ExternalizationBean test=new ExternalizationBean();

FileInputStream fis = new FileInputStream("external.txt");

ObjectInputStream ois = new ObjectInputStream(fis);

test.writeExternal(ois);

**De-externalize the object:**

ExternalizationBean test=new ExternalizationBean();

FileInputStream fis = new FileInputStream("external.txt");

ObjectInputStream ois = new ObjectInputStream(fis);

test.readExternal(ois);

System.out.println("\*\*\*\*\*name\*\*\*\*\*\*\*\*\*\*\*"+test.getName());

## . Difference between Serialization and Externalization

* In case of Serializable, default serialization process is used. while in case of Externalizable custom Serialization process is used which is implemented by application.Externalizing gives control over the read/write process by implementing the readExternal and writeExternal methods of Externalizable.
* Serialization needs serialVersionUID, But Exteralization no need of serialVersionUID
* Serializable object will have object with data, but Externalizable object will have only data. That means in serialization will store directly object. But in terms of externalization we have to store the Data which have an object.
* Dserializable subclass can be serializable but externalizable object subclass can't be externalizable.

# [**Can an abstract class have a final method?**](https://stackoverflow.com/questions/1299398/can-an-abstract-class-have-a-final-method)

Classes that extend Abstract would still need to implement all abstract methods, but they'd be unable to extend final because it is declared final.

An abstract class can also have methods that are neither abstract nor final, just regular methods. These methods must be implemented in the abstract class, but it's up to the implementer to decide whether extending classes need to override them or not.

**Can we Override static methods in java?**  
We can declare static methods with same signature in subclass, but it is not considered overriding as there won’t be any run-time polymorphism. Hence the answer is ‘No’.  
If a derived class defines a static method with same signature as a static method in base class, the method in the derived class hides the method in the base class.

**Can Main method be overridden any number of times?**

No, it can't be **overridden** even once due to the plain fact that you can't override static methods.

Following are some important points for method overriding and static methods in Java.  
**1)** For class (or static) methods, the method according to the type of reference is called, not according to the abject being referred, which means method call is decided at compile time.

**2)** For instance (or non-static) methods, the method is called according to the type of object being referred, not according to the type of reference, which means method calls is decided at run time.

**3)** An instance method cannot override a static method, and a static method cannot hide an instance method. For example, the following program has two compiler errors.

# [**Can we overload the main method in Java?**](https://stackoverflow.com/questions/3759315/can-we-overload-the-main-method-in-java)

You can overload the main() method, but only public static void main(String[] args) will be used when your class is launched by the JV.  you have to call the overloaded main method from the actual main method.

that means main method acts as an entry point for the java interpreter to start the execute of the application. whereas a overloaded main need to be called from main.

public class Test {

public static void main(String[] args) {

System.out.println("main(String[] args)");

}

public static void main(String arg1) {

System.out.println("main(String arg1)");

}

public static void main(String arg1, String arg2) {

System.out.println("main(String arg1, String arg2)");

}

}

That will *always* print main(String[] args) when you run java Test ... from the command line, even if you specify one or two command-line arguments.

You can call the main() method yourself from code, of course - at which point the normal overloading rules will be applied.

EDIT: Note that you can use a varargs signature, as that's equivalent from a JVM standpoint:

public static void main(String... args)

Differences Between Static Binding And Dynamic Binding In Java :

http://javaconceptoftheday.com/static-binding-and-dynamic-binding-in-java/

|  |  |
| --- | --- |
| **Static Binding** | **Dynamic Binding** |
| It is a binding that happens at compile time. | It is a binding that happens at run time. |
| Actual object is not used for binding. | Actual object is used for binding. |
| It is also called early binding because binding happens during compilation. | It is also called late binding because binding happens at run time. |
| Method overloading is the best example of static binding. | Method overriding is the best example of dynamic binding. |
| Private, static and final methods show static binding. Because, they can not be overridden. | Other than private, static and final methods show dynamic binding. Because, they can be overridden. |

# Which Java packages are imported by default?

 java.lang package.

Note : one can import the same package or same class multiple times.  but JVM internally loads the required class only once, no matter how many times you import the same class.

**java.lang -**  
It Provides classes that are fundamental to the design of the Java programming language such as **String**, Math, and basic runtime support for threads and processes.  
e.g. System,Exception,Runtime,Thread…etc

**java.util** -  
It Provides the **collections framework**, formatted printing and scanning, array manipulation utilities, event model, date and time facilities, internationalization, and miscellaneous utility classes.

**Strings compare**

== tests for reference equality (whether they are the same object).

.equals() tests for value equality (whether they are logically "equal").

[Objects.equals()](http://docs.oracle.com/javase/7/docs/api/java/util/Objects.html#equals(java.lang.Object,%20java.lang.Object)) checks for nulls before calling .equals() so you don't have to (available as of JDK7, also available in [Guava](https://github.com/google/guava/wiki/CommonObjectUtilitiesExplained#equals)).

Consequently, if you want to test whether two strings have the same value you will probably want to use Objects.equals().

// These two have the same value

new String("test").equals("test") // --> true

// ... but they are not the same object

new String("test") == "test" // --> false

// ... neither are these

new String("test") == new String("test") // --> false

// ... but these are because literals are interned by

// the compiler and thus refer to the same object

"test" == "test" // --> true

// ... but you should really just call Objects.equals()

Objects.equals("test", new String("test")) // --> true

Objects.equals(null, "test") // --> false

You almost **always** want to useObjects.equals(). In the **rare** situation where you **know** you're dealing with [interned](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#intern--) strings, you can use ==