

# Java Bsics

JDK 1.0

* This is the first version of JDK introduced in way back in year 1996
* In the initial years, Java was actually referred as OAK, after an OAK tree that stood outside the office of developer/inventor James Gosling
* Through minor release, a stable version of JDK 1.0.2 is introduced by a new name called Java, which is now very popular computer masses even after 20 years
* **Release date:** 26th January, 1996

JDK 1.1

* This release is referred as Java 1
* **Release date:** 19th February, 1997

New features added in JDK 1.1:

* AWT event model
* Inner classes
* JavaBeans
* JDBC (**J**ava **D**ata**B**ase **C**onnectivity)
* RMI (**R**emote **M**ethod **I**nvocation)
* Reflection
* JIT (**J**ust-**I**n-**T**ime compiler)
* Internalization (I18N)
* Unicode

J2SE 1.2

* This release is commonly referred as Java 2 and alternatively as **Playground** after its codename
* Due to the 2nd edition of JDK release it is re-branded as J2SE, to distinguish it from J2ME
* This is very significant release, as size got tripled with 1520 classes and 59 packages
* **Release date:** 8th December, 1998

New features added in J2SE 1.2:

* Collection framework
* strictfp keyword
* Swing graphical API
* Sun’s JVM was equipped with a JIT compiler for the first time
* Java plug-in

**J2SE 1.3**

* This release is commonly referred as Java 3 and alternatively as **Kestrel** after its codename
* **Release date:** 8th May, 2000

New features added in J2SE 1.3:

* HotSpot JVm
* RMI (actually this is modified to support CORBA)
* JNDI (**J**ava **N**aming and **D**irectory **I**nterface)
* JPDA (**J**ava **P**latform **D**ebugger **A**rchitecture)
* JavaSound
* Synthetic Proxy classes

J2SE 1.4

* This release is commonly referred as Java 4 and alternatively as **Merlin** after its codename
* This release was first time that, it is developed under Java Community Process as JSR 59
* **Release date:** 6th February, 2002

New features added in J2SE 1.4:

* assert keyword
* Regex (Regular Expression)
* Exception chaining process
* Support for IPV6 (Internet Protocol Version 6)
* NIO (Non-blocking I/O)
* Logging API
* Image I/O API
* JAXP (Integrated XML parser and XSLT processor)
* Integrated security and cryptography extensions (JCE, JSSE, JAAS)
* Java Web Start
* Preferences API (*java.util.prefs*)

J2SE 5.0

* This release is commonly referred as Java 5 and alternatively as **Tiger** after its codename
* After this release, numbering concept totally changed and it is referred as J2SE 5.0
* It is one of the popular version among its release for major overhaul in its features
* This is release ensemble with major Java language features. We will list them below
* **Release date:** 30th September, 2004

New features added in J2SE 5.0:

* Auto-boxing and Un-boxing (automatic type-conversion)
* Generics
* Annotations (Metadata)
* forEach loop
* varargs
* static imports
* Enumerations
* Concurrent classes (*java.util.concurrent*)
* Co-variant return-type
* Scanner classes

Java SE 6

* This release is commonly referred as Java 6 and alternatively as **Mustang** after its codename
* From here, Sun replaced J2SE with just Java SE and point notation like .0 from its actual version (i.e.; 2 is removed, actually it was continued from successful 2nd edition and also 6.0 replaced with just 6)
* **Release date:** 11th December, 2006

New features added in Java SE 6:

* Support for Scripting Language (JSR-223)
* JAXB 2.0
* JAX-WS (JSR-224)
* JDBC 4.0 API
* Java Compiler API
* Pluggable annotation (JSR-269)
* GC algorithm
* Performance improvement
* NavigableSet and NavigableMap interface

Java SE 7

* This release is commonly referred as Java 7 and alternatively as **Dolphin** after its codename
* **Release date:** 28th July, 2011

New features added in Java SE 7:

* switch statement with String
* ARM (Automatic Resource Management)
* try-with-resources for ARM
* Multi catch block (with pipe separator)
* JVM support for dynamic languages
* Binary integer literals
* Numeric literals with underscore support
* NIO 2.0
* Simplified varargs method declaration
* Timsort is used to sort collections and arrays of objects instead of merge sort
* Compressed 64-bit pointers
* Diamond operator (Improved type inference for generic instance creation)

Java SE 8

* This release is commonly referred as Java 8 and alternatively as **Spider** after its codename
* **Release date:** 18th March, 2014

New features added in Java SE 8:

* Lambda expression
* Method references
* Functional interface
* Static and default methods in interface
* New date and time API
* Streams
* forEach method
* Collectors
* Project Nashorn, a JavaScript runtime which allows developers to embed JavaScript code within applications
* Annotation on Java Types
* Unsigned Integer Arithmetic
* Repeating annotations
* Statically-linked JNI libraries
* Launch JavaFX applications
* Remove the permanent generation

What is Class Loader in java ?

Java Runtime Environment that dynamically loads Java classes into the Java Virtual Machine on demand

There are three types of class loaders. There are

 Bootstrap class loader

         The bootstrap class loader loads the core Java libraries[5] located in the {JAVA\_HOME}/lib directory  
  Extensions class loader

          extensions class loader loads the code in the extensions directories {JAVA\_HOME}/lib/ext directory  
   System Class  Path Loader

              The system class loader loads code found on java.class.path, which maps to the system CLASSPATH variable

# What are the SOLID principles in Java?

| Principle | Description |
| --- | --- |
| **Single Responsibility Principle** | Each class should be responsible for a single part or functionality of the system. |
| **Open-Closed Principle** | Software components should be open for extension, but not for modification. |
| **Liskov Substitution Principle** | Objects of a superclass should be replaceable with objects of its subclasses without breaking the system. |
| **Interface Segregation Principle** | No client should be forced to depend on methods that it does not use. |
| **Dependency Inversion Principle** | High-level modules should not depend on low-level modules, both should depend on abstractions. |

# **Why string is immutable in java?**

[String in java](https://javagoal.com/string-class-in-java/) is a [final class](https://javagoal.com/final-class-in-java/) and **immutable class**, so it can’t be inherited, and we can’t alter the object. There are a number of factors that considered when the string was introduced in java. We will discuss them one by one and need the concept of memory, synchronization, data structures, etc.

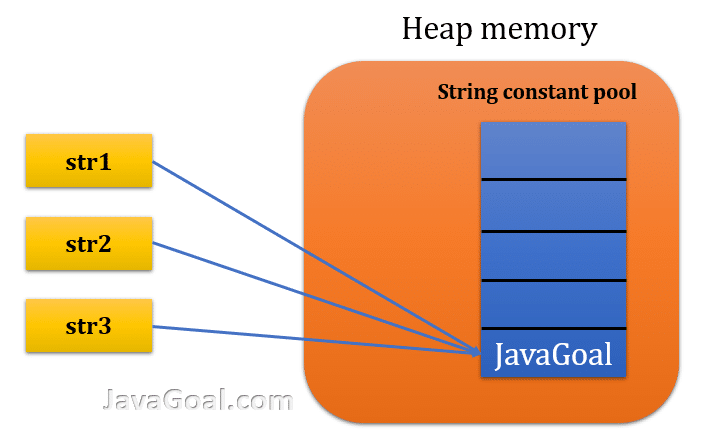
**1. Requirement of String Pool**

In java, the String is the most widely used data structure. The **string constant pool** is special storage in heap memory that Caching the **String literals** and reusing them. Whenever we create a string, the **JVM**checks whether the string already exists in the**string constant pool** or not. If string already presented the reference of the existing string will be returned, instead of creating a new object. Let’s see how to **string constant pool** working in memory. Here we will create three strings with same and different reference variables.

String str1 = "JavaGoal";

String str2 = "JavaGoal";

String str3 = "JavaGoal";



Suppose if the string is not immutable and here three reference variables pointing to the same object. If we change the string with one reference will lead to the wrong value for the other references. It creates problems and affects all other references.

**2. Cache of Hashcode**

As we know string objects have a large use in java. The **hashcode()** implementation of the string is widely used in java like **HashMap, HashTable, HashSet,** etc. The [hashcode() method](https://javagoal.com/hashcode-method-in-java/" \t "_blank) returns the value of a particular string object. As we know strings are very popular as HashMap key, The immutable string gives guarantees that their value will not be changed.It means, there is no need to calculate hashcode every time it is used.

**3. Security**

In java, security parameters are also represented as String like in network connections, opening files, database connection URLs, usernames, and passwords, etc. Suppose if the string is mutable then any hacker could change the referenced value break the security.

**4. Synchronized**

Due to the immutable nature of string, it is thread-safe. Suppose multiple threads are running but they can’t be changed because if a thread changes the value, then instead of modifying the same, a new String would be created in the String pool. This eliminates the requirements of doing synchronization.

**5. Performance**

As we have seen the string constant pool enhances memory storage and performance. Since String is the most widely used data structure, improving the performance

**6. Classloader**

The String object is also used as arguments for class loading. Due to the immutability of string the correct class is getting loaded by Classloader.

**What is the order of execution of Class Loader in java?**  
Java applications  involve three classloaders – Bootstrap, Extensions and System-Classpath classloaders  
Bootstrap classloader is the parent of all classloaders ,extensions Classloader is the immediate child of Bootstrap classloader and System-Classpath classloader is the immediate child of Extensions classloader. Generally  Order of execution happens from sub-sequent children to parent.   
So order of execution is starts from System-classpath-loader then extension then Bootstrap Loader.

**If we are using importing a class e.g java.util.ArrayList , and we are not using that class in side our program, does class loader will load the class?**  
No, it won't

**Local variable, instance variable, method and object, amonng of these which one will store in heap and which one will store in stack?**  
local variable and method will store in stack  
instance variable and object will store in heap

**Can I write like this?**  
void methodA()  
{  
int a;  
System.out.println("I am in methodA " + (a+5));  
}  
no, because as long as we are not using local variable it won't give any error, but we should initialise it before using it(using of a).

**Can we declare a local variable with in static block, like below?**  
public class TestVariable {  
        static{  
         int i=9;  
         System.out.println("print  "+i);  
    }  
yes

**Backpropagation** is a common method for training a neural network.Backpropagation is a method to calculate the gradient of the loss function with respect to the weights in an artificial neural network. It is commonly used as a part of algorithms that optimize the performance of the network by adjusting the weights, for example in the gradient descent algorithm.

**Can I use instance variable in static variable like below?**  
public class TestBlock {  
    int z=98;  
    static{  
        System.out.println("I am in static block"+new TestBlock().z);  
    }  
    public static void main(String a[]){  
        TestBlock block=new TestBlock();    }}  
yes, you can execute this program, out put will  
I am in static block98

**What will be the output for following program?**  
public class TypeCast {  
    public static void main(String [] args) {  
        long l = 129L;  
        byte b = (byte)l;  
        System.out.println("The byte is " + b);  
        }  
}  
Ans The byte is -127  
byte range is -128 to 127. After 127 it'll calculate  form -128

Super class object can not cast to sub class.

**What is the difference between stackoverflowerror vs outofmemoryerror ?**

 stackoverflow happens in overflow of stack ,and outofmemory ocuures if you dreained your heap.

**How can you make your custom  class iterate through advanced for loop ?**  
You need to make your class implement the Iterable interface, override the iterator() method , and create Iterator class , which can iterate your items in the class.

class A{}

class B extends A{}

class C extends B{}

public class MainClass {

static void overloadedMethod(A a) {

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

static void overloadedMethod(B b) {

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

static void overloadedMethod(Object obj) {

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

public static void main(String[] args) {

C c = new C();

overloadedMethod(c);

}}

**Answer : TWO**

**2) In a class, one method has two overloaded forms. One form is defined as static and another form is defined as non-static. Is that method properly overloaded?**

**Answer :**  
Yes. Compiler checks only method signature to verify whether a particular method is properly overloaded or not. It doesn’t check static or non-static feature of the method.

class A{

    static void staticMethod()    {

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

    }}

public class MainClass{

    public static void main(String[] args)    {

        A a = null;

        a.staticMethod();

    }}

Yes, the code is correct. You can call static methods through reference variable which is pointing to null. Output will be**, Static Method**

**In the below class, is ‘method’ overloaded or duplicated?**

public class MainClass {

    void method(int ... a)    {

        System.out.println(1);    }

    void method(int[] a)    {

        System.out.println(2);

    }

}

Duplicated. Because, var args (int … a) are nothing but the arrays. So here, (int … a) and (int[] a) are the same.

**Can you instantiate this class?**

|  |  |
| --- | --- |
| 1  2  3  4 | public class A  {      A a = new A();  } |

 Not possible. Because while instantiating, constructor will be called recursively.

**What is cloning in java?**

Cloning is done for copying the object, cloning can be done using shallow or deep copy, Few key points about clone method>

1) Definition of clone method -

|  |
| --- |
| protected native Object clone() throws CloneNotSupportedException; |

Clone is a protected method - clone method can’t be called outside class without inheritance.

Clone is native method, if not overridden its implementation is provided by JVM.

It returns Object - Means explicitly cast is needed to convert it to original object.

2) By default clone method do shallow copy.

3) Class must implement marker interface java.lang.Cloneable. If class doesn’t implement Cloneable than calling clone method on its object will throw CloneNotSupportedException.

4) shallow copy- If we implement Cloneable interface, we must override clone method and call super.clone() from it, invoking super.clone() will do shallow copy.

5) Deep copy - We need to provide custom implementation of clone method for deep copying.  When the copied object contains some other object its references are copied recursively in deep copy.

|  |
| --- |
| /\*\* Copyright (c), AnkitMittal [www.JavaMadeSoEasy.com](http://www.javamadesoeasy.com/) \*/  **public** **class** ReverseStringRecursionExample {  **public** **static** **void** main(String...args){            String originalString="abcde"; //String to be reversed              System.*out*.println("Original String: "+originalString);            System.*out*.print("Reversed String: ");  ***reverseRecursively***(originalString);   }  **public** **static** **void** **reverseRecursively**(String str) {  **if** (str.length() == 1){                   System.*out*.print(str);          }  **else** {  ***reverseRecursively***(str.substring(1, str.length()));                   System.*out*.print(str.substring(0, 1));            }  }} |

public class reverseString {

public static void main(String[] args) {

String input = "Be in present";

char[] temparray= input.toCharArray();

int left,right=0;

right=temparray.length-1;

for (left=0; left < right ; left++ ,right--)

{

// Swap values of left and right

char temp = temparray[left];

temparray[left] = temparray[right];

temparray[right]=temp;

}

for (char c : temparray) System.out.print(c); System.out.println(); }}

|  |  |  |
| --- | --- | --- |
|  | Method overloading | Method overriding |
| 1 | When a class have same method name with different argument, than it is called method overloading. | Method overriding - Method of superclass is overridden in subclass to provide more specific implementation. |
| 2 | Method is overloaded by - keeping same name of method and only changing number of arguments  Let’s compare with method overriding in java.  Method name - same method name.  Access modifier - Does not matter.  return type - Does not matter.    Number of parameters in java - Have different number of [parameters](http://www.javamadesoeasy.com/2015/06/difference-between-arguments-and.html)  Exception thrown - Does not matter. | In Method overriding - Method of superclass is overridden in subclass when overriding method of subclass in java -  Method name - Have same name as of superclass method,  [Access modifier](http://www.javamadesoeasy.com/2015/06/access-modifier-access-specifier-in.html) - Must not have more restrictive modifier. Example - public method cannot be overridden by private method.  https://lh4.googleusercontent.com/57jtmPxTx9UfXgbCbDGbAUgpEG4xsBuQIBfO790kOsGKYeFsmqlp7SsIemRoefbLVzcBrZqEDLk3JnAEKr7q2Rd3EOGV_3BOGPsNMH7ZSvw5W9AAsn12WJy0rbWL2SnWo9fBz-A  Return type - Java allow overriding by changing the return type, but only Covariant return type are allowed in java.  Number of parameters in java - Have same number of parameters in java.  [Exception thrown](http://www.javamadesoeasy.com/2015/05/throwdeclare-checked-and-unchecked.html) -  Overriding method must not throw new or broader [checked exception](http://www.javamadesoeasy.com/2015/05/checked-compile-time-exceptions-and.html),  though Overriding method may throw new narrower(subclass) of checked exception or  Overriding method can throw any runtime exception in java.  For more detail on this point please Read : [Throw/declare checked and unchecked exception while overriding superclass method in java](http://www.javamadesoeasy.com/2015/05/throwdeclare-checked-and-unchecked.html) |
| 3 | Method overloading is generally done in same class but can also be done in SubClass (See [Program 3](http://www.javamadesoeasy.com/2015/06/method-overloading-in-java-in-detail.html)) | Method overriding is always done in subClass in java. |
| 4 | Both [Static](http://www.javamadesoeasy.com/2015/05/static-keyword-in-java-variable-method.html) and instance method can be overloaded in java. | Only instance methods can be overridden in java.  Static methods can’t be overridden in java. ([Please refer this article for detailed analysis and explanation with program](http://www.javamadesoeasy.com/2015/05/why-static-method-cannot-be-overridden.html)) |
| 5 | Main method can also be overloaded in java (In [Program 4](http://www.javamadesoeasy.com/2015/06/method-overloading-in-java-in-detail.html)) | Main method can’t be overridden in java, because main is static method and static methods can’t be overridden in java (as mentioned in above point) |
| 6 | private methods can be overloaded in java. | private methods can’t be overridden in java, because private methods are not inherited in subClass in java. |
| 7 | [final](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) methods can be overloaded in java. | final methods can’t be overridden in java, because final methods are not inherited in subClass in java. |
| 8 | Call to overloaded method is bonded at compile time in java. | Call to overridden method is bonded at runtime in java. |

The purpose of a nested class is to clearly group the nested class with its surrounding class, signaling that these two classes are to be used together. Or perhaps that the nested class is only to be used from inside its enclosing (owning) class.

Java developers often refer to *nested classes* as *inner classes*, but inner classes (non-static nested classes) are only one out of several different types of nested classes in Java.

In Java nested classes are considered members of their enclosing class. Thus, a nested class can be declared public, package (no access modifier), protected and private (see [access modifiers](http://tutorials.jenkov.com/java/access-modifiers.html) for more info). Therefore nested classes in Java can also be inherited by subclasses as explained in my tutorial about [Java inheritance](http://tutorials.jenkov.com/java/inheritance.html).You can create several different types of nested classes in Java. The different Java nested class types are:

Static nested classes

Non-static nested classes

Local classes

Anonymous classes

All these types of nested classes will be covered in the following sections.

Static Nested Classes

Static nested classes are declared in Java like this:

public class Outer {

public static class Nested { }}

In order to create an instance of the Nested class you must reference it by prefixing it with the Outer class name, like this:

Outer.Nested instance = new Outer.Nested();

In Java a static nested class is essentially a normal class that has just been nested inside another class. Being static, a static nested class can only access instance variables of the enclosing class via a reference to an instance of the enclosing class.

Non-static Nested Classes (Inner Classes)

Non-static nested classes in Java are also called *inner classes*. Inner classes are associated with an instance of the enclosing class. Thus, you must first create an instance of the enclosing class to create an instance of an inner class. Here is an example inner class definition:

public class Outer {

public class Inner { }}

Here is how you create an instance of the Inner class:

Outer outer = new Outer();

Outer.Inner inner = outer.new Inner();

Notice how you put new after the reference to the outer class in order to create an instance of the inner class.

Non-static nested classes (inner classes) have access to the fields of the enclosing class, even if they are declared private. Here is an example of that:

public class Outer {

private String text = "I am private!";

public class Inner {

public void printText() { System.out.println(text); } }}

Notice how the printText() method of the Inner class references the private text field of the Outer class. This is perfectly possible. Here is how you would call the printText() method:

Outer outer = new Outer();

Outer.Inner inner = outer.new Inner();

**inner.printText();**

Inner Class Shadowing

If a Java inner class declares fields or methods with the same names as field or methods in its enclosing class, the inner fields or methods are said to *shadow* over the outer fields or methods. Here is an example:

public class Outer {

private String text = "I am Outer private!";

public class Inner {

private String text = "I am Inner private";

public void printText() {

System.out.println(text); } }}

In the above example both the Outer and Inner class contains a field named text. When the Inner class refers to text it refers to its own field. When Outer refers to text it also refers to its own field.

Java makes it possible though, for the Inner class to refer to the text field of the Outer class. To do so it has to prefix the text field reference with Outer.this. (the outer class name + .this. + field name) like this:

public class Outer {

private String text = "I am Outer private!";

public class Inner {

private String text = "I am Inner private";

public void printText() {

System.out.println(text);

System.out.println(**Outer.this.text**);

} }}

Now the Inner.printText() method will print both the Inner.text and Outer.text fields.Local Classes

Local classes in Java are like inner classes (non-static nested classes) that are defined inside a method or scope block ({ ... }) inside a method. Here is an example:

class Outer {

public void printText() {

class Local { }

Local local = new Local();

}}

Local classes can only be accessed from inside the method or scope block in which they are defined.

Local classes can access members (fields and methods) of its enclosing class just like regular inner classes.

Local classes can also access local variables inside the same method or scope block, provided these variables are declared final.

From Java 8 local classes can also access local variables and parameters of the method the local class is declared in. The parameter will have to be declared final or be *effectually final*. Effectually final means that the variable is never changed after it is initialized. Method parameters are often effectually final.

Local classes can also be declared inside static methods. In that case the local class only has access to the static parts of the enclosing class. Local classes cannot contain all kinds of static declarations (constants are allowed - variables declared static final), because local classes are non-static in nature - even if declared inside a static method.

The same shadowing rules apply for local classes as for inner classes.

Anonymous Classes

Anonymous classes in Java are nested classes without a class name. They are typically declared as either subclasses of an existing class, or as implementations of some [**interface**](http://tutorials.jenkov.com/java/interfaces.html).

Anonymous classes are defined when they are instantiated. Here is an example that declares an anonymous subclass of a superclass called SuperClass:

public class SuperClass {

public void doIt() {

System.out.println("SuperClass doIt()");

}}

SuperClass instance = new SuperClass() {

public void doIt() {

System.out.println("Anonymous class doIt()"); }};

instance.doIt();

Running this Java code would result in Anonymous class doIt() being printed to System.out. The anonymous class subclasses (extends) SuperClass and overrides the doIt() method.A Java anonymous class can also implement an interface instead of extending a class. Here an example:

public interface MyInterface {

public void doIt();}

MyInterface instance = new MyInterface() {

public void doIt() {

System.out.println("Anonymous class doIt()");

}};

instance.doIt();As you can see, an anonymous class implementing an interface is pretty similar to an anonymous class extending another class.

An anonymous class can access members of the enclosing class. It can also access local variables which are declared final or effectively final (since Java 8).You can declare fields and methods inside an anonymous class, but you cannot declare a constructor. You can declare a static initializer for the anonymous class instead, though. Here is an example:

final Strint textToPrint = "Text...";

MyInterface instance = new MyInterface() {

private String text;

//static initializer

{ this.text = textToPrint; }

public void doIt() { System.out.println(this.text); }};

instance.doIt();

The same shadowing rules apply to anonymous classes as to inner classes.

Nested Class Benefits

The benefits of Java nested classes are that you can group classes together that belong together. You could do so already by putting them in the same package, but putting one class inside another makes an even stronger grouping.

A nested class is typically only used *by* or *with* its enclosing class. Sometimes a nested class is only visible to the enclosing class, is only used internally, and is thus never visible outside the enclosing class. Other times the nested class is visible outside its enclosing class, but can only be used in conjunction with the enclosing class.

An example would be a Cache class. Inside the Cache class you might declare a CacheEntry class which can contain information about a specific cache entry (cached value, time inserted, number of times accessed etc.). Users of the Cache class may never see the CacheEntry class, if they have no need to obtain information about the CacheEntry itself, but only the cached value. However, the Cache class may choose to make the CacheEntry class visible to the outside world, so they can access more than just the cached value (for instance information about when the value was last refreshed etc.).

public boolean equals(Object obj){

          if(obj==null)

                 return false;

          if(this.getClass()!=obj.getClass())

                 return false;

             Employee emp=(Employee)obj;

          return (emp.id==this.id || emp.id.equals(this.id))

                              && (emp.name==this.name || emp.name.equals(this.name));

   }             @Override

   public int hashCode(){

          int hash=(this.id==null ? 0: this.id.hashCode() ) +

                       (this.name==null ? 0: this.name.hashCode() );

          return hash;         }

Association

Association is a relationship between two objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. Aggregation is a special form of association. Composition is a special form of aggregation.

http://javapapers.com/wp-content/uploads/2010/06/association.jpg

Example: A Student and a Faculty are having an association.

Aggregation

Aggregation is a special case of association. A directional association between objects. When an object ‘has-a’ another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a “Has-a” relationship.

http://javapapers.com/wp-content/uploads/2010/06/aggregation.jpg

Composition

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

http://javapapers.com/wp-content/uploads/2010/06/composition.jpg

What are immutable classes in java? How we can create immutable classes in java? And what are advantages of using immutable classes?

Any change made to object of immutable class produces new object.

Example- [String is Immutable class in java](http://www.javamadesoeasy.com/2015/05/string-is-immutable-in-java.html), any changes made to Sting class.

We must follow following steps for creating immutable classes -

    1) [Final](http://www.javamadesoeasy.com/2015/05/final-keyword-in-java-20-salient.html) class - Make class final so that it cannot be inherited

2) private member variable -> Making member variables private ensures that fields cannot be accessed outside class.

3) final member variable -> Make member variables final so that once assigned their values cannot be changed

4) Constructor -> Initialize all fields in constructor.

     assign all mutable member variable using new keyword.

5) Don't provide setter methods in class/ provide only getter methods.

Iterator returned by few Collection framework Classesare **fail-fast,** means any structural modification made to these classes during iteration will throw ConcurrentModificationException.

Some important classes whose returned iterator is **fail-fast >**

[ArrayList](http://www.javamadesoeasy.com/2015/04/arraylist-in-java.html) [LinkedList](http://www.javamadesoeasy.com/2015/04/linkedlist-in-java.html) [vector](http://www.javamadesoeasy.com/2015/04/arraylist-vs-vector-similarity-and.html) [HashSet](http://www.javamadesoeasy.com/2015/04/hashset-in-java.html)

Welcome to my introduction to Java 8. This tutorial guides you step by step through all new language features. Backed by short and simple code samples you'll learn how to use default interface methods, lambda expressions, method references and repeatable annotations. At the end of the article you'll be familiar with the most recent API changes like streams, functional interfaces, map extensions and the new Date API.

No walls of text - just a bunch of commented code snippets. Enjoy!

Default Methods for Interfaces

Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the default keyword. This feature is also known as Extension Methods. Here is our first example:

interface Formula {

double calculate(int a);

default double sqrt(int a) {

return Math.sqrt(a);

}}

Besides the abstract method calculate the interface Formula also defines the default method sqrt. Concrete classes only have to implement the abstract method calculate. The default method sqrt can be used out of the box.

Formula formula = new Formula() {

@Override

public double calculate(int a) {

return sqrt(a \* 100);

}};

formula.calculate(100); // 100.0

formula.sqrt(16); // 4.0

The formula is implemented as an anonymous object. The code is quite verbose: 6 lines of code for such a simple calucation of sqrt(a \* 100). As we'll see in the next section, there's a much nicer way of implementing single method objects in Java 8.

Lambda expressions

Let's start with a simple example of how to sort a list of strings in prior versions of Java:

List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {

@Override

public int compare(String a, String b) {

return b.compareTo(a); }});

The static utility method Collections.sort accepts a list and a comparator in order to sort the elements of the given list. You often find yourself creating anonymous comparators and pass them to the sort method.

Instead of creating anonymous objects all day long, Java 8 comes with a much shorter syntax, lambda expressions:

Collections.sort(names, (String a, String b) -> {

return b.compareTo(a);});

As you can see the code is much shorter and easier to read. But it gets even shorter:

Collections.sort(names, (String a, String b) -> b.compareTo(a));

For one line method bodies you can skip both the braces {} and the return keyword. But it gets even more shorter:

Collections.sort(names, (a, b) -> b.compareTo(a));

The java compiler is aware of the parameter types so you can skip them as well. Let's dive deeper into how lambda expressions can be used in the wild.

Functional Interfaces

How does lambda expressions fit into Javas type system? Each lambda corresponds to a given type, specified by an interface. A so called functional interface must contain exactly one abstract method declaration. Each lambda expression of that type will be matched to this abstract method. Since default methods are not abstract you're free to add default methods to your functional interface.

We can use arbitrary interfaces as lambda expressions as long as the interface only contains one abstract method. To ensure that your interface meet the requirements, you should add the @FunctionalInterface annotation. The compiler is aware of this annotation and throws a compiler error as soon as you try to add a second abstract method declaration to the interface.

Example:

@FunctionalInterface

interface Converter<F, T> {

T convert(F from);}

Converter<String, Integer> converter = (from) -> Integer.valueOf(from);

Integer converted = converter.convert("123");

System.out.println(converted); // 123

Keep in mind that the code is also valid if the @FunctionalInterface annotation would be ommited.

Method and Constructor References

The above example code can be further simplified by utilizing static method references:

Converter<String, Integer> converter = Integer::valueOf;

Integer converted = converter.convert("123");

System.out.println(converted); // 123

Java 8 enables you to pass references of methods or constructors via the :: keyword. The above example shows how to reference a static method. But we can also reference object methods:

class Something {

String startsWith(String s) {

return String.valueOf(s.charAt(0)); }}

Something something = new Something();

Converter<String, String> converter = something::startsWith;

String converted = converter.convert("Java");

System.out.println(converted); // "J"

Let's see how the :: keyword works for constructors. First we define an example bean with different constructors:

class Person {

String firstName;

String lastName;

Person() {}

Person(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName; }}

Next we specify a person factory interface to be used for creating new persons:

interface PersonFactory<P extends Person> {

P create(String firstName, String lastName);}

Instead of implementing the factory manually, we glue everything together via constructor references:

PersonFactory<Person> personFactory = Person::new;

Person person = personFactory.create("Peter", "Parker");

We create a reference to the Person constructor via Person::new. The Java compiler automatically chooses the right constructor by matching the signature of PersonFactory.create.

Lambda Scopes

Accessing outer scope variables from lambda expressions is very similar to anonymous objects. You can access final variables from the local outer scope as well as instance fields and static variables.

Accessing local variables

We can read final local variables from the outer scope of lambda expressions:

final int num = 1;

Converter<Integer, String> stringConverter =

(from) -> String.valueOf(from + num);

stringConverter.convert(2); // 3

But different to anonymous objects the variable num does not have to be declared final. This code is also valid:

int num = 1;

Converter<Integer, String> stringConverter =

(from) -> String.valueOf(from + num);

stringConverter.convert(2); // 3

However num must be implicitly final for the code to compile. The following code does not compile:

int num = 1;

Converter<Integer, String> stringConverter =

(from) -> String.valueOf(from + num);

num = 3;

Writing to num from within the lambda expression is also prohibited.

Accessing fields and static variables

In constrast to local variables we have both read and write access to instance fields and static variables from within lambda expressions. This behaviour is well known from anonymous objects.

class Lambda4 {

static int outerStaticNum;

int outerNum;

void testScopes() {

Converter<Integer, String> stringConverter1 = (from) -> {

outerNum = 23;

return String.valueOf(from); };

Converter<Integer, String> stringConverter2 = (from) -> {

outerStaticNum = 72;

return String.valueOf(from); }; }}

Accessing Default Interface Methods

Remember the formula example from the first section? Interface Formula defines a default method sqrt which can be accessed from each formula instance including anonymous objects. This does not work with lambda expressions.

Default methods cannot be accessed from within lambda expressions. The following code does not compile:

Formula formula = (a) -> sqrt( a \* 100);

Built-in Functional Interfaces

The JDK 1.8 API contains many built-in functional interfaces. Some of them are well known from older versions of Java like Comparator or Runnable. Those existing interfaces are extended to enable Lambda support via the @FunctionalInterface annotation.

But the Java 8 API is also full of new functional interfaces to make your life easier. Some of those new interfaces are well known from the Google Guava library. Even if you're familiar with this library you should keep a close eye on how those interfaces are extended by some useful method extensions.

Predicates

Predicates are boolean-valued functions of one argument. The interface contains various default methods for composing predicates to complex logical terms (and, or, negate)

Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo"); // true

predicate.negate().test("foo"); // false

Predicate<Boolean> nonNull = Objects::nonNull;

Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;

Predicate<String> isNotEmpty = isEmpty.negate();

Functions

Functions accept one argument and produce a result. Default methods can be used to chain multiple functions together (compose, andThen).

Function<String, Integer> toInteger = Integer::valueOf;

Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123"); // "123"

Suppliers

Suppliers produce a result of a given generic type. Unlike Functions, Suppliers don't accept arguments.

Supplier<Person> personSupplier = Person::new;

personSupplier.get(); // new Person

Consumers

Consumers represents operations to be performed on a single input argument.

Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);

greeter.accept(new Person("Luke", "Skywalker"));

Comparators

Comparators are well known from older versions of Java. Java 8 adds various default methods to the interface.

Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");

Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2); // > 0

comparator.reversed().compare(p1, p2); // < 0

Optionals

Optionals are not functional interfaces, instead it's a nifty utility to prevent NullPointerException. It's an important concept for the next section, so let's have a quick look at how Optionals work.

Optional is a simple container for a value which may be null or non-null. Think of a method which may return a non-null result but sometimes return nothing. Instead of returning null you return an Optional in Java 8.

Optional<String> optional = Optional.of("bam");

optional.isPresent(); // true

optional.get(); // "bam"

optional.orElse("fallback"); // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0))); // "b"

Streams

A java.util.Stream represents a sequence of elements on which one or more operations can be performed. Stream operations are either intermediate or terminal. While terminal operations return a result of a certain type, intermediate operations return the stream itself so you can chain multiple method calls in a row. Streams are created on a source, e.g. a java.util.Collection like lists or sets (maps are not supported). Stream operations can either be executed sequential or parallel.

Let's first look how sequential streams work. First we create a sample source in form of a list of strings:

List<String> stringCollection = new ArrayList<>();

stringCollection.add("ddd2");

stringCollection.add("aaa2");

stringCollection.add("bbb1");

stringCollection.add("aaa1");

stringCollection.add("bbb3");

stringCollection.add("ccc");

stringCollection.add("bbb2");

stringCollection.add("ddd1");

Collections in Java 8 are extended so you can simply create streams either by calling Collection.stream() or Collection.parallelStream(). The following sections explain the most common stream operations.

Filter

Filter accepts a predicate to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation (forEach) on the result. ForEach accepts a consumer to be executed for each element in the filtered stream. ForEach is a terminal operation. It's void, so we cannot call another stream operation.

stringCollection

.stream()

.filter((s) -> s.startsWith("a"))

.forEach(System.out::println);

// "aaa2", "aaa1"

Sorted

Sorted is an intermediate operation which returns a sorted view of the stream. The elements are sorted in natural order unless you pass a custom Comparator.

stringCollection

.stream().sorted().filter((s) -> s.startsWith("a")) .forEach(System.out::println);

// "aaa1", "aaa2"

Keep in mind that sorted does only create a sorted view of the stream without manipulating the ordering of the backed collection. The ordering of stringCollection is untouched:

System.out.println(stringCollection);

// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1

Map

The intermediate operation map converts each element into another object via the given function. The following example converts each string into an upper-cased string. But you can also use map to transform each object into another type. The generic type of the resulting stream depends on the generic type of the function you pass to map.

stringCollection

.stream().map(String::toUpperCase).sorted((a, b) -> b.compareTo(a)) .forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"

Match

Various matching operations can be used to check whether a certain predicate matches the stream. All of those operations are terminal and return a boolean result.

boolean anyStartsWithA =

stringCollection.stream() .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA); // true

boolean allStartsWithA =

stringCollection.stream() .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA); // false

boolean noneStartsWithZ =

stringCollection .stream().noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ); // true

Count

Count is a terminal operation returning the number of elements in the stream as a long.

long startsWithB =

stringCollection.stream().filter((s) -> s.startsWith("b")).count();

System.out.println(startsWithB); // 3

Reduce

This terminal operation performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value.

Optional<String> reduced =

stringCollection

.stream()

.sorted()

.reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);

// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"

Parallel Streams

As mentioned above streams can be either sequential or parallel. Operations on sequential streams are performed on a single thread while operations on parallel streams are performed concurrent on multiple threads.

The following example demonstrates how easy it is to increase the performance by using parallel streams.

First we create a large list of unique elements:

int max = 1000000;

List<String> values = new ArrayList<>(max);

for (int i = 0; i < max; i++) {

UUID uuid = UUID.randomUUID();

values.add(uuid.toString());}

Now we measure the time it takes to sort a stream of this collection.

Sequential Sort

long t0 = System.nanoTime();

long count = values.stream().sorted().count();

System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);

System.out.println(String.format("sequential sort took: %d ms", millis));

// sequential sort took: 899 ms

Parallel Sort

long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();

System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);

System.out.println(String.format("parallel sort took: %d ms", millis));

// parallel sort took: 472 ms

As you can see both code snippets are almost identical but the parallel sort is roughly 50% faster. All you have to do is change stream() to parallelStream().

Map

As already mentioned maps don't support streams. Instead maps now support various new and useful methods for doing common tasks.

Map<Integer, String> map = new HashMap<>();

for (int i = 0; i < 10; i++) {

map.putIfAbsent(i, "val" + i);}

map.forEach((id, val) -> System.out.println(val));

The above code should be self-explaining: putIfAbsent prevents us from writing additional if null checks; forEach accepts a consumer to perform operations for each value of the map.

This example shows how to compute code on the map by utilizing functions:

map.computeIfPresent(3, (num, val) -> val + num);

map.get(3); // val33

map.computeIfPresent(9, (num, val) -> null);

map.containsKey(9); // false

map.computeIfAbsent(23, num -> "val" + num);

map.containsKey(23); // true

map.computeIfAbsent(3, num -> "bam");

map.get(3); // val33

Next, we learn how to remove entries for a a given key, only if it's currently mapped to a given value:

map.remove(3, "val3");

map.get(3); // val33

map.remove(3, "val33");

map.get(3); // null

Another helpful method:

map.getOrDefault(42, "not found"); // not found

Merging entries of a map is quite easy:

map.merge(9, "val9", (value, newValue) -> value.concat(newValue));

map.get(9); // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValue));

map.get(9); // val9concat

Merge either put the key/value into the map if no entry for the key exists, or the merging function will be called to change the existing value.

UPDATE - I'm currently working on a JavaScript implementation of the Java 8 Streams API for the browser. If I've drawn your interest check out Stream.js on GitHub. Your Feedback is highly appreciated.

Date API

Java 8 contains a brand new date and time API under the package java.time. The new Date API is comparable with the Joda-Time library, however it's not the same. The following examples cover the most important parts of this new API.

Clock

Clock provides access to the current date and time. Clocks are aware of a timezone and may be used instead of System.currentTimeMillis() to retrieve the current milliseconds. Such an instantaneous point on the time-line is also represented by the class Instant. Instants can be used to create legacy java.util.Date objects.

Clock clock = Clock.systemDefaultZone();

long millis = clock.millis();

Instant instant = clock.instant();

Date legacyDate = Date.from(instant); // legacy java.util.Date

Timezones

Timezones are represented by a ZoneId. They can easily be accessed via static factory methods. Timezones define the offsets which are important to convert between instants and local dates and times.

System.out.println(ZoneId.getAvailableZoneIds());

// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");

ZoneId zone2 = ZoneId.of("Brazil/East");

System.out.println(zone1.getRules());

System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]

// ZoneRules[currentStandardOffset=-03:00]

LocalTime

LocalTime represents a time without a timezone, e.g. 10pm or 17:30:15. The following example creates two local times for the timezones defined above. Then we compare both times and calculate the difference in hours and minutes between both times.

LocalTime now1 = LocalTime.now(zone1);

LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2)); // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);

long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween); // -3

System.out.println(minutesBetween); // -239

LocalTime comes with various factory method to simplify the creation of new instances, including parsing of time strings.

LocalTime late = LocalTime.of(23, 59, 59);

System.out.println(late); // 23:59:59

DateTimeFormatter germanFormatter =

DateTimeFormatter

.ofLocalizedTime(FormatStyle.SHORT)

.withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);

System.out.println(leetTime); // 13:37

LocalDate

LocalDate represents a distinct date, e.g. 2014-03-11. It's immutable and works exactly analog to LocalTime. The sample demonstrates how to calculate new dates by adding or substracting days, months or years. Keep in mind that each manipulation returns a new instance.

LocalDate today = LocalDate.now();

LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);

LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);

DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();

System.out.println(dayOfWeek); // FRIDAY

Parsing a LocalDate from a string is just as simple as parsing a LocalTime:

DateTimeFormatter germanFormatter =

DateTimeFormatter

.ofLocalizedDate(FormatStyle.MEDIUM)

.withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);

System.out.println(xmas); // 2014-12-24

LocalDateTime

LocalDateTime represents a date-time. It combines date and time as seen in the above sections into one instance. LocalDateTime is immutable and works similar to LocalTime and LocalDate. We can utilize methods for retrieving certain fields from a date-time:

LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();

System.out.println(dayOfWeek); // WEDNESDAY

Month month = sylvester.getMonth();

System.out.println(month); // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE\_OF\_DAY);

System.out.println(minuteOfDay); // 1439

With the additional information of a timezone it can be converted to an instant. Instants can easily be converted to legacy dates of type java.util.Date.

Instant instant = sylvester

.atZone(ZoneId.systemDefault())

.toInstant();

Date legacyDate = Date.from(instant);

System.out.println(legacyDate); // Wed Dec 31 23:59:59 CET 2014

Formatting date-times works just like formatting dates or times. Instead of using pre-defined formats we can create formatters from custom patterns.

DateTimeFormatter formatter =

DateTimeFormatter

.ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);

String string = formatter.format(parsed);

System.out.println(string); // Nov 03, 2014 - 07:13

Unlike java.text.NumberFormat the new DateTimeFormatter is immutable and thread-safe.

For details on the pattern syntax read here.

Annotations

Annotations in Java 8 are repeatable. Let's dive directly into an example to figure that out.

First, we define a wrapper annotation which holds an array of the actual annotations:

@interface Hints {

Hint[] value();

}

@Repeatable(Hints.class)

@interface Hint {

String value();

}

Java 8 enables us to use multiple annotations of the same type by declaring the annotation @Repeatable.

Variant 1: Using the container annotation (old school)

@Hints({@Hint("hint1"), @Hint("hint2")})

class Person {}

Variant 2: Using repeatable annotations (new school)

@Hint("hint1")

@Hint("hint2")

class Person {}

Using variant 2 the java compiler implicitly sets up the @Hints annotation under the hood. That's important for reading annotation informations via reflection.

Hint hint = Person.class.getAnnotation(Hint.class);

System.out.println(hint); // null

Hints hints1 = Person.class.getAnnotation(Hints.class);

System.out.println(hints1.value().length); // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);

System.out.println(hints2.length); // 2

Although we never declared the @Hints annotation on the Person class, it's still readable via getAnnotation(Hints.class). However, the more convenient method is getAnnotationsByType which grants direct access to all annotated @Hint annotations.

Furthermore the usage of annotations in Java 8 is expanded to two new targets:

@Target({ElementType.TYPE\_PARAMETER, ElementType.TYPE\_USE})

@interface MyAnnotation {}

**Rules for Exception handling w.r.t Method Overriding in Java**

We will consider **different cases** for discussing the rules for exception handling with respect to method overriding

1. If parent class method **doesn’t**declare any exception
2. If parent class method declares**unchecked**exception
3. If parent class method declares**checked**exception
4. If parent class method declares both**checked**&**unchecked**exceptions

**Rule 1:** If parent class method **doesn’t**declare any exception

1. Then child class overriding method can declare **any type of unchecked exception** (this is the only possibility)
2. If child class overriding method declares checked exception, then compiler throws compile-time error stating “**Exception IOException is not compatible with throws clause in ParentClass.testMethod()**”
3. Then child class overriding method can declare **no exception** in the overriding method of child class (very much same as that of overridden method of parent class –> exactly same method signature)

**Rule 2:** If parent class method declares**unchecked**exception

1. Then child class overriding method can declare **any type of unchecked exception** (not necessarily same exception as that of parent class method –> only for unchecked exception)
2. If child class overriding method declares any checked exception, then compiler throws compile-time error stating “**Exception IOException is not compatible with throws clause in ParentClass.testMethod()**”
3. Then child class overriding method can declare **no exception** in the overriding method of child class

**Rule 3:** If parent class method declares**checked**exception

1. Then child class overriding method can declare **any type of unchecked exception**
2. Then child class overriding method can declare **same type of checked exception or one of its sub-class or noexception**  
   OR, sub -type of declared checked exception
3. Then child class overriding method can declare **no exception** in the overriding method of child class

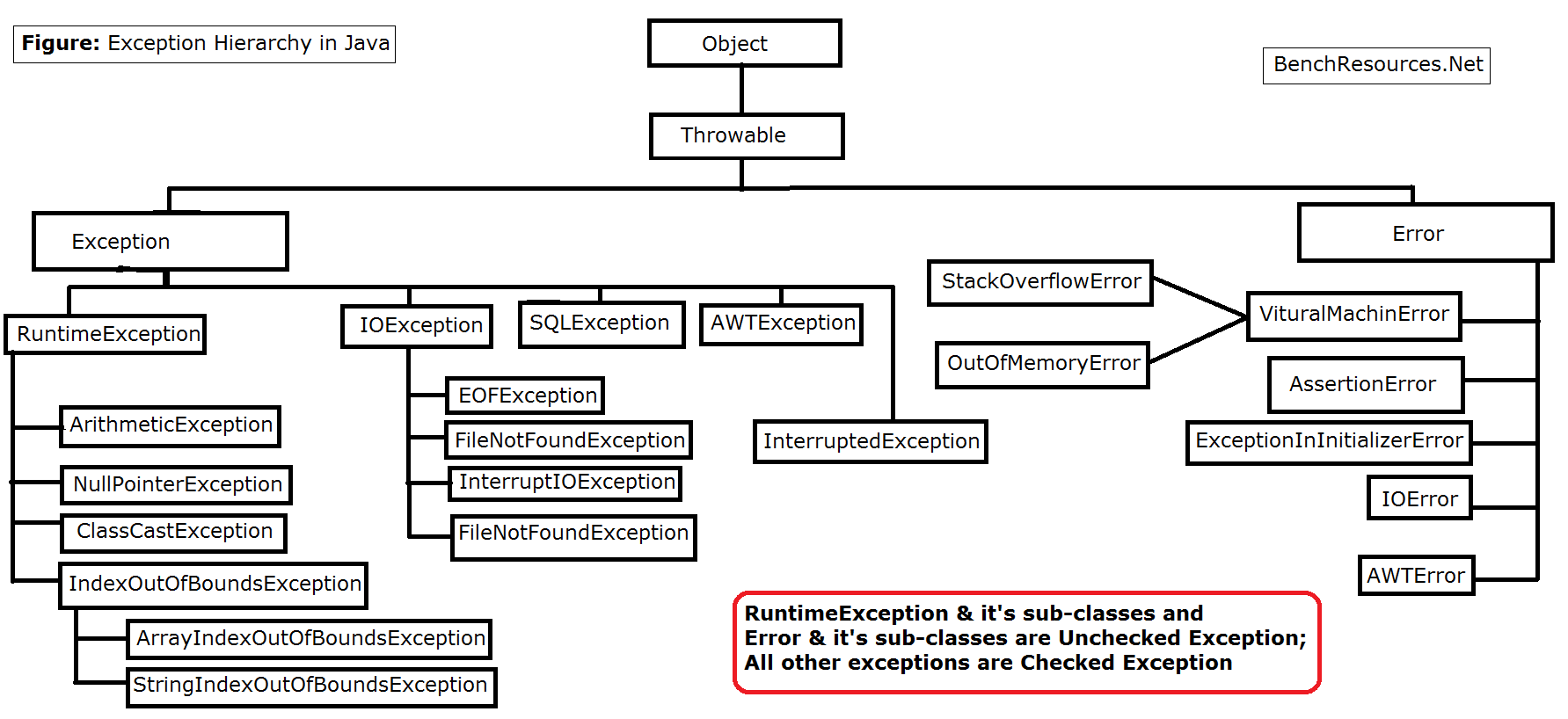
**Rule 4:** If parent class method declares combination of both**checked**&**unchecked**exceptions

1. Then child class overriding method can declare **any type of unchecked exception**
2. Then child class overriding method can declare **same type of checked exception or one of its sub-class or no exception**
3. Then child class overriding method can declare **no exception** in the overriding method of child class

**Conclusion:**

* When parent class method declares **no exception**, then child class overriding method can declare  
  1. No exception or  
  2. Any number of unchecked exception (but strictly no checked exception)
* When parent class method declares **unchecked exception**, then child class overriding method can declare  
  1. No exception or  
  2. Any number of unchecked exception (but strictly no checked exception)
* When parent class method declares **checked exception**, then child class overriding method can declare  
  1. No exception or  
  2. Same checked exception or  
  3. Sub-type of checked exception or  
  4. any number of unchecked exception
* All above conclusion hold true, even if combination of both checked & unchecked exception is declared in parent class method

# Exception Handling



**Q) Whether it is possible to write any statements after finally block ?**

* If there is no return statement for a method, then it is valid to write any valid statements after finally block
* But if there is a method that returns a value then writing any statement after finally block results in compile-time error
* If there is a return statement after finally block, then it is valid

**Q) Whether exception raises from catch block ?**

* It is very much possible that, code inside catch block too raises exception and this need to be handled
* Otherwise, program terminates abnormally

**Can we declare unchecked exception using throws keyword in method signature ?**

* Yes, it is possible to declare unchecked exception using throws clause

**Q) Whether it is valid to throw Java object, which isn’t extending any Exception/Error from exception hierarchy ?**

* As explained in the above question, only exception can be thrown which should extend any one of the types of Throwable class
* Throwing normal Java object which isn’t extending any exception-type from exception hierarchy will results in compile-time error stating “***incompatible types***”

**Q) Explain, what are the new features introduced in Java 1.7 version ?**

* New featured introduced in Java 1.7 version are,
* [***try-with-resources***](http://www.benchresources.net/try-with-resources-in-java-1-7-version/) for automatic resource management
* [***multi-catch block***](http://www.benchresources.net/multi-catch-block-in-java-1-7-version/) for grouping different exception-type for similar handler code with pipe character separating them

**Q) Explain Automatic Resource management feature in Java exception handling ?**

**try-with-resources statement:**

* Using ***try-with-resources*** statement, programmer doesn’t need to ***explicitly*** ***close*** the opened resources
* Rather it will be ***automatically closed*** once control reaches the ***end of try-catch block***
* This new feature introduced in ***Java 1.7***version is alternatively referred as ***Automatic Resource Management***e.;***ARM***
* Read [**try-with-resources statement**](http://www.benchresources.net/try-with-resources-in-java-1-7-version/) for more detail with example

**Rules:**

* All resources declared as part of ***try-with-resources*** statement must be ***AutoCloseable*** (i.e.; all resources must implements ***java.lang.AutoCloseable*** interface)
* ***Multiple resources*** can be declared inside try block argument; but they are all must be ***separated*** by ***semi-colon*** (;)
* While using ***try-with-resources*** statement, try block itself is enough. There is ***no compulsion*** to write either ***catch block*** or ***finally block*** following the ***try block***, whereas in prior versions try block must be followed by either catch block or finally block
* All ***resource reference variable*** declared inside try block argument are ***implicitly final***. Therefore, resource reference variable ***can’t changed***or***re-assigned*** within try block

**Q) Whether it is mandatory to follow catch block or finally block, after try-with-resources statement (try block) ?**

* It isn’t mandatory to have either catch block or finally block following try block
* try block alone can work without the need of catch block or finally block

**Q) How is multi-catch block is useful over traditional multiple catch blocks ?**

**Multi-catch block:**

* In ***Java 1.6*** or lesser version, whenever ***multiple exception***is***thrown***, then programmer has to provide ***multiple catch block***to catch different types of exception, although exception handling code is same
* But in ***Java 1.7*** version, we can write/code ***single catch block*** to ***handle*** multiple types of exceptions using ***multi-catch block***
* By using ***multi-catch block***, helps to provide same handler code by grouping different exception-types. And program/code becomes more readable with lesser lines of code
* Read [**Multi catch block in Java 1.7 version**](http://www.benchresources.net/multi-catch-block-in-java-1-7-version/) for more detail with example

**Rules:**

* There ***shouldn’t*** be any ***relationship*** between the ***declared exception-type*** in ***multi-catch block***.
* Otherwise, compile-time error will be thrown stating “***The exception <child-exception-type> is already caught by the alternative <parent-exception-type>***”
* If a catch block ***handles***more than one***exception-type*** (i.e.; multi-catch block), then exception variable is ***implicitly final***
* Any ***changes***or***re-assignment*** to this implicit final variable within catch block results in ***compile-time error***

# Collections

|  |  |  |  |
| --- | --- | --- | --- |
|  | Property | ***java.lang.Comparable*** | ***java.util.Comparator*** |
| 1 | Comparing instances of class | Comparable is used to compare instances of same class in java. | Comparator can be used to compare instances of same or different classes in java. |
| **2** | **sorting order** | Comparable can be implemented by class which need to define a **natural ordering for its objects.**  **Example** - String, Integer, Long , [Date](http://www.javamadesoeasy.com/2015/07/creating-date-in-java-using-calendar.html) and all other wrapper classes implements Comparable. | Comparator is implemented when one wants a **different sorting order** and define custom way of comparing two instances. |
| 3 | Changes to class | For using Comparable, original Class must implement it.    **Example of Comparable in java-**  **class** Employee **implements Comparable<Employee>**    For using Comparable, Employee Class must implement it, no other class can implement it.  As used in **Program 1** | Class itself can implement Comparator or  any other class can implement Comparator. Hence avoiding modification to original class.  **Example of Comparator in java-**  **class ComparatorName implements Comparator<Employee>**  **class ComparatorId implements Comparator<Employee>**  In above example modifications were made to **ComparatorName** and **ComparatorId.** Hence avoiding modification to Employee class.  As used in **Program 4** |
| 4 | Sorting on basis on one or many criteria in java | Provides sorting only on **one** criteria, **because** Comparable can be implemented by original class only in java. | We can use Comparator to sort class on **many** criterias **because** class itself or any other class can implement Comparator in java. |
| 5 | Method | After implementing Comparable class must override **compareTo** method  **@Override**  **public int compareTo(Employee obj) {**  **//sort Employee on basis of name(ascending order)**  **return this.name.compareTo(obj.name);**  **}**  Method compares **this** with **obj** object and returns a integer.  positive – **this** is **greater** than **obj**  zero – **this** is **equal** to **obj**  negative – **this** is **less** than **obj**    As used in **Program 1** | After implementing Comparable class must override **compare** method  **@Override**  **public int compare(Employee obj1, Employee obj2) {**  **//sort Employee on basis of name(ascending order)**  **return obj1.name.compareTo(obj2.name);**  **}**    Method compares **obj1** with **obj2** object and returns a integer.  positive – **obj1** is **greater** than **obj2**  zero – **obj1** is **equal** to **obj2**  negative – **obj1** is **less** than **obj2**  As used in **Program 3** |
| 6 | Package | **java.lang**  **java.lang** package is automatically imported by every program in java.  Hence, we need **not** to write explicit statement for importing java.lang.Comparable. | **java.util**  We need to write explicit import statement -  **import** java.util.Comparator |
| 7 | Using **Collections.sort** | Let's say we wanna sort list of Employee,  **Collections.sort(**list**)** uses Comparable interface for sorting class.  As used in Program 1 | Let's say we wanna sort list of Employee,  **Collections.*sort*(list,new ComparatorName());**  uses Comparator interface for sorting class.  As used in Program 5 |

Iterator returned by few Collection framework Classes are fail-safe, means any structural modification made to these classes during iteration won’t throw any Exception.

Some important classes whose returned iterator is fail-safe >

[CopyOnWriteArrayList](http://www.javamadesoeasy.com/2015/04/arraylist-vs-copyonwritearraylist.html)

[CopyOnWriteArraySet](http://www.javamadesoeasy.com/2015/04/hashset-vs-copyonwritearrayset.html)

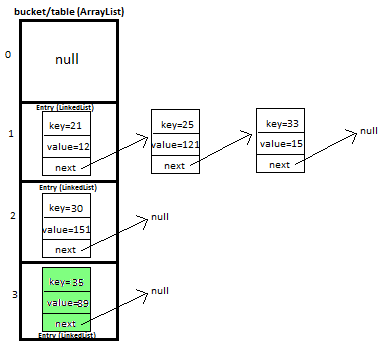
[ConcurrentSkipListSet](http://www.javamadesoeasy.com/2015/04/treeset-vs-concurrentskiplistset.html)

ConcurrentHashMap is divided into different segments based on concurrency level. So different threads can access different segments concurrently in java.

Can threads read the segment locked by some other thread?

Yes. When thread locks one segment for updation it does not block it for retrieval (done by get method) hence some other thread can read the segment (by get method), but it will be able to read the data before locking in java.

For operations such as putAll concurrent retrievals may reflect removal of only some entries in java.For operations such as clear concurrent retrievals may reflect removal of only some entries in java  
1) Custom HashMap in java >



In this tutorial we will learn how to create and implement own/custom [HashMap](http://www.javamadesoeasy.com/2015/04/hashmap-in-java.html) in java with full working source code.

This is very important and trending topic in java. In this post i will be explaining HashMap custom implementation in lots of detail with diagrams which will help you in visualizing the HashMap implementation. This is must prepare topic for interview and from knowledge point of view as well.

I will be explaining how we will put and get key-value pair in HashMap by overriding-

>equals method - helps in checking equality of entry objects.

>hashCode method - helps in finding bucket’s index on which data will be stored.

We will maintain bucket ([ArrayList](http://javamadesoeasy.com/2015/02/arraylist-custom-implementation.html)) which will store Entry ([LinkedList](http://javamadesoeasy.com/2015/01/doublylinkedlist-insert-and-delete-at.html)).

2) Entry<K,V>

We store key-value pair by using Entry<K,V>

Entry contains

K key,

V value and

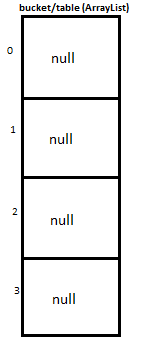
Entry<K,V> next  (i.e. next entry on that location of bucket).

|  |
| --- |
| static class Entry<K, V> {         K key;         V value;         Entry<K,V> next;           public Entry(K key, V value, Entry<K,V> next){             this.key = key;             this.value = value;             this.next = next;         }     } |

3) Putting 5 key-value pairs in own/custom HashMap (step-by-step)

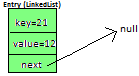
I will explain you the whole concept of HashMap by putting 5 key-value pairs in HashMap.

Initially, we have bucket of capacity=4. (all indexes of bucket i.e. 0,1,2,3 are pointing to null)

****

Let’s put first key-value pair in HashMap-

Key=21, value=12

newEntry Object will be formed like this >

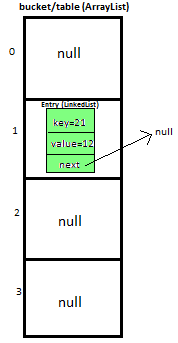
We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 21%4= 1.

So, 1 will be the index of bucket on which newEntry object will be stored.

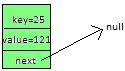
We will go to 1st index as it is pointing to null we will put our newEntry object there.

At completion of this step, our HashMap will look like this-



Let’s put second key-value pair in HashMap-

Key=25, value=121

newEntry Object will be formed like this >

We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 25%4= 1.

So, 1 will be the index of bucket on which newEntry object will be stored.

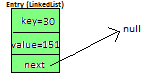
We will go to 1st index, it contains entry with key=21, we will compare two keys(i.e. compare 21 with 25 by using equals method), as two keys are different we check whether entry with key=21’s next is null or not, if next is null we will put our newEntry object on next.

At completion of this step our HashMap will look like this-



Let’s put third key-value pair in HashMap-

Key=30, value=151

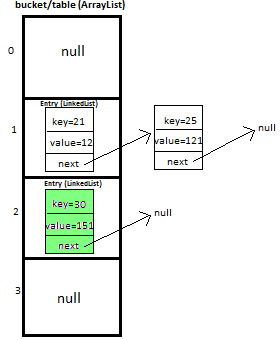
newEntry Object will be formed like this >

We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 30%4= 2.

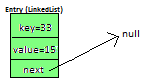
So, 2 will be the index of bucket on which newEntry object will be stored.We will go to 2nd  index as it is pointing to null we will put our newEntry object there.

At completion of this step, our HashMap will look like this-



Let’s put fourth key-value pair in HashMap-

Key=33, value=15

Entry Object will be formed like this >

We will calculate hash by using our hash(K key) method - in this case it returns

key/capacity= 33%4= 1,

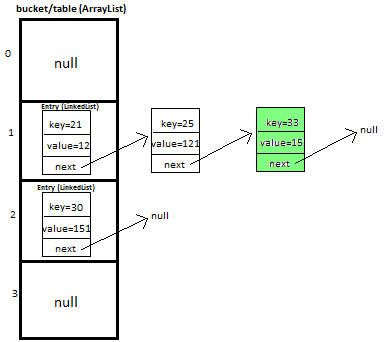
So, 1 will be the index of bucket on which newEntry object will be stored.

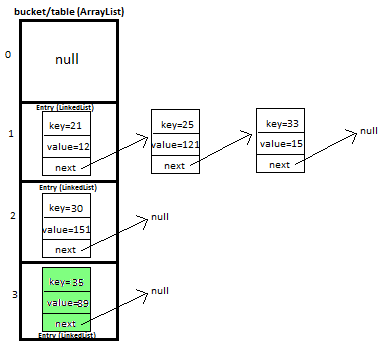
We will go to 1st index -

>it contains entry with key=21, we will compare two keys (i.e. compare 21 with 33 by using equals method, as two keys are different,  proceed to next  of entry with key=21 (proceed only if next is not null).

>now, next contains entry with key=25, we will compare two keys (i.e. compare 25 with 33 by using equals method, as two keys are different,  now next of entry with key=25 is pointing to null so we won’t proceed further, we will put our newEntry object on next.

At completion of this step our HashMap will look like this-

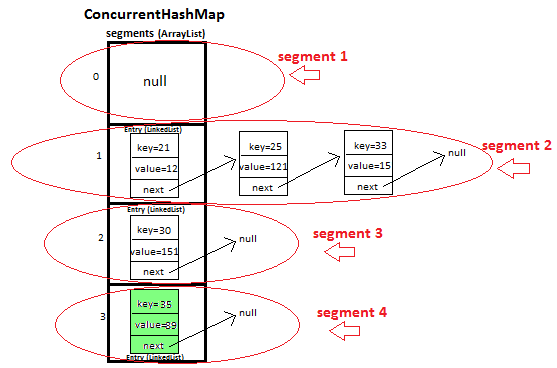
  
Let’s put fifth key-value pair in HashMap-

**Key=35, value=89Repeat above mentioned steps.**At completion of this step our HashMap will look like this-****

**Segments** in ConcurrentHashMap with **diagram** >

we have ConcurrentHashMap with **4 segments -**

(Diagram shows how **segments** are formed in ConcurrentHashMap)

****

Now let’s form few questions to clear your doubts (based on above diagram) >

Question 1 : What will happen map.put(25,12) is called and some other thread concurrently calls map.get(25)?

Answer : When map.put(25,12) is called segment 2 will be locked,

key=25 also lies in segment 2, When thread locks one segment for updation it does not block it for retrieval hence some other thread can read the same segment, but it will be able to read the data before locking (hence map.get(25) will return 121)

Question 2 : What will happen map.put(25,12) is called and some other thread concurrently calls map.get(33)?

Answer : When map.put(25,12) is called segment 2 will be locked,

key=33 also lies in segment 2, When thread locks one segment for updation it does not block it for retrieval hence some other thread can read the same segment, but it will be able to read the data before locking (hence map.get(33) will return 15)

Question 3 : What will happen map.put(25,12) is called and some other thread concurrently calls map.put(33,24)?

Answer : When map.put(25,12) is called segment 2 will be locked,

key=33 also lies in segment 2, When thread locks one segment for updation it does not allow any other thread to perform updations in same segment until lock is not released on segment.

hence map.put(33,24) will have to wait for map.put(25,12) operation to release lock on segment.

Question 4 : What will happen map.put(25,12) is called and some other thread concurrently calls map.put(30,29)?

Answer : When map.put(25,12) is called segment 2 will be locked,

but key=30 lies in segment 3.

Both the kays lies in different segments, hence both operations can be performed concurrently.

Question 5 : What will happen updations (put/remove) are in process in certain segments and new key-pair have to be put/remove in same segment ?

Answer : When updations are in process thread locks the segment and it does not allow any other thread to perform updations (put/remove) in same segment until lock is not released on segment.

load() :   
- Use load() method only when you are sure that object you want to read already exists.   
- If unique Id of an object does not exists in database then load() method will throw an exception.  
- load() method return proxy object default and database won't be hit until the proxy is first invoked.  
get() :  
- Use get() method when you are not sure obout the object existance in the database.  
- If object does not exists in the database, the get() method will return null.  
- get() method will hit database immediately.  
sess.refresh(cat); //re-read the state (after the trigger executes)

s**orting HashSet – 2 ways:**

1. Using **Collections.sort(list);** method
2. Using **TreeSet** inter-conversion constructor

Let us move forward to discuss 2 ways of sorting HashSet contents

**Way 1:**Using Collections.sort(list); method

**Steps:**

1. Create new HashSet object
2. Store HashSet contents into ArrayList using inter-conversion constructor
3. Finally, invoke Collections.sort(al); method to sort elements in ascending order
4. **Note:** similarly elements can be sorted in descending order as well using Comparator

**Syntax:**

[?](http://www.benchresources.net/how-to-sort-hashset-in-java-2-ways/)

|  |  |  |  |
| --- | --- | --- | --- |
| 1  2  3 | List<String> al = new ArrayList<String>(hsObject);  Collections.sort(al);  Way 2: Using TreeSet inter-conversion constructor  Steps:  Create new HashSet object  Create TreeSet object and pass HashSet contents as constructor-argument to TreeSet  That’s it, HashSet are sorted and stored inside new TreeSet object  Note: similarly elements can be sorted in descending order as well using Comparator  Syntax:  [?](http://www.benchresources.net/how-to-sort-hashset-in-java-2-ways/)   |  |  | | --- | --- | | 1 | TreeSet<String> set = new TreeSet<String>(hashSet); |     **Various ways to remove duplicate elements from Arrays:**  using List implemented classes  using Set implemented classes  using combination of both List & Set implemented classes  without using any collection classes  using Streams class in Java 8    **Way 1:**Using List implemented classes (i.e.; ArrayList class)  **Steps:**  Iterate through original Arrays to read duplicate elements  Initialize ArrayList (i.e.; to store unique elements, after checking)  While iterating String Array, check whether element already present in the unique list (created in step-2)  // check whether list contains duplicate, while iterating              if(!uniqueList.contains(strArray[index])) {  If it doesn’t contains inside unique List, then add element to unique List  Repeat step 3-4, until all elements of Arrays are compared and unique elements are stored inside List  Convert unique list into Arrays using toArray() method  Again, iterate through Arrays to print unique elements  Way 2: Using Set implemented classes (i.e.; HashSet class)  Steps:  Iterate through original Arrays to read duplicate elements  Initialize HashSet (i.e.; to store unique elements, while iterating)  While iterating String Array, just simply add elements to HashSet; because Set allows only unique items thereby removing duplicate items  Convert Set into Arrays using toArray() method  Again, iterate through Arrays to print unique elements  Difference between above 2 implemented approaches:  In the 1st way, we have to check manually for each items inside String Arrays with unique List and then add to List for each iteration  Whereas for Set implemented classes, we don’t need to do any comparison or checking because Set allows only unique items thereby removing duplicate items  In addition to this, 1st approach maintains insertion order whereas 2nd approach follows random order as HashSet internally uses hashing algorithm to store elements    Way 3: Using both List & Set classes  Steps:  Iterate through original Arrays to read duplicate elements  Convert Arrays into List; using Arrays’ asList(arrObj); method  Add converted List object into HashSet using inter-conversion collection constructor; for removing duplicates  Create new Arrays of required data-type  Convert Set to Arrays (using newly created Arrays in step-4)  // convert Arrays into List  List<String> lst = Arrays.asList(strArray);    // again convert List into Set, for removing duplicates  // using inter-conversion constructor  Set<String> set = new HashSet<String>(lst);    // create new String[] with no. of elements inside Set  String[] uniqueArr = new String[set.size()];    // convert back Set into Arrays  set.toArray(uniqueArr);  Way 4: Without using any collection classes  Steps:  Iterate through original Arrays to read duplicate elements  Construct outer for-loop for iterating Arrays elements  Construct inner for-loop for iterating Arrays elements starting from 2nd position (or 1st index)  Within inner for-loop, check whether outer for-loop element with inner for-loop elements (while iterating)  If comparison comes out to be true, then assign last element at this position (thereby replacing duplicate element)  While doing step 5, decrement the size of Arrays by 1 as well as decrement inner for-loop count (this will give unique elements inside Arrays after all iterations are completed)  Finally print Arrays elements again  Way 5: Using Streams class in Java 8  Steps:  Iterate through original Arrays to read duplicate elements  Using Streams class, remove duplicate elements (i.e.; using distinct() method)  At the same time, convert back to Object[] arrays by invoking toArray() method  Finally print Arrays elements again  // convert to unique/distinct Arrays using Java 8 Streams class          Object[] uniqueArrays = Arrays.stream(strArray).distinct().toArray();    Remove duplicate elements from ArrayList in Java  Solution:  using collection class (i.e.; convert ArrayList to HashSet class to remove duplicates)  without using any collection class (i.e.; using 2 for-loop iterations)  Using Collection class:  Steps:  Iterate through original ArrayList to read duplicate elements  Create HashSet (using inter-conversion collection constructor)  Add ArrayList object to constructor argument of HashSet (to remove duplicates)  Reason: Set allows only unique elements   // outer for-loop          for(int outForLoop = 0; outForLoop < techCompanies.size(); outForLoop++) {              // inner for-loop              for(int inForLoop = outForLoop + 1; inForLoop < techCompanies.size(); inForLoop++) {                  // check whether, it already contains this element                  if(techCompanies.get(outForLoop).equals(techCompanies.get(inForLoop))) {                      // remove, if its already duplicate element                      techCompanies.remove(inForLoop);                  }            }        } |

**Without using Collection class:**

**Steps:**

Iterate through original ArrayList to read duplicate elements

Construct outer for-loop for iterating ArrayList elements

Construct inner for-loop for iterating ArrayList elements starting from 2nd position (or 1st index)

Within inner for-loop, check whether outer for-loop element with inner for-loop elements (while iterating)

Remove element from ArrayList, if it is found to be same; otherwise repeat step until both for-loop iteration gets completed

Note: we will count of number of duplicate elements present in List using Collections class’s utility  public static int frequency(Collection c, Object o) **method**

// remove duplicates, maintaining insertion order

// convert to LinkedHashSet

Collection<String> lhs = new LinkedHashSet<String>(ArrayList0bject);

**Note: ConcurrentModificationException**will be thrown when **one thread** is iterating and **other thread** is trying to modify ArrayList contents (i.e.; add or remove)

* Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered. For example Roll Numbers of students. Whereas, Comparator interface sorting is done through a separate class.
* Logically, Comparable interface compares “this” reference with the object specified and Comparator in Java compares two different class objects provided.
* If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and objects will be sorted based on there natural order defined by CompareTo method.

***To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if you sorting needs to be done on attributes of different objects, then use Comparator in Java.***

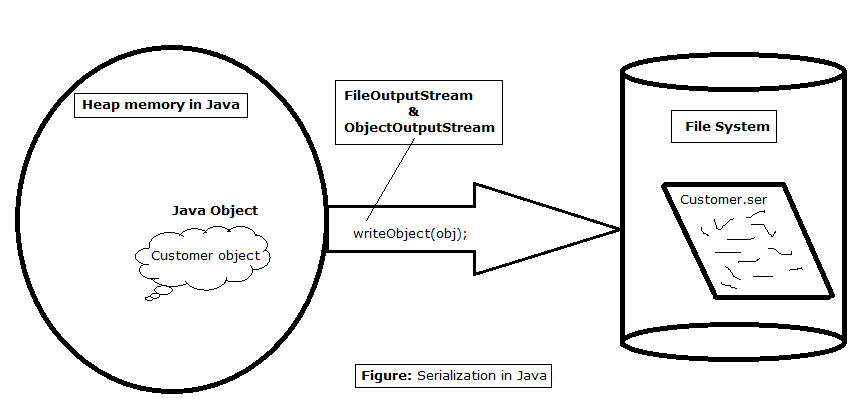
# **Serialization**

* **Rule 1:** all classes that need to be serialized must implement ***java.io.Serializable*** interface
* **Rule 2:** All reference classes inside a serializable class must be ***java.io.Serializable***
* **Rule 3:** If any of the class is not implementing ***java.io.Serializable***in the serialization process***,*** then JVM will throw ***NotSerializableException***

**Serialization** is a process of saving the state of an Object to File, only instance variables will be participated and persisted to file storage or some other storage via network capability

**De-Serialization** is a process of restoring Object’s state back from file storage to Java heap memory

* The process of **writing a state of an Object**to a file is called ***Serialization***
* In other words, the ***process of saving an Object’s state to a file*** is called ***Serialization***
* But practically, it is the ***process of converting & storing Java Object’s state from heap memory*** (in byte stream)**to file supported form** (in binary format)

[](http://www.benchresources.net/wp-content/uploads/2016/10/1-Serialization-in-java.png)

  fos = new FileOutputStream("Customer.ser");

            // converting java-object to binary-format

            oos = new ObjectOutputStream(fos);

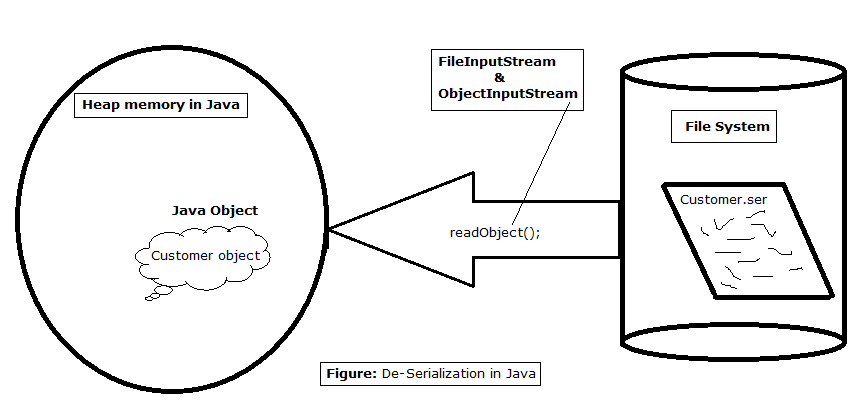
**Customer class**

* Customer class is the one to be ***serialized***
* Therefore, it is must to implement ***java.io.Serializable*** interface
* Consists of **3 member variables** namely
* Two integer member (customer id and customer age) and String member (customer name)

**Note:** Object must implement **java.io.Serializable**, otherwise Run time exception will be thrown saying **NotSerializableException**

**De-Serialization:**

* The ***process of reading a state of an Object from a file*** is called **De-Serialization**
* But practically, it is the **process of converting & re-storing Java Object’s state into heap memory from file supported form**(which is in binary format)

**[](http://www.benchresources.net/wp-content/uploads/2016/10/2-DeSerialization-in-java.png)**

 fis = new FileInputStream("Customer.ser");

            // converting binary-data to java-object

            ois = new ObjectInputStream(fis);

            // reading object's value and casting to Customer class

            customer = (Customer) ois.readObject();

Using **ObjectInputStream** and **FileInputStream** classes available from **java.io** package, we can***de-serialize*** an Object from file

**Note:** we will use **Customer object** from above example to de-serialize (and make sure class implements **java.io.Serializable**interface)

Serialization with Inheritance

When ***super*** ***class*** is serialized, only ***properties of super class*** will be serialized

When ***sub*** ***class*** is serialized, ***properties of sub class***as well as***inherited properties of super class*** will also be serialized

**Case 1:** ***Super* *class***implements ***java.io.Serializable*** but ***sub* *class***doesn’t implement ***java.io.Serializable***

When super class is serializable, then ***any class extending super class*** will also be serializable ***by default***(inheritance principle)

So, here sub class ***not required*** to implement ***java.io.Serializable*** explicitly

When ***sub class is serialized***, then sub class properties as well as ***inherited***super class properties will also be ***serialized***(during serialization process)

**Note:** To ***prevent sub class from serializing by default***, then we need to override ***writeObject()*** and ***readObject()*** methods

**Case 2:** ***Sub* *class***implements ***java.io.Serializable*** but ***super* *class***doesn’t implement ***java.io.Serializable***

Before moving ahead, we should understand ***is it possible to serializable sub class***, if its ***super class isn’t serializable?***

The answer is ***yes***, because if the condition to serialize any class on the basis of its super classes implementing ***java.io.Serializable*** interface, then ***no class in Java can be serialized***

**Reason:** *java.lang.Object* is the base class for any class defined in Java, and it ***doesn’t***implements ***java.io.Serializable***interface

In that way, it is very well possible to serialize a sub class even if its super class ***doesn’t***implement ***java.io.Serializable***interface

**Serialization process:**

While serializing sub class, ***JVM will check if there are any super class***which is not implementing ***java.io.Serializable***interface

Then, inheriting instance variables of ***non-serializable*** super class will be stored to ***default value*** ignoring their original values

Like 0 for Integer, null for String, etc

**De-Serialization process:**

While de-serializing sub class, ***JVM will check if there are any non-serializable*** super class

Then, it will ***execute instance initialization*** flow (i.e.; similar to object instantiation flow)

**1st check:** if there are direct initialization at ***instance variable declaration***

**2nd check:** if there are any***initialization block*** for instance variable assignment

**3rd check:** invokes ***no-argument constructor*** and looks for instance variable assignment

To execute the ***3rd check***, non-serializable super class requires ***no-argument constructor***

**Exception:** otherwise ***InvalidClassException***will be thrown

|  |  |
| --- | --- |
| **Serializable** | **Externalizable** |
| Serializable is a marker interface which doesn’t contain any methods and JVM provides special ability during serialization process | Externalizable is a sub-interface of Serializable interface and contains 2 methods viz.; readExternal() and writeExternal() |
| During Serialization process, all member variables of an object is serialized (even if some of the variables not required to be serialized) | But in Externalization, programmer has to provide serialization logic |
| That’s why, it is referred as default serialization | This is referred as custom serialization, as programmer has to write custom logic for serialization to happen |
| From above stated points, it is clear that JVM takes the complete control over serialization process | Programmer has complete control over serialization process (to write custom logic for the required variables to be serialized) |
| Performance-wise, Serializable is relatively low as complete object need to be serialized, even if we require only partial object | Performance is high in extenalizable, as programmer design what all required variable need to be serialized |
| Doesn’t require any public no-argument constructor for serializable | Public no-argument constructor is very must in externalizable, otherwise InvalidClassException is thrownThis is mainly required during readExternal() method; that is while restoring object back to heap memory from file storage |
| For variable that needn’t to be serialized use transient modifier (but still its default value is stored into file)Transient modifier play a very important role in serializable | Variable with transient modifier not required; as programmer can write custom logic to ignore those variables which is not requiredSo, transient modifier doesn’t play any important role in externaizable |
| This is the best suit; when whole or complete object required to be serialized to the file storage | This is the best suit; when partial object or few of the member variables of an object need to be serialized to the file storage |

Note: For any other case, constructor is not invoked with only exception being for non-serializable super class

Important points to remember while Serialization with Inheritance:

If super class implements java.io.Serializable interface, then all sub class is also serializable by default

It is possible to serialize sub class, even if its corresponding super class doesn’t implements java.io.Serializable interface

While serializing sub class whose super class doesn’t implements io.Serializable interface, then during serialization process inheriting instance variables of non-serializable super class will be stored to default value ignoring their original values (like 0 for Integer, null for String, etc)

During de-serialization process, JVM will execute instance initialization flow in 3 steps i.e.;  
1st checks direct variable assignment,  
2nd check inside initialization block and  
3rd check inside no-argument constructor

For the 3rd check, it is very must to code a no-argument constructor inside non-serializable super class

Otherwise, InvalidClassException will be thrown at run time

**serialVersionUID**

In addition to implementing the Serializable interface, a class intended for serialization should also contain a private static final long variable named serialVersionUID.

Here is the Person class from before, with a serialVersionUID variable added:

import java.io.Serializable;

public static class Person implements Serializable {

private static final long serialVersionUID = 1234L;

public String name = null;

public int age = 0;}

The serialVersionUID variable is used by Java's object serialization API to determine if a deserialized object was serialized (written) with the same version of the class, as it is now attempting to deserialize it into.

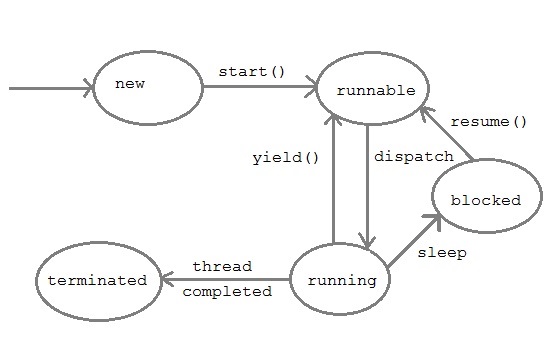
Imagine that a Person object is serialized to disk. Then a change is made to the Person class. Then you try to deserialize the stored Person object. Now the serialized Person object may not correspond to the new version of the Person class.

To detect such problems a class implementing Serializable should contain a serialVersionUID field. If you make big changes to the class, you should also change its serialVersionUID value.

The Java SDK and many Java IDEs contains tools to generate the serialVersionUID so you don't have to.

# THREAD

Life cycle of a Thread



New : A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.

Runnable : After invocation of start() method on new thread, the thread becomes runnable.

Running : A thread is in running state if the thread scheduler has selected it.

Waiting : A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.

Terminated : A thread enter the terminated state when it complete its task.

Thread Priorities

Every thread has a priority that helps the operating system determine the order in which threads are scheduled for execution. In java thread priority ranges between 1 to 10,

MIN-PRIORITY (a constant of 1)

MAX-PRIORITY (a constant of 10)

By default every thread is given a NORM-PRIORITY(5). The main thread always have NORM-PRIORITY.

Note: Thread priorities cannot guarantee that a higher priority thread will always be executed first than the lower priority thread. The selection of the threads for execution depends upon the thread scheduler which is platform dependent.

The Logic Behind The Synchronization In Java :The synchronization in java is built around an entity called **object lock**or**monitor**. Here is the brief description about lock or monitor.

Whenever an object is created to any class, an object lock is created and is stored inside the object.

One object will have only one object lock associated with it.

Any thread wants to enter into synchronized methods or blocks of any object, they must acquire object lock associated with that object and release the lock after they are done with the execution.

The other threads which wants to enter into synchronized methods of that object have to wait until the currently executing thread releases the object lock.

To enter into static synchronized methods or blocks, threads have to acquire class lock associated with that class as static members are stored inside the class memory.

Synchronized Blocks :

Some times, you need only some part of the method to be synchronized not the whole method. This can be achieved with synchronized blocks. Synchronized blocks must be defined inside a definition blocks like methods, constructors, static initializer or instance initializer.

synchronized block takes one argument and it is called **mutex**. if synchronized block is defined inside non-static definition blocks like non-static methods, instance initializer or constructors, then this mutex must be an instance of that class. If synchronized block is defined inside static definition blocks like static methods or static initializer, then this mutex must be like ClassName.class.

**1)** You can use **synchronized** keyword only with methods but not with variables, constructors, static initializer and instance initializers.

**2)** Constructors, Static initializer and instance initializer can’t be declared with synchronized keyword, but they can contain synchronized

**3)** Both static and non-static methods can use synchronized keyword. For static methods, thread need class level lock and for non-static methods, thread need object level lock.

**4)** It is possible that both static synchronized and non-static synchronized methods can run simultaneously. Because, static methods need class level lock and non-static methods need object level lock.

**5)** A method can contain any number of synchronized blocks. This is like synchronizing multiple parts of a method.

Synchronization blocks can be nested.

|  |  |
| --- | --- |
| [?](http://javaconceptoftheday.com/synchronization-in-java/)1  2  3  4  5  6  7 | synchronized (this)  {      synchronized (this)      {          //Nested synchronized blocks      }} |

**7)** Lock acquired by the thread before executing a synchronized method or block must be released after the completion of execution, no matter whether execution is completed normally or abnormally (due to exceptions).

**8)** Synchronization in java is **Re-entrant in nature**. A thread can not acquire a lock that is owned by another thread. But, a thread can acquire a lock that it already owns. That means if a synchronized method gives a call to another synchronized method which needs same lock, then currently executing thread can directly enter into that method or block without acquiring the lock.

**9)** synchronized method or block is very slow. They decrease the performance of an application. So, special care need to be taken while using synchronization. Use synchronization only when you needed it the most.

**10)** Use synchronized blocks instead of synchronized methods. Because, synchronizing some part of a method improves the performance than synchronizing the whole method.

[**Differences between implementing Runnable interface and extending Thread class**](http://www.javamadesoeasy.com/2015/03/differences-between-implementing.html) **-**

Multiple inheritance in not allowed in java : When we [implement Runnable](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) interface we can extend another class as well, but if we extend Thread class we cannot extend any other class because java does not allow multiple inheritance. So, same work is done by implementing Runnable and [extending Thread](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) but in case of implementing Runnable we are still left with option of extending some other class. So, it’s better to implement Runnable.

[Thread safety](http://www.javamadesoeasy.com/2015/03/guidelines-to-thread-safe-code-most.html) : When we implement Runnable interface, same object is shared amongst multiple threads, but when we extend Thread class each and every thread gets associated with new object.

Inheritance (Implementing Runnable is lightweight operation) : When we extend Thread unnecessary all Thread class features are inherited, but when we implement Runnable interface no extra feature are inherited, as Runnable only consists only of one abstract method i.e. run() method.  So, implementing Runnable is lightweight operation.

Coding to interface : Even java recommends coding to interface. So, we must implement Runnable rather than extending thread. Also, Thread class implements Runnable interface.

Don’t extend unless you wanna modify fundamental behaviour of class, Runnable interface has only one abstract method i.e. run()  : We must [extend Thread](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) only when you are looking to modify run() and other methods as well. If you are simply looking to modify only the run() method [implementing Runnable](http://www.javamadesoeasy.com/2015/03/implementing-threads-in-java-by.html) is the best option (Runnable interface has only one abstract method i.e. run() ). We must not extend Thread class unless we're looking to modify fundamental behaviour of Thread class.

Flexibility in code when we implement Runnable : When we extend Thread first a fall all thread features are inherited and our class becomes direct subclass of Thread , so whatever action we are doing is in Thread class. But, when we implement Runnable we create a new thread and pass runnable object as parameter, we could pass runnable object to executorService & much more. So, we have more options when we implement Runnable and our code becomes more flexible.

ExecutorService : If we implement Runnable, we can start multiple thread created on runnable object  with ExecutorService (because we can start Runnable object with new threads), but not in the case when we extend Thread (because thread can be started only once).

**Question 7 . When threads are not lightweight process in java?**

Answer. Threads are [lightweight process](http://www.javamadesoeasy.com/2015/03/when-threads-are-not-lightweight.html) only if threads of same process are executing concurrently. But if threads of different processes are executing concurrently then threads are [heavy weight process](http://www.javamadesoeasy.com/2015/03/when-threads-are-not-lightweight.html).

**Question 8. How can you ensure all threads that started from main must end in order in which they started and also main should end in last? (Important)**

We can use [join() method](http://www.javamadesoeasy.com/2015/03/join-method-ensure-all-threads-that.html) to ensure all threads that started from main must end in order in which they started and also main should end in last.In other words waits for this thread to die. Calling join() method internally calls join(0);

**Question 15. Why wait(), notify()  and notifyAll() are in Object class and not in Thread class? (Important)**

**Answer.**

Every Object has a monitor, acquiring that monitors allow thread to hold lock on object. But Thread class does not have any monitors.

wait(), notify() and notifyAll() are called on objects only > When wait() method is called on object by thread it waits for another thread on that object to release object monitor by calling [notify() or notifyAll()](http://www.javamadesoeasy.com/2015/03/difference-between-notify-and-notifyall.html) method on that object.

When notify() method is called on object by thread it notifies all the threads

which are waiting for that object monitor that object monitor is available now.

So, this shows that wait(), notify() and notifyAll() are called on objects only.

Wait(), notify() and notifyAll() method being in Object class allows all the threads created on that object to communicate with other.  [As multiple threads may exist on same object].

As multiple threads exists on same object. Only one thread can hold object monitor at a time. As a result thread can notify other threads of same object that lock is available now. But, thread having these methods does not make any sense because multiple threads exists on object its not other way around (i.e. multiple objects exists on thread).

Now let’s discuss one hypothetical scenario, what will happen if Thread class contains wait(), notify() and notifyAll() methods?

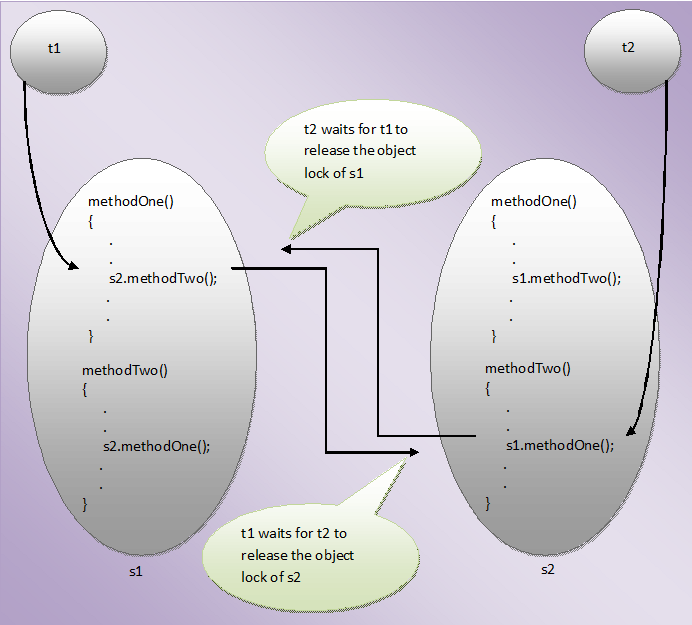
|  |
| --- |
| public static native void yield(); |

[synchronized block](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) : thread need not to to acquire object lock before calling yield() method i.e. yield() method can be called from outside synchronized block.

What Is Deadlock In Java?

Deadlock in java is a condition which occurs when two or more threads get blocked waiting for each other for an infinite period of time to release the resources(Locks) they hold. Deadlock is the common problem in multi threaded programming which can completely stops the execution of an application. So, extra care need to be taken while writing the multi threaded programs so that deadlock never occurs.

In the above multithreaded program, thread **t1** and **t2** are concurrent threads i.e they are executing their task simultaneously. There are two Shared class objects, **s1** and **s2**, which are shared by both the threads. Shared class has two synchronized methods, **methodOne()** and **methodTwo()**. That means, only one thread can execute these methods at a given time.



First, thread **t1** enters the **methodOne()** of **s1** object by acquiring the object lock of **s1**. At the same time, thread **t2** also enters the **methodTwo()** of **s2** object by acquiring the object lock of **s2**. **methodOne()** of **s1** object, currently executing by thread **t1**, calls **methodTwo()**of **s2** object from it’s body. So, thead **t1** tries to acquire the object lock of **s2** object. But object lock of **s2** object is already acquired by thread **t2**. So, thread **t1** waits for thread **t2** to release the object lock of **s2** object.

At the same time, thread **t2** is also executing **methodTwo()** of **s2** object. **methodTwo()** of **s2** object also makes a call to **methodOne()** of **s1**object. So, thread **t2** tries to acquire the object lock of **s1** object. But, it is already acquired by thread **t1**. So, thread **t2** also waits for thread **t1**to release the object lock of **s1** object.

Thus, both the threads wait for each other to release the object locks they own. They wait for infinite period of time to get the object locks owned by opposite threads. This condition of threads waiting forever is called Deadlock

Deadlock is a dangerous condition, if it happens , it will bring the whole application to complete halt. So, extra care need to be taken to avoid the deadlock. Followings are some tips that can be used to avoid the deadlock in java.

Try to avoid nested synchronized blocks. Nested synchronized blocks makes a thread to acquire another lock while it is already holding one lock. This may create the deadlock if another thread wants the same lock which is currently held by this thread.

[?](http://javaconceptoftheday.com/avoid-the-deadlock-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | synchronized (Lock A)  {      //Some statements        synchronized (Lock B)      {          //Try to avoid this block      }  } |

**Lock Ordering :**

If you needed nested synchronized blocks at any cost, then make sure that threads acquire the needed locks in some predefined order. For example, If there are three threads t1, t2 and t3 running concurrently and they needed locks A, B and C in the following manner,[?](http://javaconceptoftheday.com/avoid-the-deadlock-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Thread t1 :          Lock A          Lock B  Thread t2 :          Lock A          Lock C  Thread t3 :          Lock A          Lock B          Lock C |

In the above scenario, t1 needs A and B locks, t2 needs A and C locks and t3 needs A, B and C locks. If you define an order to acquire the locks like, Lock A must be acquired before Lock B and Lock B must be acquired before Lock c, then deadlock never occurs in the above case.

If you define such lock ordering, then thread t2 never acquire lock C and t3 never acquire lock B and lock C until they got lock A. They will wait for lock A until it is released by t1. After lock A is released by t1, any one of these threads will acquire lock A on the priority basis and finishes their task. Other thread which is waiting for lock A, will never try to acquire remaining locks.

By defining such lock ordering, you can avoid the deadlock.

**Lock Timeout :**

Another deadlock preventive tip is to specify the time for a thread to acquire the lock. If it fails to acquire the specified lock in the given time, then it should give up trying for a lock and retry after some time. Such method of specifying time to acquire the lock is called lock timeout.

Lock the code where it is actually needed. For example,If you want only some part of the method to be thread safety, then lock only that part not the whole method.

[?](http://javaconceptoftheday.com/avoid-the-deadlock-java/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | void method()  {      //Some statements        synchronized (this)      {          //Locking only some part of the method      }        //Some statements  } |

The Executor framework is based on the Executor interface, which describes an *executor* as any object capable of executing java.lang.Runnable tasks. This interface declares the following solitary method for executing a Runnable task:

void execute(Runnable command)

You submit a Runnable task by passing it to execute(Runnable). If the executor cannot execute the task for any reason (for instance, if the executor has been shut down), this method will throw a RejectedExecutionException.

The key concept is that *task submission is decoupled from the task-execution policy*, which is described by an Executor implementation. The *runnable* task is thus able to execute via a new thread, a pooled thread, the calling thread, and so on.

Note that Executor is very limited. For example, you can't shut down an executor or determine whether an asynchronous task has finished. You also can't cancel a running task. For these and other reasons, the Executor framework provides an [ExecutorService](http://www.javaworld.com/javaworld/jw-10-2011/111004-jtip-recursion-in%20-java-7.html" \t "_blank) interface, which extends Executor.

Five of ExecutorService's methods are especially noteworthy:

**boolean awaitTermination(long timeout, TimeUnit unit)** blocks the calling thread until all tasks have completed execution after a shutdown request, the timeout occurs, or the current thread is interrupted, whichever happens first. The maximum time to wait is specified by timeout, and this value is expressed in the unit units specified by the TimeUnit enum; for example, TimeUnit.SECONDS. This method throws java.lang.InterruptedExceptionwhen the current thread is interrupted. It returns *true* when the executor is terminated and *false* when the timeout elapses before termination.

**boolean isShutdown()** returns *true* when the executor has been shut down.

**void shutdown()** initiates an orderly shutdown in which previously submitted tasks are executed but no new tasks are accepted.

**<T> Future<T> submit(Callable<T> task)** submits a value-returning task for execution and returns a Future representing the pending results of the task.

**Future<?> submit(Runnable task)** submits a Runnable task for execution and returns a Future representing that task.

The Future<V> interface represents the result of an asynchronous computation. The result is known as a *future* because it typically will not be available until some moment in the future. You can invoke methods to cancel a task, return a task's result (waiting indefinitely or for a timeout to elapse when the task hasn't finished), and determine if a task has been cancelled or has finished.

The Callable<V> interface is similar to the Runnable interface in that it provides a single method describing a task to execute. Unlike Runnable's void run()method, Callable<V>'s V call() throws Exception method can return a value and throw an exception.

Executor factory methods

At some point, you'll want to obtain an executor. The Executor framework supplies the Executors utility class for this purpose. Executors offers several factory methods for obtaining different kinds of executors that offer specific thread-execution policies. Here are three examples:

**ExecutorService newCachedThreadPool()** creates a thread pool that creates new threads as needed, but which reuses previously constructed threads when they're available. Threads that haven't been used for 60 seconds are terminated and removed from the cache. This thread pool typically improves the performance of programs that execute many short-lived asynchronous tasks.

**ExecutorService newSingleThreadExecutor()** creates an executor that uses a single worker thread operating off an unbounded queue -- tasks are added to the queue and execute sequentially (no more than one task is active at any one time). If this thread terminates through failure during execution before shutdown of the executor, a new thread will be created to take its place when subsequent tasks need to be executed.

**ExecutorService newFixedThreadPool(int nThreads)** creates a thread pool that re-uses a fixed number of threads operating off a shared unbounded queue. At most nThreads threads are actively processing tasks. If additional tasks are submitted when all threads are active, they wait in the queue until a thread is available. If any thread terminates through failure during execution before shutdown, a new thread will be created to take its place when subsequent tasks need to be executed. The pool's threads exist until the executor is shut down.

The Executor framework offers additional types (such as the ScheduledExecutorService interface), but the types you are likely to work with most often are ExecutorService, Future, Callable, and Executors.

ExecutorService Implementations

Since ExecutorService is an interface, you need to its implementations in order to make any use of it. The ExecutorService has the following implementation in the java.util.concurrent package:

[**ThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/threadpoolexecutor.html)

[**ScheduledThreadPoolExecutor**](http://tutorials.jenkov.com/java-util-concurrent/scheduledexecutorservice.html)

Creating an ExecutorService

How you create an ExecutorService depends on the implementation you use. However, you can use the Executors factory class to create ExecutorService instances too. Here are a few examples of creating an ExecutorService:

ExecutorService executorService1 = Executors.newSingleThreadExecutor();

ExecutorService executorService2 = Executors.newFixedThreadPool(10);

ExecutorService executorService3 = Executors.newScheduledThreadPool(10);

ExecutorService Usage

There are a few different ways to delegate tasks for execution to an ExecutorService:

execute(Runnable)

submit(Runnable)

submit(Callable)

invokeAny(...)

invokeAll(...)

I will take a look at each of these methods in the following sections.

execute(Runnable)

The execute(Runnable) method takes a java.lang.Runnable object, and executes it asynchronously. Here is an example of executing a Runnable with an ExecutorService:

ExecutorService executorService = Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

executorService.shutdown();

There is no way of obtaining the result of the executed Runnable, if necessary. You will have to use a Callable for that (explained in the following sections).

submit(Runnable)

The submit(Runnable) method also takes a Runnable implementation, but returns a Future object. This Future object can be used to check if the Runnable as finished executing.Here is a ExecutorService submit() example:

Future future = executorService.submit(new Runnable() {

public void run() {

System.out.println("Asynchronous task");

}

});

future.get(); //returns null if the task has finished correctly.

submit(Callable)

The submit(Callable) method is similar to the submit(Runnable) method except for the type of parameter it takes. The Callable instance is very similar to a Runnable except that its call() method can return a result. The Runnable.run() method cannot return a result.

The Callable's result can be obtained via the Future object returned by the submit(Callable) method. Here is an ExecutorService Callable example:

Future future = executorService.submit(new Callable(){

public Object call() throws Exception {

System.out.println("Asynchronous Callable");

return "Callable Result";

}

});

System.out.println("future.get() = " + future.get());

The above code example will output this:

Asynchronous Callable

future.get() = Callable Result

invokeAny()

The invokeAny() method takes a collection of Callable objects, or subinterfaces of Callable. Invoking this method does not return a Future, but returns the result of one of the Callable objects. You have no guarantee about which of the Callable's results you get. Just one of the ones that finish.

If one of the tasks complete (or throws an exception), the rest of the Callable's are cancelled.Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

String result = executorService.invokeAny(callables);

System.out.println("result = " + result);

executorService.shutdown();

This code example will print out the object returned by one of the Callable's in the given collection. I have tried running it a few times, and the result changes. Sometimes it is "Task 1", sometimes "Task 2" etc.

invokeAll()

The invokeAll() method invokes all of the Callable objects you pass to it in the collection passed as parameter. The invokeAll() returns a list of Future objects via which you can obtain the results of the executions of each Callable.

Keep in mind that a task might finish due to an exception, so it may not have "succeeded". There is no way on a Future to tell the difference.

Here is a code example:

ExecutorService executorService = Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 1";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 2";

}

});

callables.add(new Callable<String>() {

public String call() throws Exception {

return "Task 3";

}

});

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

System.out.println("future.get = " + future.get());

}

executorService.shutdown();

ExecutorService Shutdown

When you are done using the ExecutorService you should shut it down, so the threads do not keep running.

For instance, if your application is started via a main() method and your main thread exits your application, the application will keep running if you have an active ExexutorService in your application. The active threads inside this ExecutorService prevents the JVM from shutting down.

To terminate the threads inside the ExecutorService you call its shutdown() method. The ExecutorServicewill not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down. All tasks submitted to the ExecutorService before shutdown() is called, are executed.

If you want to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks. There are no guarantees given about the executing tasks. Perhaps they stop, perhaps the execute until the end. It is a best effort attempt.

Thread Pools

Managing thread life cycle is expensive so one of the reason behind this framework is to abstract it. In complicated or real life applications, allocating and deallocating memory to multiple threads become even more complicated (compared to single thread). So to handle this, Java introduced concept of thread pools.

*Thread pool consists of worker threads. Tasks are submitted to a thread pool via a queue. Worker threads have a simple life : request new task from a work queue, execute it, and go back to pool for assignment to next task.Executors* class provides factory method to create thread pools. Let's cover them

***Executors.newFixedThreadPool(10)****:*This is one of the most common type of thread pool. This type of pool always has a specified number of threads running. If a thread from the pool is terminated due to some reason then it automatically gets replaced with a new thread.

**Executors.newCachedThreadPool()**: Creates an expandable thread pool which reuses previously created threads if available. These pools will improve the performance of the program that creates many short-lived asynchronous tasks. Threads that have been used for 60 seconds are terminated and removed from the cache. Thus if pool remains idle for longer time will not consume any resource.  
  
**Executors.newSingleThreadExecutor()**: Creates a single-threaded executor. It's a single worker thread to process tasks, and replaces it if it dies due to some reason/error. So using this, tasks will be executed sequentially. So if you have a queue having 10 tasks, then tasks will get executed one after another depending on the order.  
  
**Executors.newScheduledThreadPool()** : Creates a fixed size thread pool that supports delayed and periodic task execution.

Thread concurrency interview Question 9.

What is CountDownLatch in java?

Answer. There might be situation where we might like our thread to wait until one or more threads completes certain operation in java.

A [CountDownLatch](http://www.javamadesoeasy.com/2015/03/countdownlatch-in-java.html) is initialized with a given count .

count specifies the number of events that must occur before latch is released.

Every time a event happens count is reduced by 1. Once count reaches 0 latch is released.

CountDownLatch’s  constructor >

CountDownLatch(int count)

CountDownLatch is initialized with given count.

count specifies the number of events that must occur befor latch is released.

CountDownLatch’s await() method has 2 forms :

void await( ) throws InterruptedException

Causes the current thread to wait until  one of the following things happens-

latch count has down to reached 0, or

unless the thread is interrupted.

boolean await(long timeout, TimeUnit unit)

Causes the current thread to wait until  one of the following things happens-

latch count has down to reached 0,

unless the thread is interrupted, or

specified timeout elapses.

CountDownLatch’s countDown() method in java :

void **countDown**( )

Reduces latch **count** by 1.

If **count** reaches 0, all waiting threads are released.

Differences between synchronized and volatile keyword in Java? (Important)

[Volatile](http://www.javamadesoeasy.com/2015/03/volatile-keyword-in-java-difference.html) does not acquire any lock on variable or object, but [Synchronization](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html) acquires lock on method or block in which it is used.

Volatile variables are not cached, but variables used inside synchronized method or block are cached.

When volatile is used will never create deadlock in program, as volatile never obtains any kind of lock . But in case if synchronization is not done properly, we might end up creating dedlock in program.

What is race condition in multithreading and how can we solve it? (Important)

This is very important question, this forms the core of multi threading, you should be able to explain about [race condition in detail](http://www.javamadesoeasy.com/2015/03/race-condition-in-multithreading-and.html). When more than one thread try to access same resource without synchronization causes race condition.

So we can solve race condition by using either [synchronized block or synchronized method](http://www.javamadesoeasy.com/2015/03/synchronization-blocks-and-methods.html). When no two threads can access same resource at a time phenomenon is also called as mutual exclusion.

**Threads can communicate** with each other by using [**wait(), notify() and notifyAll()**](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) methods.

What is busy spin?

When one thread loops continuously waiting for another thread to signal.

Performance point of view - Busy spin is very bad from performance point of view, because one thread keeps on looping continuously ( and consumes CPU) waiting for another thread to signal.

Solution to busy spin -

We must use [sleep()](http://www.javamadesoeasy.com/2015/03/sleep-method-in-threads-10-key-features.html) or [wait() and notify()](http://www.javamadesoeasy.com/2015/03/wait-and-notify-methods-definition-8.html) method. Using wait() is better option.

Program - Consumer Producer problem with busy spin >

Consumer thread continuously execute (busy spin) in while loop till productionInProcess is true. Once producer thread has ended it will make boolean variable productionInProcess false and busy spin will be over.

|  |
| --- |
| while(productionInProcess){    System.out.println("BUSY SPIN - Consumer waiting for production to get over");  } |

**Overview**

[ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) is a framework provided by the JDK which simplifies the execution of tasks in asynchronous mode. Generally speaking, ExecutorService automatically provides a pool of threads and API for assigning tasks to it.

**2. Instantiating**ExecutorService

**2.1. Factory Methods of the**Executors**Class**

The easiest way to create ExecutorService is to use one of the factory methods of the Executorsclass.

For example, the following line of code will create a thread-pool with 10 threads:

|  |  |
| --- | --- |
| 1 | ExecutorService executor = Executors.newFixedThreadPool(10); |

The are several other factory methods to create predefined ExecutorService that meet specific use cases. To find the best method for your needs, consult [Oracle’s official documentation](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html).

**2.2. Directly Create an**ExecutorService

Because ExecutorService is an interface, an instance of any its implementations can be used. There are several implementations to choose from in the [java.util.concurrent](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html) package or you can create your own.

For example, the ThreadPoolExecutor class has a few constructors which can be used to configure an executor service and its internal pool.

|  |  |
| --- | --- |
| 1  2  3 | ExecutorService executorService =    new ThreadPoolExecutor(1, 1, 0L, TimeUnit.MILLISECONDS,    new LinkedBlockingQueue<Runnable>()); |

You may notice that the code above is very similar to the [source code](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/Executors.java#Executors.newSingleThreadExecutor%28%29) of the factory method newSingleThreadExecutor(). For most cases, a detailed manual configuration isn’t necessary.

**3. Assigning Tasks to the**ExecutorService

ExecutorService can execute Runnable and Callable tasks. To keep things simple in this article, two primitive tasks will be used. Notice that lambda expressions are used here instead of anonymous inner classes:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Runnable runnableTask = () -> {      try {          TimeUnit.MILLISECONDS.sleep(300);      } catch (InterruptedException e) {          e.printStackTrace();      }  };    Callable<String> callableTask = () -> {      TimeUnit.MILLISECONDS.sleep(300);      return "Task's execution";  };    List<Callable<String>> callableTasks = new ArrayList<>();  callableTasks.add(callableTask);  callableTasks.add(callableTask);  callableTasks.add(callableTask); |

Tasks can be assigned to the ExecutorService using several methods, including execute(), which is inherited from the Executor interface, and also submit(), invokeAny(), invokeAll().

The execute() method is void, and it doesn’t give any possibility to get the result of task’s execution or to check the task’s status (is it running or executed).

|  |  |
| --- | --- |
| 1 | executorService.execute(runnableTask); |

submit() submits a Callable or a Runnable task to an ExecutorService and returns a result of type Future.

|  |  |
| --- | --- |
| 1  2 | Future<String> future =    executorService.submit(callableTask); |

invokeAny() assigns a collection of tasks to an ExecutorService, causing each to be executed, and returns the result of a successful execution of one task (if there was a successful execution).

|  |  |
| --- | --- |
| 1 | String result = executorService.invokeAny(callableTasks); |

**invokeAll()** assigns a collection of tasks to an ExecutorService, causing each to be executed, and returns the result of all task executions in the form of a list of objects of type Future.

|  |  |
| --- | --- |
| 1 | List<Future<String>> futures = executorService.invokeAll(callableTasks); |

Now, before going any further, two more things must be discussed: shutting down an ExecutorService and dealing with Future return types.

**4. Shutting Down an**ExecutorService

In general, the ExecutorService will not be automatically destroyed when there is not task to process. It will stay alive and wait for new work to do.

In some cases this is very helpful; for example, if an app needs to process tasks which appear on an irregular basis or the quantity of these tasks is not known at compile time.

On the other hand, an app could reach its end, but it will not be stopped because a waiting ExecutorService will cause the JVM to keep running.

To properly shut down an ExecutorService, we have the shutdown() and shutdownNow() APIs.

The **shutdown()** method doesn’t cause an immediate destruction of the ExecutorService. It will make the ExecutorService stop accepting new tasks and shut down after all running threads finish their current work.

|  |  |
| --- | --- |
| 1 | executorService.shutdown(); |

The shutdownNow() method tries to destroy the ExecutorService immediately, but it doesn’t guarantee that all the running threads will be stopped at the same time. This method returns a list of tasks which are waiting to be processed. It is up to the developer to decide what to do with these tasks.

|  |  |
| --- | --- |
| 1 | List<Runnable> notExecutedTasks = executorService.shutDownNow(); |

One good way to shut down the ExecutorService (which is also [recommended by Oracle](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html)) is to use both of these methods combined with the awaitTermination() method. With this approach, the ExecutorService will first stop taking new tasks, the wait up to a specified period of time for all tasks to be completed. If that time expires, the execution is stopped immediately:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | executorService.shutdown();  try {      if (!executorService.awaitTermination(800, TimeUnit.MILLISECONDS)) {          executorService.shutdownNow();      }  } catch (InterruptedException e) {      executorService.shutdownNow();  } |

**5. The**Future**Interface**

The submit() and invokeAll() methods return an object or a collection of objects of type Future, which allows us to get the result of a task’s execution or to check the task’s status (is it running or executed).

The Future interface provides a special blocking method get() which returns an actual result of the Callable task’s execution or null in the case of Runnable task. Calling the get() method while the task is still running will cause execution to block until the task is properly executed and the result is available.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Future<String> future = executorService.submit(callableTask);  String result = null;  try {      result = future.get();  } catch (InterruptedException | ExecutionException e) {      e.printStackTrace();  } |

With very long blocking caused by the get() method, an application’s performance can degrade. If the resulting data is not crucial, it is possible to avoid such a problem by using timeouts:

|  |  |
| --- | --- |
| 1 | String result = future.get(200, TimeUnit.MILLISECONDS); |

If the execution period is longer than specified (in this case 200 milliseconds), a TimeoutException will be thrown.

The isDone() method can be used to check if the assigned task is already processed or not.

The Future interface also provides for the cancellation of task execution with the cancel() method, and to check the cancellation with isCancelled() method:

|  |  |
| --- | --- |
| 1  2 | boolean canceled = future.cancel(true);  boolean isCancelled = future.isCancelled(); |

**6. The**ScheduledExecutorService**Interface**

The ScheduledExecutorService runs tasks after some predefined delay and/or periodically. Once again, the best way to instantiate a ScheduledExecutorService is to use the factory methods of the Executors class.

For this section, a ScheduledExecutorService with one thread will be used:

|  |  |
| --- | --- |
| 1  2 | ScheduledExecutorService executorService = Executors    .newSingleThreadScheduledExecutor(); |

To schedule a single task’s execution after a fixed delay, us the scheduled() method of the ScheduledExecutorService. There are two scheduled() methods that allow you to execute Runnable or Callable tasks:

|  |  |
| --- | --- |
| 1  2 | Future<String> resultFuture =    executorService.schedule(callableTask, 1, TimeUnit.SECONDS); |

The scheduleAtFixedRate() method lets execute a task periodically after a fixed delay. The code above delays for one second before executing callableTask.

The following block of code will execute a task after an initial delay of 100 milliseconds, and after that, it will execute the same task every 450 milliseconds. If the processor needs more time to execute an assigned task than the period parameter of the scheduleAtFixedRate()method, the ScheduledExecutorService will wait until the current task is completed before starting the next:

|  |  |
| --- | --- |
| 1  2 | Future<String> resultFuture =    service.scheduleAtFixedRate(callableTask, 100, 450, TimeUnit.MILLISECONDS); |

If it is necessary to have a fixed length delay between iterations of the task, scheduleWithFixedDelay() should be used. For example, the following code will guarantee a 150-millisecond pause between the end of the current execution and the start of another one.

|  |  |
| --- | --- |
| 1 | service.scheduleWithFixedDelay(task, 100, 150, TimeUnit.MILLISECONDS); |

According to the scheduleAtFixedRate() and scheduleWithFixedDelay() method contracts, period execution of the task will end at the termination of the ExecutorService or if an exception is thrown during task execution.

**7.**ExecutorService**vs. Fork/Join**

After the release of Java 7, many developers decided that the ExecutorService framework should be replaced by the fork/join framework. This is not always the right decision, however. Despite the simplicity of usage and the frequent performance gains associated with fork/join, there is also a reduction in the amount of developer control over concurrent execution.

ExecutorService gives the developer the ability to control the number of generated threads and the granularity of tasks which should be executed by separate threads. The best use case for ExecutorService is the processing of independent tasks, such as transactions or requests according to the scheme “one thread for one task.”

In contrast, [according to Oracle’s documentation](https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html), fork/join was designed to speed up work which can be broken into smaller pieces recursively.

**8. Conclusion**

Even despite the relative simplicity of ExecutorService, there are a few common pitfalls. Let’s summarize them:

**Keeping an unused**ExecutorService**alive:** There is a detailed explanation in section 4 of this article about how to shut down an ExecutorService;

**Wrong thread-pool capacity while using fixed length thread-pool:**It is very important to determine how many threads the application will need to execute tasks efficiently. A thread-pool that is too large will cause unnecessary overhead just to create threads which mostly will be in the waiting mode. Too few can make an application seem unresponsive because of long waiting periods for tasks in the queue;

**Calling a**Future**‘s**get()**method after task cancellation:**An attempt to get the result of an already canceled task will trigger a CancellationException.

**Unexpectedly-long blocking with**Future**‘s**get()**method:** Timeouts sh

**Using Blocking Queue to implement Producer Consumer Pattern**

*BlockingQueue* amazingly simplifies implementation of Producer-Consumer design pattern by providing outofbox support of blocking on put() and take(). Developer doesn't need to write confusing and critical piece of wait-notify code to implement communication. **BlockingQuue** is an interface and Java 5 provides different implantation like ArrayBlockingQueue and LinkedBlockingQueue , both implement FIFO order or elements, while ArrayLinkedQueue isbounded in nature LinkedBlockingQueue is optionally bounded. here is a complete **code example of Producer Consumer pattern** with BlockingQueue. Compare it with classic [wait notify](http://javarevisited.blogspot.com/2012/02/why-wait-notify-and-notifyall-is.html) code, its much simpler and easy to understand.

public class **ProducerConsumerPattern** {

    public static void main(String args[]){

**//Creating shared object**

     BlockingQueue sharedQueue = new LinkedBlockingQueue();

**//Creating Producer and Consumer Thread**

     Thread prodThread = new Thread(new Producer(sharedQueue));

     Thread consThread = new Thread(new Consumer(sharedQueue));

**//Starting producer and Consumer thread**

     prodThread.start();

     consThread.start();

    } }

//Producer Class in java

class Producer implements Runnable {

    private final BlockingQueue sharedQueue;

    public Producer(BlockingQueue sharedQueue) {

        this.sharedQueue = sharedQueue;

    }

    @Override

    public void run() {

        for(int i=0; i<10; i++){

            try {

                System.out.println("Produced: " + i);

                sharedQueue.put(i);

            } catch (InterruptedException ex) {

                Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);

            }        }    } }

**//Consumer Class in Java**

class Consumer implements Runnable{

    private final BlockingQueue sharedQueue;

    public Consumer (BlockingQueue sharedQueue) {

        this.sharedQueue = sharedQueue;

    }

     @Override

    public void run() {

        while(true){

            try {

                System.out.println("Consumed: "+ sharedQueue.take());

            } catch (InterruptedException ex) {

                Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);

            }        }    } }  
**Reentrant Lock in Java**

On class level,  ReentrantLock is a concrete implementation of Lock [interface](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) provided in Java concurrency package from Java 1.5 onwards.  As per Javadoc, ReentrantLock is mutual exclusive lock, similar to implicit locking provided by [synchronized keyword in Java](http://javarevisited.blogspot.sg/2011/04/synchronization-in-java-synchronized.html), with extended feature like fairness, which can be used to provide lock to longest waiting thread. Lock is acquired by lock() method and held by [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html)until a call to unlock() method. Fairness  parameter is provided while creating instance of ReentrantLock in constructor. ReentrantLock provides same visibility and ordering guarantee, provided by implicitly locking, which means, unlock() happens before another thread get lock().

Difference between ReentrantLock and synchronized keyword in Java

Though ReentrantLock provides same visibility and orderings guaranteed as implicit lock, acquired by synchronized keyword in Java, it provides more functionality and differ in certain aspect. As stated earlier,  **main difference between synchronized and ReentrantLock** is ability to trying for lock interruptibly, and with timeout. [Thread](http://javarevisited.blogspot.com/2012/01/difference-thread-vs-runnable-interface.html)doesn’t need to block infinitely, which was the case with synchronized. Let’s see few more differences between synchronized and Lock in Java.

1) Another significant difference between ReentrantLock and synchronized keyword is **fairness**. synchronized keyword doesn't support fairness. Any thread can acquire lock once released, no preference can be specified, on the other hand you can make ReentrantLock fair by specifying fairness property, while creating instance of ReentrantLock. Fairness property provides lock to longest waiting thread, in case of contention.

2) Second difference between synchronized and Reentrant lock is **tryLock()** method. ReentrantLock provides convenient tryLock() method, which acquires lock only if its available or not held by any other thread. This reduce [blocking](http://javarevisited.blogspot.com/2012/02/what-is-blocking-methods-in-java-and.html)of thread waiting for lock in Java application.

3) One more worth noting difference between ReentrantLock and synchronized keyword in Java is, **ability to interrupt**Thread while waiting for Lock. In case of [synchronized](http://javarevisited.blogspot.com/2012/03/mixing-static-and-non-static.html) keyword, a thread can be blocked waiting for lock, for an indefinite period of time and there was no way to control that. ReentrantLock provides a method called lockInterruptibly(), which can be used to interrupt thread when it is [waiting for lock](http://javarevisited.blogspot.com/2011/05/wait-notify-and-notifyall-in-java.html). Similarly tryLock() with timeout can be used to timeout if lock is not available in certain time period.

4) ReentrantLock also provides convenient method to get List of all threads waiting for lock.

So, you can see, lot of significant differences between synchronized keyword and ReentrantLock in Java. In short, Lock interface adds lot of power and flexibility and allows some control over lock acquisition process, which can be leveraged to write highly scalable systems in Java.  
Read more: <http://javarevisited.blogspot.com/2013/03/reentrantlock-example-in-java-synchronized-difference-vs-lock.html#ixzz4lSKXKi6n>

# Design Patterns

## [Singleton Pattern](http://www.learn4master.com/tag/singleton-pattern)

To stop cloning of Object we will implement the Cloneable interface and throw CloneNotSupportedException. Also Serialized interface will be implemented and readObject will be used to return only one object at all time.

class JBT implements Cloneable, Serializable{

@Override

protected Object clone() throws CloneNotSupportedException {

return new CloneNotSupportedException();

}

protected Object readResolve() {

return createInstance();

}

/\*

\* As private constructor is used so can not create object of this class

\* directly. Except by using static method of same class.

\*/

private JBT() {

}

/\*

\* Here static inner class is used instead of Static variable. It means

\* Object will be lazy initialized.

\*/

private static class LazyInit {

private static final JBT instance = new JBT();

}

/\*

\* Whenever object JBT is required this method will be invoked and it will

\* return the instance of JBT.

\*/

static JBT createInstance() {

return LazyInit.instance;

}

int i;

}

Five ways to implement Singleton pattern in Java

Tags: [Design pattern](http://www.learn4master.com/tag/design-pattern), [Enum](http://www.learn4master.com/tag/enum), [Java](http://www.learn4master.com/tag/java), [Singleton Pattern](http://www.learn4master.com/tag/singleton-pattern)

In this post, I will describe five ways to implement the Singleton pattern in Java. They are Synchronization the getInstane() method,static final variable, using double checking lock with volatile keyword, using SingletonHolder, and Enum.

1. Classic Java Singleton pattern

The following code is the simplest implementation of Singleton Pattern in Java. However it is not multiple thread-safe.

Java Singleton Pattern 1

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton(); }

return instance;

}}

The singleton pattern implemented in the above example is easy to understand. We maintain a static reference to the singleton instance and returns that reference from the static getInstance() method. We also make the Constructor private, such that the client can only get the object of the Singleton class by calling the static getInstance() method.

The above example also uses a technique known as lazy instantiation to create the singleton object. So the singleton instance is not created until the getInstance() method is called for the first time.

The advantage of the lazy instantiation is that it ensures the singleton instances are created only when needed. This is important for the system performance if it is very expensive to create the singleton instance.

Why the above code is not thread-safe

The code in example 1 is not thread-safe because of the following code:

if(instance == null) {

instance = new Singleton();}

If a thread is preempted at Line 2 before the assignment is made, the instancemember variable will still be null, and another thread can subsequently enter the ifblock. In that case, two distinct singleton instances will be created. Unfortunately, that scenario rarely occurs and is therefore difficult to produce during testing.

2 A thread-safe singleton pattern in java using Synchronization

We can synchronize thegetInstance() method to make the above code thread-safe:

thread save singleton pattern in Java

public synchronized static Singleton getInstance() {

if(singleton == null) {

singleton = new Singleton();

} return singleton;}

The code in example 2 is thread-safe now, as only one thread can enter to the getInstance() method.

However, this method is not efficient, as we actually only need the method be synchronized the first time when it is called . Because synchronization is very expensive, **synchronized methods can run up to 100 times slower than unsynchronized methods**, we need to introduce a performance enhancement that only synchronize the singleton assignment in getInstance().

3.  Synchronize the critical code only

We might choose to rewrite thegetInstance() method by synchronizing the critical code only, like the following example:

public static Singleton getInstance() {

if(singleton == null) {

synchronized(Singleton.class) {

singleton = new Singleton();

} }

return singleton;

}

 However, this method is not thread-safe. Consider the following scenario: Thread 1 enters the synchronized block, and, before it can assign the singleton member variable, the thread is preempted. Then another thread can enter the if block. The second thread will wait for the first thread to finish, but we will still wind up with two distinct singleton instances. Is there a way to fix this problem?

4. Double-checked locking

Double-checked locking is a technique appears to make lazy instantiation thread-safe. That technique is illustrated in the following code:

public static Singleton getInstance() {

if(singleton == null) {

synchronized(Singleton.class) {

if(singleton == null) {

singleton = new Singleton();

} } }

return singleton;}

Let’s reconsider the case that two threads enter the if code at the same tie. Imagine Thread 1 enters the synchronized block and is preempted. Subsequently, a second thread enters the if block. When Thread 1 exits the synchronized block, Thread 2 makes a second check to see if the singleton instance is still null. Since Thread 1 set thesingleton member variable, Thread 2’s second check will fail, and a second singleton will not be created. Or so it seems.

Unfortunately, **double-checked locking** is not guaranteed to work because **the compiler is free to assign a value to the singleton member variable before the singleton’s constructor is called**. If that happens, **Thread 1 can be preempted after the singleton reference has been assigned, but before the singleton is initialized**, so **Thread 2 can return a reference to an uninitialized singleton instance**.

Since double-checked locking is not guaranteed to work, we have to develop alternative methods

5.Double-checked locking with volatile keyword

A thread safe Single pattern

public class Singleton {

private volatile static Singleton instance; //声明成 volatile

private Singleton (){}

public static Singleton getSingleton() {

if (instance == null) {

synchronized (Singleton.class) {

if (instance == null) {

instance = new Singleton(); }

} }

return instance; }}

 The volatile prevents memory writes from being re-ordered, making it impossible for other threads to read uninitialized fields of your singleton through the singleton’s pointer.

However, we should avoid use the volatile based method, as it is hard to understand and it is easy to make mistakes.

6. Thread-safe but not lay initialized

public class Singleton{

//the variable will be created when the class is loaded

private static final Singleton instance = new Singleton();

private Singleton(){}

public static Singleton getInstance(){

return instance;

}

}

This method is thread-safe, but it is not lazy initialized . The singleton object is created as soon as the class is loaded.

7. The ultimate Thread-safe and efficient singleton pattern in Java

Java singleton Pattern

/ Correct lazy initialization in Java

@ThreadSafe

class Singleton {

private static class SingletonHolder {

public static Singleton instance = new Singleton();

} public static Singleton getInstance() {

return Singleton

The method is recommend by effective java.  It is lazy initialized, and multiple-thread safe.

8. Using Enum

The best way to implement a thread safe Singleton Pattern in Java is using Enum. See the following example.

enum Color {

RED(1), GREEN(2), YELLOW(3);

private int nCode ;

private Color( int \_nCode) {

this.nCode = \_nCode; }

@Override

public String toString() {

return String.valueOf ( this . nCode );

} }

public class ColorTest {

public static void main(String[] args) {

Color red = Color.RED;

Color red2 = Color.RED;

System.out.println(red == red2); // return true

**Summary**

In this post, I described 8 ways to implement java Singleton pattern. Actually, only 5 ways are correct: Synchronization the method, static final variable, using double checking

//to prevent reflection API

private Singleton() {

// Check if we already have an instance

if (INSTANCE != null) {

throw new IllegalStateException("Singleton" +

" instance already created.");

}

System.out.println("Singleton Constructor Running..."); }

public class Student implements Serializable, Cloneable {

// Eager-Instantiation: only-time INSTANCE created

private volatile static Student INSTANCE = new Student();

// private constructor

private Student() {

// Check if we already have an instance

if (INSTANCE != null) {

throw new IllegalStateException("Singleton" +

" instance already created.");

}

}

// create static method to get same instance every time

public static Student getInstance(){

return INSTANCE;

}

// readResolve method to suppress creating new object during de-serialization

private Object readResolve() throws ObjectStreamException {

return INSTANCE;

}

@Override

protected Object clone() throws CloneNotSupportedException {

// directly throw Clone Not Supported Exception

throw new CloneNotSupportedException();

}

// other utility methods and details of this class

}

## **factory pattern:**

Factory method is a [creational design pattern](http://www.geeksforgeeks.org/design-patterns-set-1-introduction/), i.e., related to object creation. In Factory pattern, we create object without exposing the creation logic to client and the client use the same common interface to create new type of object.  
The idea is to use a static member-function (static factory method) which creates & returns instances, hiding the details of class modules from user.

A factory pattern is one of the core design principles to create an object, allowing clients to create objects of a library(explained below) in a way such that it doesn’t have tight coupling with the class hierarchy of the library.

***What is meant when we talk about library and clients?***  
A library is something which is provided by some third party which exposes some public APIs and clients make calls to those public APIs to complete its task. A very simple example can be different kinds of Views provided by Android OS.

***Why factory pattern?***  
Let us understand it with an example:

|  |
| --- |
|  |

Output: I am two wheeler

***What is the problems with above design?***  
As you must have observed in the above example, Client creates objects of either TwoWheeler or FourWheeler based on some input during constructing its object.  
Say, library introduces a new class ThreeWheeler to incorporate three wheeler vehicles also. What would happen? Client will end up chaining a new else if in the conditional ladder to create objects of ThreeWheeler. Which in turn will need Client to be recompiled. So, each time a new change is made at the library side, Client would need to make some corresponding changes at its end and recompile the code. Sounds bad? This is a very bad practice of design.

**How to avoid the problem?**  
The answer is, create a static (or factory) method. Let us see below code.

|  |
| --- |
|  |

Run on IDE

Output:

I am three wheeler

In the above example, we have totally decoupled the selection of type for object creation from Client. The library is now responsible to decide which object type to create based on an input. Client just needs to make call to library’s factory Create method and pass the type it wants without worrying about the actual implementation of creation of objects.

Thanks to [Rumplestiltskin](http://qa.geeksforgeeks.org/user/Rumplestiltskin) for providing above explanation [here](http://qa.geeksforgeeks.org/559/what-is-factory-pattern-how-to-implement-it-in-c?show=559#q559).  
**Other examples of Factory Method:**

Say, in a ‘Drawing’ system, depending on user’s input, different pictures like square, rectangle, circle can be drawn. Here we can use factory method to create instances depending on user’s input. For adding new type of shape, no need to change client’s code.

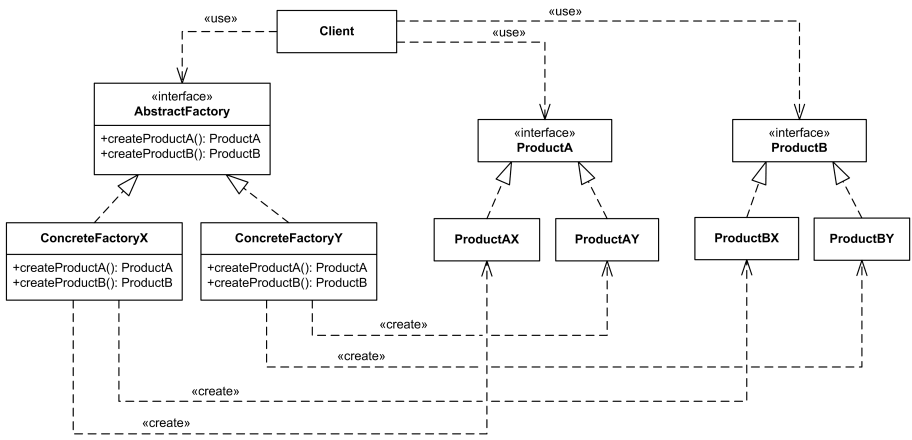
Another example: In travel site, we can book train ticket as well bus tickets and flight ticket. In this case user can give his travel type as ‘bus’, ‘train’ or ‘flight’.  
Here we have an abstract class ‘AnyTravel’ with a static member function ‘GetObject’ which depending on user’s travel type, will create & return object of ‘BusTravel’ or ‘ TrainTravel’. ‘BusTravel’ or ‘ TrainTravel’ have common functions like passenger name, Origin, destinationparameters.

Also see [this](http://qa.geeksforgeeks.org/423/factory-design-pattern?show=423#q423)interview question.

## Abstract Factory Pattern

Abstract Factory design pattern is one of the Creational pattern. Abstract Factory pattern is almost similar to [Factory Pattern](http://www.geeksforgeeks.org/design-patterns-set-2-factory-method/) is considered as another layer of abstraction over factory pattern. Abstract Factory patterns work around a super-factory which creates other factories.

Abstract factory pattern implementation provides us a framework that allows us to create objects that follow a general pattern. So at runtime, abstract factory is coupled with any desired concrete factory which can create objects of desired type.

[](http://contribute.geeksforgeeks.org/wp-content/uploads/class-example-abstract-factory.png)  
UML class diagram example for the Abstract Factory Design Pattern.

**AbstractFactory** : Declares an interface for operations that create abstract product objects.

**ConcreteFactory** : Implements the operations declared in the AbstractFactory to create concrete product objects.

**Product** : Defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.

**Client** : Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Abstract Factory provides interfaces for creating families of related or dependent objects without specifying their concrete classes.

Client software creates a concrete implementation of the abstract factory and then uses the generic interfaces to create the concrete objects that are part of the family of objects.  
The client does not know or care which concrete objects it gets from each of these concrete factories since it uses only the generic interfaces of their products.

So with this idea of Abstract Factory pattern, we will now try to create a design that will facilitate the creation of related objects.

**Implementation**

Let’s take an example, Suppose we want to build a global car factory. If it was [factory design pattern](http://www.geeksforgeeks.org/design-patterns-set-2-factory-method/), then it was suitable for a single location. But for this pattern, we need multiple locations and some critical design changes.

We need car factories in each location like IndiaCarFactory, USACarFactory and DefaultCarFactory. Now, our application should be smart enough to identify the location where it is being used, so we should be able to use appropriate car factory without even knowing which car factory implementation will be used internally. This also saves us from someone calling wrong factory for a particular location.

Here we need another layer of abstraction which will identify the location and internally use correct car factory implementation without even giving a single hint to user. This is exactly the problem, which abstract factory pattern is used to solve.

|  |
| --- |
| // Java Program to demonstrate the  // working of Abstract Factory Pattern   enum CarType  {    MICRO, MINI, LUXURY  }    abstract class Car  {      Car(CarType model, Location location)      {          this.model = model;          this.location = location;    }        abstract void construct();      CarType model = null;      Location location = null;      CarType getModel()      {        return model;    }      void setModel(CarType model)      {        this.model = model;    }      Location getLocation()      {        return location;    }      void setLocation(Location location)      {        this.location = location;    }        @Override      public String toString()      {        return "CarModel - "+model + " located in "+location;   }}  class LuxuryCar extends Car  {      LuxuryCar(Location location)    {          super(CarType.LUXURY, location);          construct();    }      protected void construct()    {          System.out.println("Connecting to luxury car");    }}   class MicroCar extends Car  {    MicroCar(Location location)      {        super(CarType.MICRO, location);          construct();    }      protected void construct()    {        System.out.println("Connecting to Micro Car ");    }}   class MiniCar extends Car  {    MiniCar(Location location)      {        super(CarType.MINI,location );          construct();    }      void construct()      {        System.out.println("Connecting to Mini car");    }}    enum Location  {  DEFAULT, USA, INDIA}   class INDIACarFactory  {    static Car buildCar(CarType model)      {        Car car = null;          switch (model)          {            case MICRO:                  car = new MicroCar(Location.INDIA);                  break;                case MINI:                  car = new MiniCar(Location.INDIA);                  break;                               case LUXURY:                car = new LuxuryCar(Location.INDIA);                break;                                   default:                break;                     }          return car;    }}    class DefaultCarFactory  {    public static Car buildCar(CarType model)      {        Car car = null;          switch (model)          {            case MICRO:                  car = new MicroCar(Location.DEFAULT);                break;                           case MINI:                car = new MiniCar(Location.DEFAULT);                break;                               case LUXURY:                car = new LuxuryCar(Location.DEFAULT);                break;                                   default:                break;          }        return car;   }}    class USACarFactory  {    public static Car buildCar(CarType model)      {          Car car = null;          switch (model)          {              case MICRO:                car = new MicroCar(Location.USA);                break;              case MINI:                car = new MiniCar(Location.USA);                break;                               case LUXURY:                car = new LuxuryCar(Location.USA);                break;                                   default:                break;                       }        return car;    }}   class CarFactory  {    private CarFactory()      {             }      public static Car buildCar(CarType type)      {        Car car = null;          // We can add any GPS Function here which          // read location property somewhere from configuration          // and use location specific car factory          // Currently I'm just using INDIA as Location          Location location = Location.INDIA;             switch(location)          {            case USA:                  car = USACarFactory.buildCar(type);                  break;                     case INDIA:                car = INDIACarFactory.buildCar(type);                break;                default:                car = DefaultCarFactory.buildCar(type);            }          return car;     }}    class AbstractDesign  {    public static void main(String[] args)      {        System.out.println(CarFactory.buildCar(CarType.MICRO));          System.out.println(CarFactory.buildCar(CarType.MINI));          System.out.println(CarFactory.buildCar(CarType.LUXURY));      } |

Output :

Connecting to Micro Car

CarModel - MICRO located in INDIA

Connecting to Mini car

CarModel - MINI located in INDIA

Connecting to luxury car

CarModel - LUXURY located in INDIA

**Difference**

The main difference between a “factory method” and an “abstract factory” is that the factory method is a single method, and an abstract factory is an object.

The factory method is just a method, it can be overridden in a subclass, whereas the abstract factory is an object that has multiple factory methods on it.

The Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

**Advantages**

This pattern is particularly useful when the client doesn’t know exactly what type to create.

**Isolation of concrete classes:** The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.

**Exchanging Product Families easily:** The class of a concrete factory appears only once in an application, that is where it’s instantiated. This makes it easy to change the concrete factory an application uses. It can use various product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once.

**Promoting consistency among products:** When product objects in a family are designed to work together, it’s important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.n.

**Disadvantages**

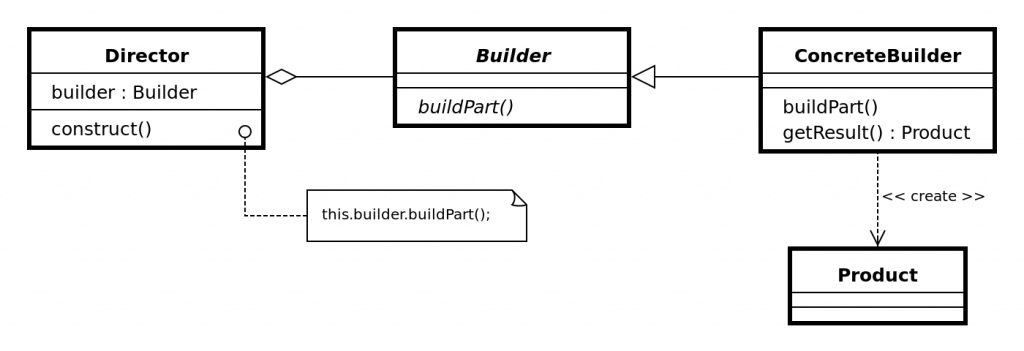
**Difficult to support new kind of products:** Extending abstract factories to produce new kinds of Products isn’t easy. That’s because the AbstractFactory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses.

**NOTE :**  
Somewhat the above example is also based on How the Cabs like uber and ola functions on the large scale.

## Builder Design Pattern:

Builder pattern aims to “Separate the construction of a complex object from its representation so that the same construction process can create different representations.” It is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.

**UML Diagram of Builder Design Pattern**

[](http://contribute.geeksforgeeks.org/wp-content/uploads/builder.png)

**Product –**The product class defines the type of the complex object that is to be generated by the builder pattern.

**Builder –**This abstract base class defines all of the steps that must be taken in order to correctly create a product. Each step is generally abstract as the actual functionality of the builder is carried out in the concrete subclasses. The GetProduct method is used to return the final product. The builder class is often replaced with a simple interface.

**ConcreteBuilder –**There may be any number of concrete builder classes inheriting from Builder. These classes contain the functionality to create a particular complex product.

**Director –**The director class controls the algorithm that generates the final product object. A director object is instantiated and its Construct method is called. The method includes a parameter to capture the specific concrete builder object that is to be used to generate the product. The director then calls methods of the concrete builder in the correct order to generate the product object. On completion of the process, the GetProduct method of the builder object can be used to return the product.

**Lets see an Example of Builder Design Pattern :**

Consider a construction of a home. Home is the final end product (object) that is to be returned as the output of the construction process. It will have many steps like basement construction, wall construction and so on roof construction. Finally the whole home object is returned. Here using the same process you can build houses with different properties.

|  |
| --- |
| interface HousePlan  {      public void setBasement(String basement);      public void setStructure(String structure);      public void setRoof(String roof);      public void setInterior(String interior);}   class House implements HousePlan  {   private String basement;      private String structure;      private String roof;      private String interior;     public void setBasement(String basement)      {        this.basement = basement;    }      public void setStructure(String structure)      {        this.structure = structure;    }       public void setRoof(String roof)     {        this.roof = roof;    }      public void setInterior(String interior)      {        this.interior = interior;    } }  interface HouseBuilder  {     public void buildBasement();      public void buildStructure();      public void bulidRoof();      public void buildInterior();      public House getHouse();}   class IglooHouseBuilder implements HouseBuilder  {    private House house;      public IglooHouseBuilder()     {          this.house = new House();    }      public void buildBasement()      {        house.setBasement("Ice Bars");    }      public void buildStructure()      {        house.setStructure("Ice Blocks");    }       public void buildInterior()      {        house.setInterior("Ice Carvings");    }       public void bulidRoof() {       house.setRoof("Ice Dome");    }       public House getHouse()      {        return this.house;    }}   class TipiHouseBuilder implements HouseBuilder  {    private House house;      public TipiHouseBuilder()      {        this.house = new House();    }       public void buildBasement()      {        house.setBasement("Wooden Poles");    }       public void buildStructure()      {        house.setStructure("Wood and Ice");    }       public void buildInterior()      {        house.setInterior("Fire Wood");    }       public void bulidRoof()      {        house.setRoof("Wood, caribou and seal skins");    }       public House getHouse()      {        return this.house;    } }   class CivilEngineer  {    private HouseBuilder houseBuilder;      public CivilEngineer(HouseBuilder houseBuilder)      {        this.houseBuilder = houseBuilder;    }       public House getHouse()      {        return this.houseBuilder.getHouse();    }       public void constructHouse()      {  this.houseBuilder.buildBasement();          this.houseBuilder.buildStructure();          this.houseBuilder.bulidRoof();          this.houseBuilder.buildInterior();    }}    class Builder  {    public static void main(String[] args)      {        HouseBuilder iglooBuilder = new IglooHouseBuilder();          CivilEngineer engineer = new CivilEngineer(iglooBuilder);           engineer.constructHouse();           House house = engineer.getHouse();           System.out.println("Builder constructed: "+ house);    }} |

Builder constructed: House@6d06d69c

**Advantages of Builder Design Pattern**

* The parameters to the constructor are reduced and are provided in highly readable method calls.
* Builder design pattern also helps in minimizing the number of parameters in constructor and thus there is no need to pass in null for optional parameters to the constructor.
* Object is always instantiated in a complete state
* Immutable objects can be build without much complex logic in object building process.

**Disadvantages of Builder Design Pattern**

* The number of lines of code increase at least to double in builder pattern, but the effort pays off in terms of design flexibility and much more readable code.
* Requires creating a separate ConcreteBuilder for each different type of Product.

## Prototype Design Pattern

Prototype allows us to hide the complexity of making new instances from the client. The concept is to copy an existing object rather than creating a new instance from scratch, something that may include costly operations. The existing object acts as a prototype and contains the state of the object. The newly copied object may change same properties only if required. This approach saves costly resources and time, especially when the object creation is a heavy process.

The prototype pattern is a creational design pattern. Prototype patterns is required, when object creation is time consuming, and costly operation, so we create object with existing object itself. One of the best available way to create object from existing objects are **clone() method**. Clone is the simplest approach to implement prototype pattern. However, it is your call to decide how to copy existing object based on your business model.

**Prototype Design Participants**

1) **Prototype** : This is the prototype of actual object.

2) **Prototype registry** : This is used as registry service to have all prototypes accessible using simple string parameters.

3) **Client** : Client will be responsible for using registry service to access prototype instances.

**When to use the Prototype Design Pattern**

When a system should be independent of how its products are created, composed, and represented and  
When the classes to instantiate are specified at run-time.  
For example,  
1) By dynamic loading or To avoid building a class hierarchy of factories that parallels the class hierarchy of products or

2) When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

|  |
| --- |
| // A Java program to demonstrate working of  // Prototype Design Pattern with example  // of a ColorStore class to store existing objects.  abstract class Color implements Cloneable{      protected String colorName;      abstract void addColor();      public Object clone()      {        Object clone = null;          try        {            clone = super.clone();        }          catch (CloneNotSupportedException e)          {            e.printStackTrace();        }          return clone;    }}  class blueColor extends Color  {    public blueColor()      {        this.colorName = "blue";    }      void addColor()     {        System.out.println("Blue color added");    }     }  class blackColor extends Color{      public blackColor()    {        this.colorName = "black";    }      @Override      void addColor()     {        System.out.println("Black color added");    }}  class ColorStore {      private static Map<String, Color> colorMap = new HashMap<String, Color>();      static      {        colorMap.put("blue", new blueColor());          colorMap.put("black", new blackColor());    }           public static Color getColor(String colorName)      {        return (Color) colorMap.get(colorName).clone();    }}  // Driver class  class Prototype  {    public static void main (String[] args)      {          ColorStore.getColor("blue").addColor();          ColorStore.getColor("black").addColor();          ColorStore.getColor("black").addColor();          ColorStore.getColor("blue").addColor();      }} |

Output :

Blue color added

Black color added

Black color added

Blue color added

**Advantages of Prototype Design Pattern**

* **Adding and removing products at run-time –** Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That’s a bit more flexible than other creational patterns, because a client can install and remove prototypes at run-time.
* **Specifying new objects by varying values –**Highly dynamic systems let you define new behavior through object composition by specifying values for an object’s variables and not by defining new classes.
* **Specifying new objects by varying structure –**Many applications build objects from parts and subparts. For convenience, such applications often let you instantiate complex, user-defined structures to use a specific subcircuit again and again.
* **Reduced subclassing –**Factory Method often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object. Hence you don’t need a Creator class hierarchy at all.

**Disadvantages of Prototype Design Pattern**

* Overkill for a project that uses very few objects and/or does not have an underlying emphasis on the extension of prototype chains.
* It also hides concrete product classes from the client
* Each subclass of Prototype must implement the clone() operation which may be difficult, when the classes under consideration already exist. Also implementing clone() can be difficult when their internals include objects that don’t support copying or have circular references.

## Facade Design Pattern

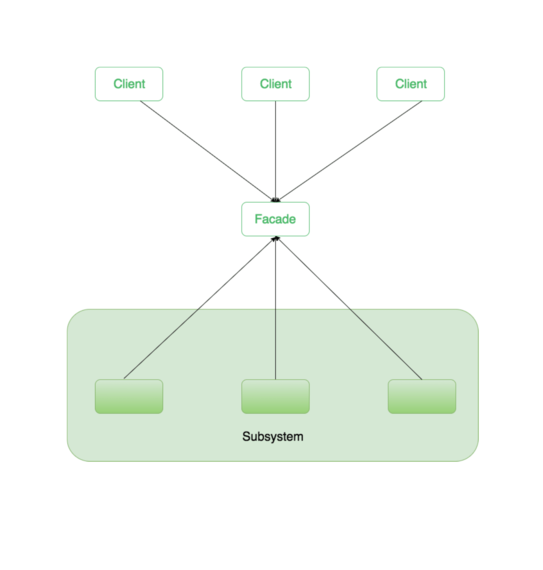
Facade is a part of Gang of Four design pattern and it is categorized under Structural design patterns. Before we dig into the details of it, let us discuss some examples which will be solved by this particular Pattern.

So, As the name suggests, it means the face of the building. The people walking past the road can only see this glass face of the building. They do not know anything about it, the wiring, the pipes and other complexities. It hides all the complexities of the building and displays a friendly face.

**More examples**

In Java, the interface JDBC can be called a facade because, we as users or clients create connection using the “java.sql.Connection” interface, the implementation of which we are not concerned about. The implementation is left to the vendor of driver.

Another good example can be the startup of a computer. When a computer starts up, it involves the work of cpu, memory, hard drive, etc. To make it easy to use for users, we can add a facade which wrap the complexity of the task, and provide one simple interface instead.  
Same goes for the **Facade Design Pattern**. It hides the complexities of the system and provides an interface to the client from where the client can access the system.

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Untitled-document-Google-Docs-1.png)

**Facade Design Pattern Diagram**

Now Let’s try and understand the facade pattern better using a simple example. Let’s consider a hotel. This hotel has a hotel keeper. There are a lot of restaurants inside hotel e.g. Veg restaurants, Non-Veg restaurants and Veg/Non Both restaurants.  
You, as client want access to different menus of different restaurants . You do not know what are the different menus they have. You just have access to hotel keeper who knows his hotel well. Whichever menu you want, you tell the hotel keeper and he takes it out of from the respective restaurants and hands it over to you. Here, the hotel keeper acts as the **facade**, as he hides the complexities of the system hotel.  
Let’s see how it works :

**Interface of Hotel**

|  |
| --- |
| package structural.facade;  public interface Hotel{      public Menus getMenus();} |

The hotel interface only returns Menus.  
Similarly, the Restaurant are of three types and can implement the hotel interface. Let’s have a look at the code for one of the Restaurants.

**NonVegRestaurant.java**

|  |
| --- |
| package structural.facade;   public class NonVegRestaurant implements Hotel  {    public Menus getMenus()      {        NonVegMenu nv = new NonVegMenu();          return nv;    }} |

**VegRestaurant.java**

|  |
| --- |
| package structural.facade;  public class VegRestaurant implements Hotel  {    public Menus getMenus()      {        VegMenu v = new VegMenu();          return v;    }} |

**VegNonBothRestaurant.java**

|  |
| --- |
| package structural.facade;  public class VegNonBothRestaurant implements Hotel  {    public Menus getMenus()      {        Both b = new Both();          return b;    }} |

Now let’s consider the facade,

**HotelKeeper.java**

|  |
| --- |
| package structural.facade;  public class HotelKeeper  {    public VegMenu getVegMenu()      {        VegRestaurant v = new VegRestaurant();          VegMenu vegMenu = (VegMenu)v.getMenus();          return vegMenu;    }      public NonVegMenu getNonVegMenu()      {        NonVegRestaurant v = new NonVegRestaurant();          NonVegMenu NonvegMenu = (NonVegMenu)v.getMenus();          return NonvegMenu;    }           public Both getVegNonMenu()      {        VegNonBothRestaurant v = new VegNonBothRestaurant();          Both bothMenu = (Both)v.getMenus();          return bothMenu;      }    } |

From this, It is clear that the complex implementation will be done by HotelKeeper himself. The client will just access the HotelKeeper and ask for either Veg, NonVeg or VegNon Both Restaurant menu.

**How will the client program access this façade?**

|  |
| --- |
| package structural.facade;  public class Client  {    public static void main (String[] args)      {        HotelKeeper keeper = new HotelKeeper();          VegMenu v = keeper.getVegMenu();          NonVegMenu nv = keeper.getNonVegMenu();          Both = keeper.getVegNonMenu();    }} |

In this way the implementation is sent to the façade. The client is given just one interface and can access only that. This hides all the complexities.

**When Should this pattern be used?**

The facade pattern is appropriate when you have a **complex system**that you want to expose to clients in a simplified way, or you want to make an external communication layer over an existing system which is incompatible with the system. Facade deals with interfaces, not implementation. Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

This article is contributed by [Saket Kumar](https://www.linkedin.com/in/saketkumar95/). If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](http://www.contribute.geeksforgeeks.org/) or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Decorator Design Pattern

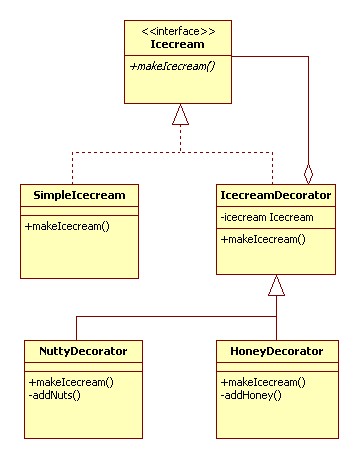
To extend or modify the behaviour of ‘an instance’ at runtime decorator [design pattern](http://javapapers.com/design-patterns/introduction-to-design-patterns/) is used. Inheritance is used to extend the abilities of ‘a class’. Unlike inheritance, you can choose any single object of a class and [modify its behaviour](http://javapapers.com/core-java/overloading-and-overriding/) leaving the other instances unmodified.

In implementing the decorator pattern you construct a wrapper around an object by extending its behavior. The wrapper will do its job before or after and delegate the call to the wrapped instance.

Design of decorator pattern

You start with an [interface](http://javapapers.com/core-java/abstract-and-interface-core-java-2/java-interface/) which creates a blue print for the class which will have decorators. Then implement that interface with basic functionalities. Till now we have got an interface and an implementation concrete class. Create an [abstract class](http://javapapers.com/core-java/abstract-and-interface-core-java-2/difference-between-a-java-interface-and-a-java-abstract-class/) that contains ([aggregation relationship](http://javapapers.com/oops/association-aggregation-composition-abstraction-generalization-realization-dependency/)) an attribute type of the interface. The constructor of this class assigns the interface type instance to that attribute. This class is the decorator base class. Now you can extend this class and create as many concrete decorator classes. The concrete decorator class will add its own methods. After / before executing its own method the concrete decorator will call the base instance’s method. Key to this decorator design pattern is the binding of method and the base instance happens at runtime based on the object [passed as parameter](http://javapapers.com/core-java/java-pass-by-value-and-pass-by-reference/) to the constructor. Thus dynamically customizing the behavior of that specific instance alone.

Decorator Design Pattern – UML Diagram



Implementation of decorator pattern

Following given example is an implementation of decorator design pattern. Icecream is a classic example for decorator design pattern. You create a basic icecream and then add toppings to it as you prefer. The added toppings change the taste of the basic icecream. You can add as many topping as you want. This sample scenario is implemented below.

public interface Icecream {

public String makeIcecream();

}

The above is an interface depicting an icecream. I have kept things as simple as possible so that the focus will be on understanding the design pattern. Following class is a concrete implementation of this interface. This is the base class on which the decorators will be added.

package com.javapapers.sample.designpattern;

public class SimpleIcecream implements Icecream {

@Override

public String makeIcecream() {

return "Base Icecream"; }}

Following class is the decorator class. It is the core of the decorator design pattern. It contains an attribute for the type of interface. Instance is assigned dynamically at the creation of decorator using its constructor. Once assigned that instance method will be invoked.

abstract class IcecreamDecorator implements Icecream {

protected Icecream specialIcecream;

public IcecreamDecorator(Icecream specialIcecream) {

this.specialIcecream = specialIcecream; }

public String makeIcecream() {

return specialIcecream.makeIcecream(); }}

Following two classes are similar. These are two decorators, concrete class implementing the abstract decorator. When the decorator is created the base instance is passed using the constructor and is assigned to the super class. In the makeIcecream method we call the base method followed by its own method addNuts(). This addNuts() extends the behavior by adding its own steps.

public class NuttyDecorator extends IcecreamDecorator {

public NuttyDecorator(Icecream specialIcecream) {

super(specialIcecream); }

public String makeIcecream() {

return specialIcecream.makeIcecream() + addNuts(); }

private String addNuts() { return " + cruncy nuts"; }}

public class HoneyDecorator extends IcecreamDecorator {

public HoneyDecorator(Icecream specialIcecream) {

super(specialIcecream); }

public String makeIcecream() {

return specialIcecream.makeIcecream() + addHoney(); }

private String addHoney() {

return " + sweet honey"; }}

Execution of the decorator pattern

I have created a simple icecream and decorated that with nuts and on top of it with honey. We can use as many decorators in any order we want. This excellent flexibility and changing the behaviour of an instance of our choice at runtime is the main advantage of the decorator design pattern.

public class TestDecorator {

public static void main(String args[]) {

Icecream icecream = new HoneyDecorator(new NuttyDecorator(new SimpleIcecream()));

System.out.println(icecream.makeIcecream()); }}

Output

Base Icecream + cruncy nuts + sweet honey

Decorator Design Pattern in java API

java.io.BufferedReader;  
java.io.FileReader;  
java.io.Reader;

The above readers of java API are designed using decorator design pattern.

# JavaPrograms

Occurences of every character in the String

String str="hhh rr rrr ttt" ;

**char**[] charArray=str.toCharArray();

Map<Character,Integer> resultMap= **new** HashMap<>();

**for**(**char** ch:charArray){

**if**(resultMap.containsKey(ch)){

resultMap.put(ch, resultMap.get(ch)+1);

}**else**{

resultMap.put(ch, 1);

}

}

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

//MostFrequent Element

**public** **static** **int** mostFrequentElement(**int** []arr){

Map<Integer,Integer> resultMap= **new** HashMap<>();

**for**(**int** ele:arr){

**if**(resultMap.containsKey(ele)){

resultMap.put(ele, resultMap.get(ele)+1);

}**else**{

resultMap.put(ele, 1);

}

}

**int** element=0,frequency=1;

Set<Entry<Integer,Integer>> elements=resultMap.entrySet();

**for**(Entry<Integer,Integer> entry:elements){

**if**(entry.getValue()>frequency){

element=entry.getKey();

frequency=entry.getValue();

} }

System.***out***.println("Most Frequent element"+element);

System.***out***.println(" Frequency element"+frequency);

**return** element;}

CountCharacterOccurence without looping

  String s = "Java is java again java again";

        char c = 'a';

        int count = s.length() - s.replace("a", "").length();

        System.out.println("Number of occurances of 'a' in "+s+" = "+count);

/\* \* Using LinkedHashMap to find first non repeated character of String

\* Algorithm : \* Step 1: get character array and loop through it to build a \* hash table with char and their count. \* Step 2: loop through LinkedHashMap to find an entry with \* value 1, that's your first non-repeated character, \* as LinkedHashMap maintains insertion order. \*/

 public static char getFirstNonRepeatedChar(String str) {

Map<Character,Integer> counts = new LinkedHashMap<>(str.length());

for (char c : str.toCharArray()) {

counts.put(c, counts.containsKey(c) ? counts.get(c) + 1 : 1); }

for (Entry<Character,Integer> entry : counts.entrySet()) {

 if (entry.getValue() == 1) {

return entry.getKey();

} } throw new RuntimeException("didn't find any non repeated Character"); }

/\* \* Finds first non repeated character in a String in just one pass. \* It uses two storage to cut down one iteration, standard space vs time \* trade-off.Since we store repeated and non-repeated character separately, \* at the end of iteration, first element from List is our first non \* repeated character from String. \*/

public static char firstNonRepeatingChar(String word) {

Set<Character> repeating = new HashSet<>();

List<Character> nonRepeating = new ArrayList<>();

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

char letter = word.charAt(i);

if (repeating.contains(letter)) { continue; }

if (nonRepeating.contains(letter)) {

nonRepeating.remove((Character) letter);

repeating.add(letter); } else {

nonRepeating.add(letter); } }

return nonRepeating.get(0); }

 /\* \* Using HashMap to find first non-repeated character from String in Java. \* Algorithm : \* Step 1 : Scan String and store count of each character in HashMap \* Step 2 : traverse String and get count for each character from Map. \* Since we are going through String from first to last character, \* when count for any character is 1, we break, it's the first \* non repeated character. Here order is achieved by going \* through String again. \*/

public static char firstNonRepeatedCharacter(String word) {

HashMap<Character,Integer> scoreboard = new HashMap<>();

 // build table [char -> count]

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

char c = word.charAt(i);

if (scoreboard.containsKey(c)) {

scoreboard.put(c, scoreboard.get(c) + 1);

} else {

scoreboard.put(c, 1); } }

// since HashMap doesn't maintain order, going through string again

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

char c = word.charAt(i);

 if (scoreboard.get(c) == 1) { return c; } }

throw new RuntimeException("Undefined behaviour"); }

**static** **int** maxSubArraySum(**int** a[]) {

**int** size = a.length;

**int** max\_so\_far = Integer.***MIN\_VALUE***, max\_ending\_here = 0, start = 0, end = 0, s = 0;

**for** (**int** i = 0; i < size; i++) {

max\_ending\_here = max\_ending\_here + a[i];

**if** (max\_so\_far < max\_ending\_here) {

max\_so\_far = max\_ending\_here;

start = s;

end = i;

}

**if** (max\_ending\_here < 0) {

max\_ending\_here = 0;

s = i + 1;

}

}

**for** (**int** i = start; i <= end; i++)

System.***out***.print(a[i] + " ");

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

**return** max\_so\_far;

}

**public** **static** **void** maxSumSubArray() {

**int** maxSumSoFar = -2147483648;

**int** curSum = 0, maxSum;

**int** start = 0;

**int** end = 0, s = 0, i;

**int**[] array = { -2, 1, -3, 4, -1, 2, 1, -5, 4 };

// int []array={-6,2,-3,-4,-1,-5,-5};

**for** (i = 0; i < array.length; i++) {

curSum = curSum + array[i];//

**if** (curSum > maxSumSoFar) {

maxSumSoFar = curSum;

start = s;

end = i;

}

**if** (curSum < 0) {

curSum = 0;

s = i + 1;

}

}

**for** (i = start; i <= end; i++)

System.***out***.print(array[i] + " ");

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

}

**static** **int** getMissingNo() {

**int** i, total;

**int** a[]={1,2,4,5};

**int** n=a.length+1;

total = (n) \* (n + 1) / 2;

**for** (i = 0; i < a.length; i++)

total -= a[i];

**return** total;

}

// print sum of two numbers equal to given number

**public** **static** **void** printPairsUsingSet() {

// given sum=7

**int** numbers[] = { 2, 4, 3, 5, 6, -2, 4, 7, 8, 9 }, sum = 7;

**if** (numbers.length < 2) {

**return**;

}

Set set = **new** HashSet(numbers.length);

**for** (**int** value : numbers) {

**int** target = sum - value;

// if target number is not in set then add

**if** (!set.contains(target)) {

set.add(value);

} **else** {

System.***out***.printf("(%d, %d) %n", value, target);

}

}

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

}

**public** **static** **void** topTwo(**int**[] numbers) {

**int** max1 = Integer.***MIN\_VALUE***;

**int** max2 = Integer.***MIN\_VALUE***;

**for** (**int** number : numbers) {

**if** (number > max1) {

max2 = max1;

max1 = number;

} **else** **if** (number > max2) {

max2 = number;

}

}

}

**public** **static** **long** getBinary(**int** n){

**long** binary=0,rem,i=1;

**while**(n!=0){

/\*r=decimal%2;

sum=r+ sum\*10;

decimal=decimal/2;\*/

rem=n%2;

n/=2;

binary+=rem\*i;

i\*=10;

}

**return** binary;

}

**public** **static** **long** getDecimal(**long** l){

**long** sum=0,r,i=0;

**while**(l!=0){

r=l%10;

sum=(**long**) (sum+r\*Math.*pow*(2, i));

l=l/10;

i++;

}

**return** sum;

}

**void** selectionSort(**int**[] ar) {

**for** (**int** i = 0; i < ar.length - 1; i++) {

**int** min = i;

**for** (**int** j = i + 1; j < ar.length; j++)

**if** (ar[j] < ar[min])

min = j;

**int** temp = ar[i];

ar[i] = ar[min];

ar[min] = temp;

}

}

**public** **static** **void** bubbleSort(**int** ar[]) {

**for** (**int** i = (ar.length - 1); i >= 0; i--) {

**for** (**int** j = 1; j <= i; j++) {

**if** (ar[j - 1] > ar[j]) {

**int** temp = ar[j - 1];

ar[j - 1] = ar[j];

ar[j] = temp;

}

}

}

**for** (**int** a : ar)

System.***out***.print(" " + a);

}

//To sort unordered list of elements, we remove its entries one at a time and then insert each of

//them into a sorted part (initially empty):

**void** insertionSort(**int**[] ar)

{

**for** (**int** i=1; i < ar.length; i++)

{

**int** index = ar[i]; **int** j = i;

**while** (j > 0 && ar[j-1] > index)

{

ar[j] = ar[j-1];

j--;

}

ar[j] = index;

} }

What Is Anagram?

Two strings are called anagrams if they contain same set of characters but in different order. For example, **“Dormitory – Dirty Room”**, **“keep – peek”,  “School Master – The Classroom”** are some.

static void isAnagram(String s1, String s2)    {

        //Removing all white spaces from s1 and s2

        String copyOfs1 = s1.replaceAll("\\s", "");

        String copyOfs2 = s2.replaceAll("\\s", "");

        //Initially setting status as true

        boolean status = true;

        if(copyOfs1.length() != copyOfs2.length())        {

            //Setting status as false if copyOfs1 and copyOfs2 doesn't have same length

            status = false;

        } else {

      //Changing the case of characters of both copyOfs1 and copyOfs2 and converting them to char array

            char[] s1Array = copyOfs1.toLowerCase().toCharArray();

            char[] s2Array = copyOfs2.toLowerCase().toCharArray();

            //Sorting both s1Array and s2Array

            Arrays.sort(s1Array);

            Arrays.sort(s2Array);

            //Checking whether s1Array and s2Array are equal

            status = Arrays.equals(s1Array, s2Array);

        }

## singleLinked List

public class SinglyLinkedListImpl<T> {

    private Node<T> head;

    private Node<T> tail;

    public void add(T element){

        Node<T> nd = new Node<T>();

        nd.setValue(element);

        System.out.println("Adding: "+element);

        /\*\*

         \* check if the list is empty

         \*/

        if(head == null){

            //since there is only one element, both head and

            //tail points to the same object.

            head = nd;

            tail = nd;

        } else {

            //set current tail next link to new node

            tail.setNextRef(nd);

            //set tail as newly created node

            tail = nd;

        }    }

    public void addAfter(T element, T after){

        Node<T> tmp = head;

        Node<T> refNode = null;

        System.out.println("Traversing to all nodes..");

        /\*\*

         \* Traverse till given element

         \*/

        while(true){

            if(tmp == null){

                break;            }

            if(tmp.compareTo(after) == 0){

                //found the target node, add after this node

                refNode = tmp;

                break;

            }

            tmp = tmp.getNextRef();

        }

        if(refNode != null){

            //add element after the target node

            Node<T> nd = new Node<T>();

            nd.setValue(element);

            nd.setNextRef(tmp.getNextRef());

            if(tmp == tail){

                tail = nd;

            }

            tmp.setNextRef(nd);

        } else {

            System.out.println("Unable to find the given element...");

        }    }

    public void deleteFront(){

        if(head == null){

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

        Node<T> tmp = head;

        head = tmp.getNextRef();

        if(head == null){

            tail = null;

        }

        System.out.println("Deleted: "+tmp.getValue());

    }

      public void deleteAfter(T after){

        Node<T> tmp = head;

        Node<T> refNode = null;

        System.out.println("Traversing to all nodes..");

        /\*\*

         \* Traverse till given element

         \*/

        while(true){

            if(tmp == null){

                break;

            }

            if(tmp.compareTo(after) == 0){

                //found the target node, add after this node

                refNode = tmp;

                break;

            }

            tmp = tmp.getNextRef();

        }

        if(refNode != null){

            tmp = refNode.getNextRef();

            refNode.setNextRef(tmp.getNextRef());

            if(refNode.getNextRef() == null){

                tail = refNode;

            }

            System.out.println("Deleted: "+tmp.getValue());

        } else {

            System.out.println("Unable to find the given element...");

        }

    }

    public void traverse(){

        Node<T> tmp = head;

        while(true){

            if(tmp == null){

                break;

            }

            System.out.println(tmp.getValue());

            tmp = tmp.getNextRef();       }    }

    public static void main(String a[]){

        SinglyLinkedListImpl<Integer> sl = new SinglyLinkedListImpl<Integer>();

        sl.add(3);        sl.add(32);        sl.add(54);        sl.add(89);        sl.addAfter(76, 54);

        sl.deleteFront();        sl.deleteAfter(76);

        sl.traverse();        }}

class Node<T> implements Comparable<T> {

    private T value;

    private Node<T> nextRef;

//getters& settrs

    @Override

    public int compareTo(T arg) {

        if(arg == this.value){

            return 0;

        } else {

            return 1;        }   }}

## Stack

public class Stack<Item> implements Iterable<Item> {

private int n; *// size of the stack*

private Node first; *// top of stack*

*// helper linked list class*

private class Node {

private Item item;

private Node next; }

*/\*\**

*\* Initializes an empty stack.*

*\*/*

public **Stack**() { first = null;

n = 0; }

*/\*\**

*\* Returns true if this stack is empty. \**

*\** @return *true if this stack is empty; false otherwise*

*\*/*

public boolean **isEmpty**() {

return first == null; }

*/\*\**

*\* Returns the number of items in this stack. \**

*\** @return *the number of items in this stack*

*\*/*

public int **size**() { return n; }

*/\*\**

*\* Adds the item to this stack. \**

*\** @param *item the item to add*

*\*/*

public void **push**(Item item) {

Node oldfirst = first;

first = new **Node**();

first.item = item;

first.next = oldfirst;

n++; }

*/\*\**

*\* Removes and returns the item most recently added to this stack. \**

*\** @return *the item most recently added*

*\** @throws *NoSuchElementException if this stack is empty*

*\*/*

public Item **pop**() {

if (**isEmpty**()) throw new **NoSuchElementException**("Stack underflow");

Item item = first.item; *// save item to return*

first = first.next; *// delete first node*

n--;

return item; *// return the saved item*

} */\*\**

*\* Returns (but does not remove) the item most recently added to this stack. \**

*\** @return *the item most recently added to this stack*

*\** @throws *NoSuchElementException if this stack is empty*

*\*/*

public Item **peek**() {

if (**isEmpty**()) throw new **NoSuchElementException**("Stack underflow");

return first.item; }

*/\*\**

*\* Returns a string representation of this stack. \**

*\** @return *the sequence of items in this stack in LIFO order, separated by spaces*

*\*/*

public String **toString**() {

StringBuilder s = new **StringBuilder**();

for (Item item : this) {

s.**append**(item);

s.**append**(' '); }

return s.**toString**(); }

/\*\*

\* Returns an iterator to this stack that iterates through the items in LIFO order. \*

\* @return an iterator to this stack that iterates through the items in LIFO order

\*/

public Iterator<Item> iterator() { return new ListIterator(); }

// an iterator, doesn't implement remove() since it's optional

private class ListIterator implements Iterator<Item> {

private Node current = first;

public boolean hasNext() { return current != null; }

public void remove() { throw new UnsupportedOperationException(); }

public Item next() {

if (!hasNext()) throw new NoSuchElementException();

Item item = current.item;

current = current.next;

return item; } }

*/\*\**

*\* Unit tests the {*@code *Stack} data type.*

*\*/*

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

Stack<String> stack = new Stack<String>();

while (!StdIn.**isEmpty**()) {

String item = StdIn.**readString**();

if (!item.**equals**("-")) stack.**push**(item);

else if (!stack.**isEmpty**()) StdOut.**print**(stack.**pop**() + " ");

}

StdOut.**println**("(" + stack.**size**() + " left on stack)"); }}

## Queue

This is a Java Program to implement a Double Ended Queue. Queue is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes queue a First-In-First-Out (FIFO) data structure. However in a double ended queue addition and removal of entities can be performed at both ends. A double-ended queue (dequeue) is an abstract data type that generalizes a queue, for which elements can be added to or removed from either the front (head) or back (tail).

Here is the source code of the Java program to implement a Double Ended Queue. The Java program is successfully compiled and run on a Windows system. The program output is also shown below.

import java.util.\*;

/\* Class Node \*/

class Node

{

protected int data;

protected Node link;

/\* Constructor \*/

public Node()

{

link = null;

data = 0;

}

/\* Constructor \*/

public Node(int d,Node n)

{

data = d;

link = n;

}

//setters and getters for link&data

}}

/\* Class Dequeue \*/

class Dequeue

{ private Node front, rear;

private int size;

/\* Constructor \*/

public Dequeue()

{ front = null;

rear = null;

size = 0; }

/\* Function to check if queue is empty \*/

public boolean isEmpty()

{ return front == null; }

/\* Function to get the size of the queue \*/

public int getSize()

{ return size; }

/\* Clear dequeue \*/

public void clear()

{ front = null;

rear = null;

size = 0; }

/\* Function to insert an element at begining \*/

public void insertAtFront(int val)

{ Node nptr = new Node(val, null);

size++ ;

if (front == null) {

front = nptr;

rear = front; }

else {

nptr.setLink(front);

front = nptr; } }

/\* Function to insert an element at end \*/

public void insertAtRear(int val)

{ Node nptr = new Node(val,null);

size++ ;

if (rear == null) {

rear = nptr;

front = rear; }

else { rear.setLink(nptr);

rear = nptr; } }

/\* Function to remove front element from the queue \*/

public int removeAtFront()

{ if (isEmpty() )

throw new NoSuchElementException("Underflow Exception");

Node ptr = front;

front = ptr.getLink();

if (front == null)

rear = null; size-- ;

return ptr.getData(); }

/\* Function to remove rear element from the queue \*/

public int removeAtRear()

{ if (isEmpty() )

throw new NoSuchElementException("Underflow Exception");

int ele = rear.getData();

Node s = front;

Node t = front;

while (s != rear) {

t = s;

s = s.getLink(); }

rear = t;

rear.setLink(null); size --;

return ele; }

/\* Function to check the front element of the queue \*/

public int peekAtFront() {

if (isEmpty() )

throw new NoSuchElementException("Underflow Exception");

return front.getData(); }

/\* Function to check the front element of the queue \*/

public int peekAtRear() {

if (isEmpty() )

throw new NoSuchElementException("Underflow Exception");

return rear.getData(); }

/\* Function to display the status of the queue \*/

public void display() {

System.out.print("\nDequeue = ");

if (size == 0)

{ System.out.print("Empty\n");

return ; }

Node ptr = front;

while (ptr != rear.getLink() ) {

System.out.print(ptr.getData()+" ");

ptr = ptr.getLink(); }

System.out.println(); }}

/\* Class DoubleEndedQueueTest \*/

public class DoubleEndedQueueTest

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

Scanner scan = new Scanner(System.in);

/\* Creating object of class Dequeue \*/

Dequeue dq = new Dequeue();

/\* Perform Dequeue Operations \*/

System.out.println("Dequeue Test\n");

char ch;

do {

System.out.println("\nDequeue Operations");

System.out.println("1. insert at front");

System.out.println("2. insert at rear");

System.out.println("3. delete at front");

System.out.println("4. delete at rear");

System.out.println("5. peek at front");

System.out.println("6. peek at rear");

System.out.println("7. size");

System.out.println("8. check empty");

System.out.println("9. clear");

int choice = scan.nextInt();

switch (choice) {

case 1 : System.out.println("Enter integer element to insert");

dq.insertAtFront( scan.nextInt() ); break;

case 2 : System.out.println("Enter integer element to insert");

dq.insertAtRear( scan.nextInt() ); break;

case 3 : try { System.out.println("Removed Element = "+ dq.removeAtFront()); }

catch (Exception e) {

System.out.println("Error : " + e.getMessage()); } break;

case 4 : try {

System.out.println("Removed Element = "+ dq.removeAtRear()); }

catch (Exception e) { System.out.println("Error : " + e.getMessage()); } break;

case 5 : try {

System.out.println("Peek Element = "+ dq.peekAtFront()); }

catch (Exception e) {

System.out.println("Error : " + e.getMessage()); } break;

case 6 : try {

System.out.println("Peek Element = "+ dq.peekAtRear());

}

catch (Exception e) {

System.out.println("Error : " + e.getMessage()); } break;

case 7 : System.out.println("Size = "+ dq.getSize()); break;

case 8 : System.out.println("Empty status = "+ dq.isEmpty()); break;

case 9 : System.out.println("\nDequeue Cleared\n"); dq.clear(); break;

default : System.out.println("Wrong Entry \n "); break; }

/\* display dequeue \*/

dq.display();

System.out.println("\nDo you want to continue (Type y or n) \n");

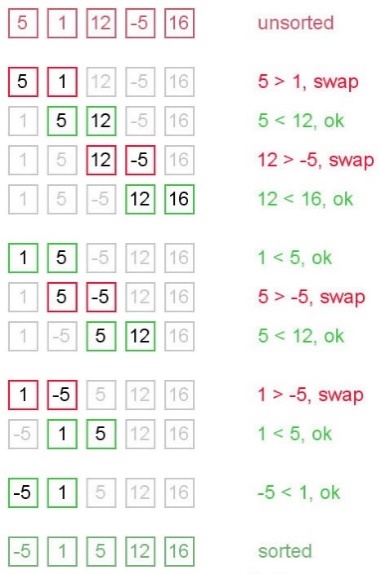
ch = scan.next().charAt(0);

} while (ch == 'Y'|| ch == 'y');

} }

## Bubble sort in java.

Bubble sort, also referred to as sinking sort, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, most of the other sorting algorithms are more efficient for large lists.



Bubble sort has worst-case and average complexity both О(n2), where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of O(n log n). Even other О(n2) sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.Performance of bubble sort over an already-sorted list (best-case) is O(n).

public class MyBubbleSort {

// logic to sort the elements

public static void bubble\_srt(int array[]) {

int n = array.length;

int k;

for (int m = n; m >= 0; m--) {

for (int i = 0; i < n - 1; i++) {

k = i + 1;

if (array[i] > array[k]) {

swapNumbers(i, k, array);

} }

printNumbers(array); } }

private static void swapNumbers(int i, int j, int[] array) {

int temp;

temp = array[i];

array[i] = array[j];

array[j] = temp;

}

private static void printNumbers(int[] input) {

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

System.out.print(input[i] + ", "); }

System.out.println("\n"); }

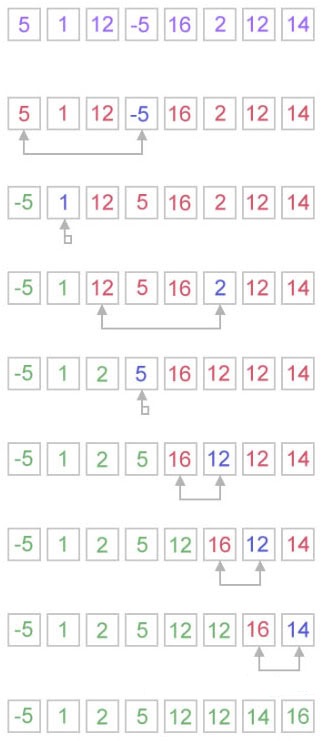
public static void main(String[] args) {

int[] input = { 4, 2, 9, 6, 23, 12, 34, 0, 1 };

bubble\_srt(input); }}

## selection sort in java.

The selection sort is a combination of searching and sorting. During each pass, the unsorted element with the smallest (or largest) value is moved to its proper position in the array. The number of times the sort passes through the array is one less than the number of items in the array. In the selection sort, the inner loop finds the next smallest (or largest) value and the outer loop places that value into its proper location.



Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takesn − 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining n − 1 elements and so on, for (n − 1) + (n − 2) + ... + 2 + 1 = n(n − 1) / 2 ∈ Θ(n2) comparisons. Each of these scans requires one swap for n − 1 elements.

public class MySelectionSort {

public static int[] doSelectionSort(int[] arr){

for (int i = 0; i < arr.length - 1; i++) {

int index = i;

for (int j = i + 1; j < arr.length; j++)

if (arr[j] < arr[index])

index = j;

int smallerNumber = arr[index];

arr[index] = arr[i];

arr[i] = smallerNumber; }

return arr; }

public static void main(String a[]){

int[] arr1 = {10,34,2,56,7,67,88,42};

int[] arr2 = doSelectionSort(arr1);

for(int i:arr2){

System.out.print(i);

System.out.print(", ");

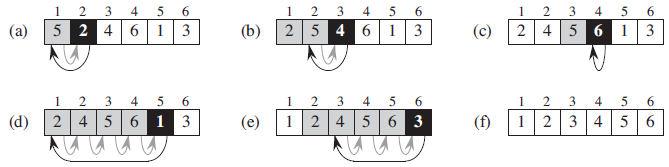
} }}

## insertion sort in java.

Insertion sort is a simple sorting algorithm, it builds the final sorted array one item at a time. It is much less efficient on large lists than other sort algorithms.

Advantages of Insertion Sort:   
1) It is very simple.  
2) It is very efficient for small data sets.  
3) It is stable; i.e., it does not change the relative order of elements with equal keys.  
4) In-place; i.e., only requires a constant amount O(1) of additional memory space.

Insertion sort iterates through the list by consuming one input element at each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

   
Image source: “Introduction to Algorithms”, The MIT Press

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., Θ(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array. The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n2)). The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

public class MyInsertionSort {

public static void main(String a[]){

int[] arr1 = {10,34,2,56,7,67,88,42};

int[] arr2 = doInsertionSort(arr1);

for(int i:arr2){

System.out.print(i);

System.out.print(", "); } }

public static int[] doInsertionSort(int[] input){

int temp;

for (int i = 1; i < input.length; i++) {

for(int j = i ; j > 0 ; j--){

if(input[j] < input[j-1]){

temp = input[j];

input[j] = input[j-1];

input[j-1] = temp;

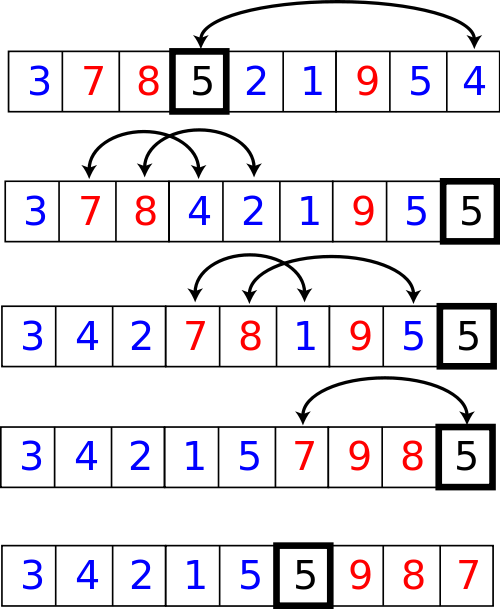
} } }

return input; }}

## quick sort in java.

Quicksort or partition-exchange sort, is a fast sorting algorithm, which is using divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

Steps to implement Quick sort:  
1) Choose an element, called pivot, from the list. Generally pivot can be the middle index element.  
2) Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.  
3) Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.



The complexity of quick sort in the average case is Θ(n log(n)) and in the worst case is Θ(n2).

public class MyQuickSort {

private int array[];

private int length;

public void sort(int[] inputArr) {

if (inputArr == null || inputArr.length == 0) {

return; }

this.array = inputArr;

length = inputArr.length;

quickSort(0, length - 1); }

private void quickSort(int lowerIndex, int higherIndex) {

int i = lowerIndex;

int j = higherIndex;

// calculate pivot number, I am taking pivot as middle index number

int pivot = array[lowerIndex+(higherIndex-lowerIndex)/2];

// Divide into two arrays

while (i <= j) {

/\*\*

\* In each iteration, we will identify a number from left side which

\* is greater then the pivot value, and also we will identify a number

\* from right side which is less then the pivot value. Once the search

\* is done, then we exchange both numbers.

\*/

while (array[i] < pivot) { i++; }

while (array[j] > pivot) { j--; }

if (i <= j) {

exchangeNumbers(i, j);

//move index to next position on both sides

i++;

j--;

} }

// call quickSort() method recursively

if (lowerIndex < j)

quickSort(lowerIndex, j);

if (i < higherIndex)

quickSort(i, higherIndex); }

private void exchangeNumbers(int i, int j) {

int temp = array[i];

array[i] = array[j];

array[j] = temp; }

public static void main(String a[]){

MyQuickSort sorter = new MyQuickSort();

int[] input = {24,2,45,20,56,75,2,56,99,53,12};

sorter.sort(input);

for(int i:input){

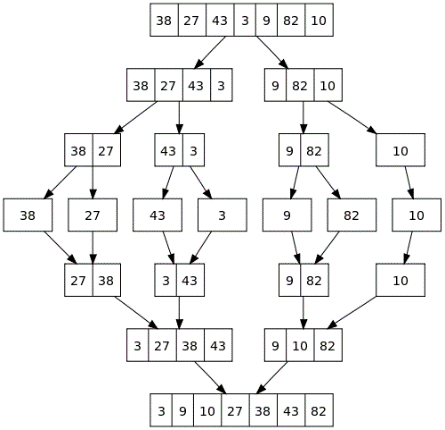
System.out.print(i);

System.out.print(" "); } }}

## merge sort in java.

Merge sort is a divide and conquer algorithm.

Steps to implement Merge Sort:  
1) Divide the unsorted array into n partitions, each partition contains 1 element. Here the one element is considered as sorted.  
2) Repeatedly merge partitioned units to produce new sublists until there is only 1 sublist remaining. This will be the sorted list at the end.



Merge sort is a fast, stable sorting routine with guaranteed O(n\*log(n)) efficiency. When sorting arrays, merge sort requires additional scratch space proportional to the size of the input array. Merge sort is relatively simple to code and offers performance typically only slightly below that of quicksort.

public class MyMergeSort {

private int[] array;

private int[] tempMergArr;

private int length;

public static void main(String a[]){

int[] inputArr = {45,23,11,89,77,98,4,28,65,43};

MyMergeSort mms = new MyMergeSort();

mms.sort(inputArr);

for(int i:inputArr){

System.out.print(i);

System.out.print(" "); } }

public void sort(int inputArr[]) {

this.array = inputArr;

this.length = inputArr.length;

this.tempMergArr = new int[length];

doMergeSort(0, length - 1); }

private void doMergeSort(int lowerIndex, int higherIndex) {

if (lowerIndex < higherIndex) {

int middle = lowerIndex + (higherIndex - lowerIndex) / 2;

// Below step sorts the left side of the array

doMergeSort(lowerIndex, middle);

// Below step sorts the right side of the array

doMergeSort(middle + 1, higherIndex);

// Now merge both sides

mergeParts(lowerIndex, middle, higherIndex); } }

private void mergeParts(int lowerIndex, int middle, int higherIndex) {

for (int i = lowerIndex; i <= higherIndex; i++) {

tempMergArr[i] = array[i];

}

int i = lowerIndex;

int j = middle + 1;

int k = lowerIndex;

while (i <= middle && j <= higherIndex) {

if (tempMergArr[i] <= tempMergArr[j]) {

array[k] = tempMergArr[i];

i++;

} else {

array[k] = tempMergArr[j];

j++; }

k++; }

while (i <= middle) {

array[k] = tempMergArr[i];

k++;

i++; } }

\*/ public static int performBinarySearch(int[] input, int number) {

int low = 0; int high = input.length - 1;

 while (high >= low) {

 int middle = (low + high) / 2; if (input[middle] == number) {

 return middle; }

else if (input[middle] < number) {

low = middle + 1;

} else if (input[middle] > number) {

high = middle - 1; }

}

return -1; }  
**import v/s static import**

The static import declaration is analogous to the normal import declaration. Where the normal import declaration imports classes from packages, allowing them to be used without package qualification, the static import declaration imports static members from classes, allowing them to be used without class qualification.

**Marker Interface** in java is an interface with no fields or methods within it. It is used to convey to the JVM that the class implementing an interface of this category will have some special behavior.

Hence, an empty interface in java is called a marker interface. In java we have the following major marker interfaces as under:

Searilizable interface

Cloneable interface

Remote interface

ThreadSafe interface

The marker interface can be described as a design pattern which is used by many languages to provide run-time type information about the objects. The marker interface provides a way to associate metadata with the class where the language support is not available.

**In java, garbage means unreferenced objects.?**

**garbage collection** (**GC**) is a form of automatic [memory management](https://en.wikipedia.org/wiki/Memory_management). Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects. The garbage collector, or just collector, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy.

**Advantages of Garbage Collection :**

It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.

It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

**Dangling Else problem**

When there is multiple IF and a single ELSE then the ELSE part doesn't get a clear view to go with which IF, this problem is called dangling else problem.

For Eg:

if(....)

{}

if(.....)

{}else{}  
Different parsers will interpret it differently, as you have not provided a definite unambiguous way as to how it should be.

[Dangling pointer](https://en.wikipedia.org/wiki/Dangling_pointer) bugs, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is dereferenced. By then the memory may have been reassigned to another use, with unpredictable results.

Double free bugs, which occur when the program tries to free a region of memory that has already been freed, and perhaps already been allocated again.

Certain kinds of memory leaks, in which a program fails to free memory occupied by objects that have become unreachable, which can lead to memory exhaustion. (Garbage collection typically does not deal with the unbounded accumulation of data that is reachable, but that will actually not be used by the program.)

Efficient implementations of persistent data structure.

**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(){}

**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.

A house has 3 switches on the ground floor and bulb on the 1st floor. Can you determine which is the correct switch of the bulb if you are allowed to go upstairs only once. Explain your ans?

i will switch on first switch and after a while switch it off. then after waiting for 5 min i will switch on 2 switch and go up.if bulb is on then switch 2 is right.if not tand bulb is hot then 1st switch was right else 3rd key will be right.

**What is time complexity?**

The **time complexity** of an algorithm quantifies the amount of **time** taken by an algorithm to run as a function of the length of the string representing the input. The **time complexity** of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms.

**Constant time:**

An algorithm is said to be **constant time** (also written as **O(1)** time) if the value of T(n) is bounded by a value that does not depend on the size of the input.

**Logarithmic time:**

An algorithm is said to take **logarithmic time** if T(n) = **O(log**n**)**. Due to the use of the [binary numeral system](https://en.wikipedia.org/wiki/Binary_numeral_system) by computers, the [logarithm](https://en.wikipedia.org/wiki/Logarithm) is frequently base 2 (that is, log2 n, sometimes written lg n). However, by the [change of base](https://en.wikipedia.org/wiki/Logarithmic_identities#Changing_the_base) for logarithms, loga n and logb n differ only by a constant multiplier, which in big-O notation is discarded; thus O(log n) is the standard notation for logarithmic time algorithms regardless of the base of the logarithm.

**Polylogarithmic time:**

An algorithm is said to run in [polylogarithmic](https://en.wikipedia.org/wiki/Polylogarithmic_function)**time** if T(n) = O((log n)k), for some constant k. For example, [matrix chain ordering](https://en.wikipedia.org/wiki/Matrix_chain_multiplication) can be solved in polylogarithmic time on a [Parallel Random Access Machine](https://en.wikipedia.org/wiki/Parallel_Random_Access_Machine).

**Sub-linear time:**

An algorithm is said to run in **sub-linear time** (often spelled **sublinear time**) if T(n) = o(n). In particular this includes algorithms with the time complexities defined above, as well as others such as the O(n½) [Grover's search](https://en.wikipedia.org/wiki/Grover%27s_algorithm) algorithm.

**Linear time:**

An algorithm is said to take **linear time**, or **O(**n**)** time, if its time complexity is O(n). Informally, this means that for large enough input sizes the running time increases linearly with the size of the input. For example, a procedure that adds up all elements of a list requires time proportional to the length of the list. This description is slightly inaccurate, since the running time can significantly deviate from a precise proportionality, especially for small values of n.

**Quasilinear time:**

An algorithm is said to run in quasilinear time if T(n) = **O(**n**log**kn**)** for some positive constant k; linearithmic time is the case k = 1.[[6]](https://en.wikipedia.org/wiki/Time_complexity#cite_note-6) Using [soft-O notation](https://en.wikipedia.org/wiki/Big_O_notation#Extensions_to_the_Bachmann.E2.80.93Landau_notations) these algorithms are Õ(n). Quasilinear time algorithms are also O(n1+ε) for every ε > 0, and thus run faster than any polynomial in n with exponent strictly greater than 1.

**Polynomial time:**

An algorithm is said to be of **polynomial time** if its running time is [upper bounded](https://en.wikipedia.org/wiki/Upper_bound) by a [polynomial expression](https://en.wikipedia.org/wiki/Polynomial_expression) in the size of the input for the algorithm, i.e., T(n) = O(nk) for some constant k.

Problem solving # 1 : (9 marks) Input is

  [1,34,3,98,9,76,45,4] and the output should be 998764543431

public class ABC {  
    public static String solve(int[] arr) {  
        String result = "";  
        Map> map = new TreeMap(new Comparator() {  
            public int compare(Integer o1, Integer o2) {  
                return o2 - o1;  
            }        });  
        for(int num: arr) {  
            int key = getKey(num);  
            Set set = map.get(key);  
            if(set == null) {  
                set = new TreeSet(new Comparator() {  
                    @Override  
                    public int compare(Integer o1, Integer o2) {  
                        return getValueToCompare(o2) - getValueToCompare(o1);  
                    }                });  
                map.put(key, set);  
            }            set.add(num);  
        }  
        for(Map.Entry> entry: map.entrySet() ) {  
            for(Integer num: entry.getValue()) {  
                result = result + num;  
            }        }  
        return result;  
    }  
    static int getValueToCompare(int num) {  
        while(num > 10) {  
            num = num % 10;  
        }        return num;    }  
    static int getKey(int num) {  
        while(num > 10) {  
            num = num/ 10;  
        }        return num;    }  
    public static void main(String[] args) {  
        System.out.println(solve(new int[]{1,34,3,98,9,76,45,4}));  
    }}

# Java Errors