* **JVM internals must read**

[**https://www.geeksforgeeks.org/jvm-works-jvm-architecture/**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)

* **Jdk vs jre vs jvm**

[**https://www.javatpoint.com/difference-between-jdk-jre-and-jvm**](https://www.javatpoint.com/difference-between-jdk-jre-and-jvm)

### JVM

JVM (Java Virtual Machine) is an abstract machine. e

JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform independent.

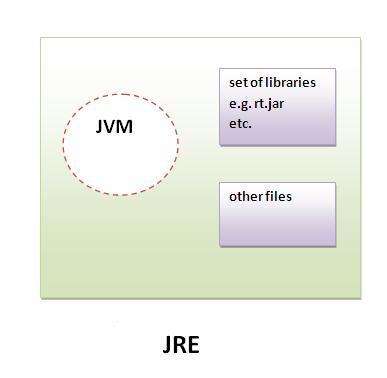
The JVM performs following main tasks:

* Loads code
* Verifies code
* Executes code
* Provides runtime environment

### JRE

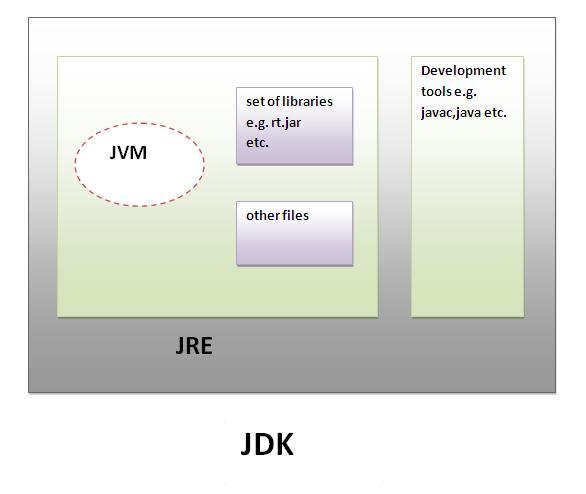
JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment. It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



### JDK

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.



* **Internal details of jvm**

[**https://www.javatpoint.com/internal-details-of-jvm**](https://www.javatpoint.com/internal-details-of-jvm)

### What is .class File and Bytecode

* When Java source file is compiled by Java compiler it is converted into Java class file with .class extension.
* The Java class file contains Java bytecode (highly optimized set of instructions) which is executed by [Java Virtual Machine (JVM)](https://www.thecrazyprogrammer.com/2014/06/jvm-java-virtual-machine-architecture-and-structure.html).
* .class file contains symbols and each bytecode instruction is stored into one byte exactly.
* .class file of each class is separately stored. Its name is same as class name in source file. A Java program can have any number of classes. If below program is compiled then two .class file will be created with name A.class and B.class.
* **Core java**
* **Refresher**

A Java program is mostly a collection of objects talking to other objects by invoking each other's methods. Every object is of a certain type, and that type is defined by a class or an interface. Most Java programs use a collection of objects of many different types.

* For classes, the names should typically be nouns
* **Object** At runtime, when the Java Virtual Machine (JVM) encounters the New keyword, it will use the appropriate class to make an object which is an instance of that class. That object will have its own state, and access to all of the equales behaviors defined by its class.
* **State(instance variables)** Each object (instance of a class) will have its own unique set of instance variables as defined in the class. Collectively, the values assigned to an object's instance variables make up the object's state.
* **Behavior(methods)** When a programmer creates a class, she creates methods for that class. Methods are where the class' logic is stored. Methods are where the real work gets done. They are where algorithms get executed, and data gets manipulated.

In addition, the names should typically be verb-noun pairs.

For example:

getBalance

doCalculation

setCustomerName

For interfaces, the names should typically be adjectives like

Runnable

Serializable

* **Class Declarations and Modifiers**

Modifiers fall into two categories:

■ Access modifiers: public, protected, private.

■ Non-access modifiers (including strictfp, final, and abstract).

a class can be declared with only public or default access; the other two access control levels don't make sense for a class

**Class Access**

What does it mean to access a class? When we say code from one class (class A) has

access to another class (class B), it means class A can do one of three things:

■Create an instanceof class B.

■ Extend class B (in other words, become a subclass of class B).

■ Access certain methods and variables within class B, depending on the access control of those methods and variables.

In effect, access means visibility. If class A can't see class B, the access level of the methods and variables within class B won't matter; class A won't have any way to access those methods and variables

**Default Access** A class with default access has no modifier preceding it in the declaration! It's the access control you get when you don't type a modifier in the class declaration. Think of default access as package-level access, because a class with default access can be seen only by classes within the same package. For example, if class A and class B are in different packages, and class A has default access, class B won't be able to create an instance of class A, or even declare a variable or return type of class A. In fact, class B has to pretend that class A doesn't even exist, or the compiler will complain.

**Public Access** A class declaration with the public keyword gives all classes from all packages access to the public class. In other words, all classes in the Java Universe (JU) have access to a public class

**Other (Non access) Class Modifiers**

**Strictfp** strictfp is a keyword and can be used to modify a class or a method, but never a variable. Marking a class as strictfp means that any method code in the class will conform to the IEEE 754 standard rules for floating points. Without that modifier, floating points used in the methods might behave in a platform-dependent way.

**Final Classes** When used in a class declaration, the final keyword means the class can't be subclassed. In other words, no other class can ever extend (inherit from) a final class, and any attempts to do so will give you a compiler error.

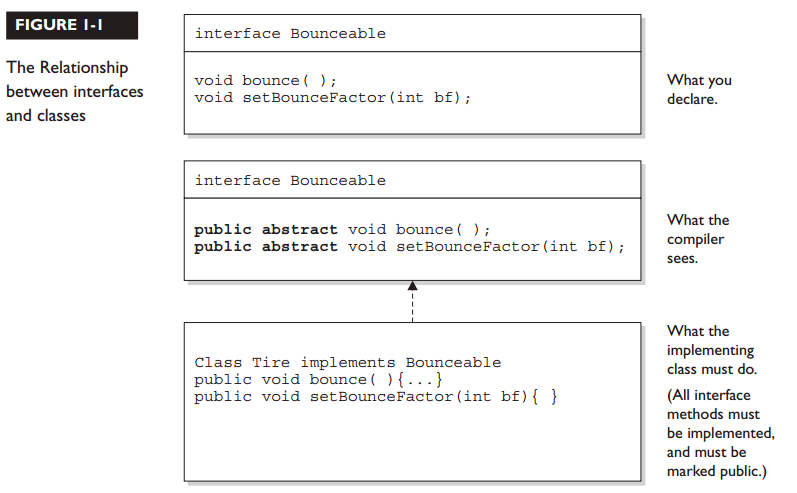
**Abstract Classes** An abstract class can never be instantiated. Its sole purpose, mission in life, raison d'être, is to be extended (subclassed).Notice that the methods marked abstract end in a semicolon rather than curly braces.

You can't mark a class as both abstract and final. They have nearly opposite meanings. An abstract class must be subclassed, whereas a final class must not be

subclassed. If you see this combination of abstract and final modifiers, used for a class or method declaration, the code will not compile.

**Declaring an Interface**: When you create an interface, you're defining a contract for what a class can do, without saying anything about how the class will do it. An interface is a contract. You could write an interface Bounceable, for example, that says in effect, "This is the Bounceable interface. Any class type that implements this interface must agree to write the code for the bounce()and setBounceFactor()methods."

By defining an interface for Bounceable, any class that wants to be treated as a Bounceable thing can simply implement the Bounceable interface and provide code for the interface's two methods. Interfaces can be implemented by any class, from any inheritance tree. This lets you take radically different classes and give them a common characteristic.



Typing in the abstract modifier is considered redundant; interfaces are implicitly abstract whether you type abstract or not. You just need to know that both of these declarations are legal, and functionally identical:

public abstract interface Rollable { }

public interface Rollable { }

**interface contants must always be public static final**

**public interfaces vs default interfaces ?**

**Java 8 default interface methods and static interface methods**

For creating a default method in java interface, we need to use “default” keyword with the method signature. For example,

package com.journaldev.java8.defaultmethod;

public interface Interface1 {

void method1(String str);

default void log(String str){

System.out.println("I1 logging::"+str);

}

}

Notice that log(String str) is the default method in the Interface1. Now when a class will implement Interface1, it is not mandatory to provide implementation for default methods of interface. This feature will help us in extending interfaces with additional methods, all we need is to provide a default implementation.

Let’s say we have another interface with following methods:

package com.journaldev.java8.defaultmethod;

public interface Interface2 {

void method2();

default void log(String str){

System.out.println("I2 logging::"+str);

}

}

We know that Java doesn’t allow us to extend multiple classes because it will result in the “Diamond Problem” where compiler can’t decide which superclass method to use. With the default methods, the diamond problem would arise for interfaces too. Because if a class is implementing both Interface1 and Interface2 and doesn’t implement the common default method, compiler can’t decide which one to chose.

Extending multiple interfaces are an integral part of Java, you will find it in the core java classes as well as in most of the enterprise application and frameworks. So to make sure, this problem won’t occur in interfaces, it’s made mandatory to provide implementation for common default methods of interfaces. So if a class is implementing both the above interfaces, it will have to provide implementation for log() method otherwise compiler will throw compile time error.

Important points about java interface default methods:

1. Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
2. Java interface default methods has bridge down the differences between interfaces and abstract classes.
3. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.
4. Java interface default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can chose which one to override.
5. One of the major reason for introducing default methods in interfaces is to enhance the Collections API in Java 8 to support lambda expressions.
6. If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from java.lang.Object. The reasoning is very simple, it’s because Object is the base class for all the java classes. So even if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That’s why to avoid confusion, we can’t have default methods that are overriding Object class methods.
7. Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

## Java Interface Static Method

Java interface static method is similar to default method except that we can’t override them in the implementation classes. This feature helps us in avoiding undesired results incase of poor implementation in implementation classes. Let’s look into this with a simple example.

package com.journaldev.java8.staticmethod;

public interface MyData {

default void print(String str) {

if (!isNull(str))

System.out.println("MyData Print::" + str);

}

static boolean isNull(String str) {

System.out.println("Interface Null Check");

return str == null ? true : "".equals(str) ? true : false;

}

}

Now let’s see an implementation class that is having isNull() method with poor implementation.

package com.journaldev.java8.staticmethod;

public class MyDataImpl implements MyData {

public boolean isNull(String str) {

System.out.println("Impl Null Check");

return str == null ? true : false;

}

public static void main(String args[]){

MyDataImpl obj = new MyDataImpl();

obj.print("");

obj.isNull("abc");

}

}

Note that isNull(String str) is a simple class method, it’s not overriding the interface method. For example, if we will add [@Override annotation](https://www.journaldev.com/817/java-override-annotation) to the isNull() method, it will result in compiler error.

Now when we will run the application, we get following output.

Interface Null Check

Impl Null Check

If we make the interface method from static to default, we will get following output.

Impl Null Check

MyData Print::

Impl Null Check

Java interface static method is visible to interface methods only, if we remove the isNull() method from the MyDataImpl class, we won’t be able to use it for the MyDataImpl object. However like other static methods, we can use interface static methods using class name. For example, a valid statement will be:

boolean result = MyData.isNull("abc");

Important points about java interface static method:

1. Java interface static method is part of interface, we can’t use it for implementation class objects.
2. Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.
3. Java interface static method helps us in providing security by not allowing implementation classes to override them.
4. We can’t define interface static method for Object class methods, we will get compiler error as “This static method cannot hide the instance method from Object”. This is because it’s not allowed in java, since Object is the base class for all the classes and we can’t have one class level static method and another instance method with same signature.
5. We can use java interface static methods to remove utility classes such as Collections and move all of it’s static methods to the corresponding interface, that would be easy to find and use.

**Access Modifiers methods and variables**

Because method and variable members are usually given access control in exactly the same way, we'll cover both in this section. Where as a class can use just two of the four access control levels (default or public), members can use all four:

■ public

■ protected

■ default

■ private

What does it mean for code in one class to have access to a member of another class? For now, ignore any differences between methods and variables. If class A has access to a member of class B, it means that class B's member is visible to class A. When a class does not have access to another member, the compiler will slap you for trying to access something that you're not even supposed to know exists!

You need to understand two different access issues:

■ Whether method code in one class can access a member of another class

■ Whether a subclass can inherit a member of its superclass

The first type of access is when a method in one class tries to access a method or a variable of another class, using the dot operator (.) to invoke a method or retrieve a variable. For example:

class Zoo {

public String coolMethod() {

return "Wow baby";

}

}

class Moo {

public void useAZoo() {

Zoo z = new Zoo();

// If the preceding line compiles Moo has access

// to the Zoo class

// But... does it have access to the coolMethod()?

System.out.println("A Zoo says, " + z.coolMethod());

// The preceding line works because Moo can access the

// public method

}

}

The second type of access revolves around which, if any, members of a superclass a subclass can access through inheritance. We're not looking at whether the subclass can, say, invoke a method on an instance of the superclass (which would just be an example of the first type of access). Instead, we're looking at whether the subclass inherits a member of its superclass. Remember, if a subclass inherits a member, it's exactly as if the subclass actually declared the member itself. In other words, if a subclass inherits a member, the subclass has the member. Access Modifiers (Exam Objectives 1.3 and 1.4) 25

class Zoo {

public String coolMethod() {

return "Wow baby";

}

}

class Moo extends Zoo {

public void useMyCoolMethod() {

// Does an instance of Moo inherit the coolMethod()?

System.out.println("Moo says, " + this.coolMethod());

// The preceding line works because Moo can inherit the

// public method

// Can an instance of Moo invoke coolMethod() on an

// instance of Zoo?

Zoo z = new Zoo();

System.out.println("Zoo says, " + z.coolMethod());

// coolMethod() is public, so Moo can invoke it on a Zoo

//reference

}

}

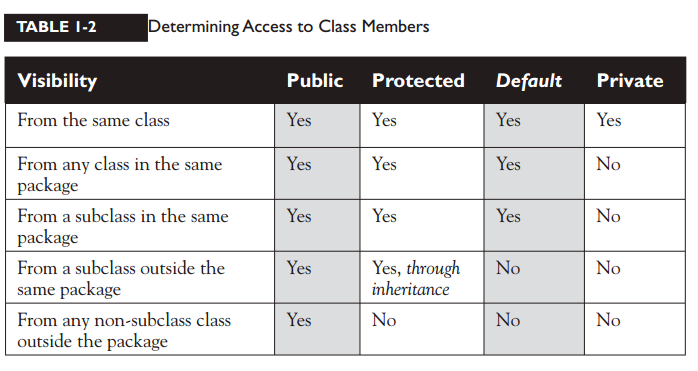
you need to recognize that a subclass can't see, use, or even think about the private members of its superclass. You can, however, declare a matching method in the subclass. But regardless of how it looks, it is not an overriding method! It is simply a method that happens to have the same name as a private method (which you're not supposed to know about) in the superclass. The rules of overriding do not apply, so you can make this newly-declared-but-just-happens-to-match method declare new exceptions, or change the return type, or anything else you want to do with it.

So, when you think of default access, think package restriction. No exceptions. But when you think protected, think package + kids. A class with a protected member is marking that member as having package-level access for all classes, but with a special exception for subclasses outside the package.

The subclass can see the protected member only through inheritance.

For a subclass outside the package, the protected member can be accessed only through inheritance. No! Once the subclass-outside-the-package inherits the protected member, that member (as inherited by the subclass) becomes private to any code outside the subclass, with the exception of subclasses of the subclass Just remember that default members are visible to subclasses only if those subclasses are in the same package as the superclass.

**Local Variables and Access Modifiers** Can access modifiers be applied to local variables? NO !



**Final Methods**

The final keyword prevents a method from being overridden in a subclass, and is often used to enforce the API functionality of a method. For example, the Thread class has a method called isAlive()that checks whether a thread is still active. If you extend the Thread class, though, there is really no way that you can correctly implement this method yourself (it uses native code, for one thing), so the designers have made it final. Just as you can't subclass the String class In other words, a final argument must keep the same value that the parameter had when it was passed into the method.

**Abstract Methods** : An abstract method is a method that's been declared(as abstract) but not implemented. You mark a method abstract when you want to force subclasses to provide the implementation. It is illegal to have even a single abstract method in a class that is not explicitly declared abstract The rule is this: The first concrete subclass of an abstract class must implement all abstract methods of the superclass.

Synchronized Methods

The synchronized keyword indicates that a method can be accessed by only one thread at a time.

Native Methods

The native modifier indicates that a method is implemented in platform-dependent code, often in C

**Final variable**: A reference variable marked final can't ever be reassigned to refer to a different object. The data within the object can be modified, but the reference variable cannot be changed. In other words, a final reference still allows you to modify the state of the object it refers to, but you can't modify the reference variable to make it refer to a different object.

Burn this in: there are no final objects, only final references.

Transient Variables If you mark an instance variable as transient, you're telling the JVM to skip (ignore) this variable when you attempt to serialize the object containing it.

Volatile Variables: The volatile modifier tells the JVM that a thread accessing the variable must always reconcile its own private copy of the variable with the master copy in memory. Say what? Don't worry about it. For the exam, all you need to know about

\* Mostly used in mutil-threaded env, in case of a volatile modifier happens before relationship happens, i.e. all the writes to the variable happen before it is read by any thread, in normal cases it is quite possible for a thread to read half baked changes on the variable thereby causing data inconsisty issues

Volatile is that, as with transient, it can be applied only to instance variables.

Local variables are alwys on the stack not on the heap.

Enums can be declared as their own separate class, or as a class member,

however they must not be declared within a method !

* **Transient usage:**

It marks a member variable not to be serialized when it is persisted to streams of bytes. When an object is transferred through the network, the object needs to be 'serialized'. Serialization converts the object state to serial bytes. Those bytes are sent over the network and the object is recreated from those bytes. Member variables marked by the java transient keyword are not transferred, they are lost intentionally.

For example, you may have fields that are derived from other fields, and should only be done so programmatically, rather than having the state be persisted via serialization.

**@transient vs transit keyword**

* **Volatile usage in java**

volatile keyword guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.

From java concurrency [tutorial](https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html):

Using volatile variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable

This means that changes to a volatile variable are always visible to other threads. It also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up the change.

Regarding your query:

How do I know when I should mark a variable volatile? What are the rules of thumb when figuring out what variables should be volatile in multithreaded code?

If you feel that all reader threads always get latest value of a variable, you have to mark variable as volatile

If you have one writer thread to modify the value of variable and multiple reader threads to read the value of variable, volatile modifier guarantees memory consistency.

If you have multiple threads to write and read variables, volatile modifier alone does not guaranty memory consistency. You have to synchronize the code or use high level [concurrency](https://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html) constructs like Locks, Concurrent Collections, Atomic variables etc.

* **Volatile vs synchronized**

<http://stackoverflow.com/questions/3519664/difference-between-volatile-and-synchronized-in-java/>

* ENUMS

Enums are lists of constants. When you need a predefined list of values which do represent some kind of numeric or textual data, you should use an enum. For [instance](http://crunchify.com/lazy-creation-of-singleton-threadsafe-instance-without-using-synchronized-keyword/), in a chess game you could represent the different types of pieces as an enum:

enum ChessPiece {

PAWN,

ROOK,

KNIGHT,

BISHOP,

QUEEN,

KING;

}

You should always use enums when a [variable](http://crunchify.com/what-is-an-interface-in-java-how-its-used-java-tutorial-example-attached/) (especially a method parameter) can only take one out of a small set of possible values. Examples would be things like type [constants](http://crunchify.com/java-static-methods-variables-static-block-and-class-with-example/) (contract status: “permanent”, “temp”, “apprentice”), or flags (“execute now”, “defer execution”).

If you use enums instead of [integers](http://crunchify.com/what-are-the-difference-between-jdbcs-statement-preparedstatement-and-callablestatement/) (or String codes), you increase [compile](http://crunchify.com/spring-framework-4-order-annotation-tutorial-sort-order-for-an-annotated-component/)-time checking and avoid [errors](http://crunchify.com/have-you-noticed-nullpointerexception-npe-there-are-few-best-practices-to-avoid-npe/) from passing in invalid constants, and you document which values are legal to use.

### Some very important points on Java Enum:

* All enums implicitly extend java.lang.Enum. Since Java does not support multiple [inheritance](http://crunchify.com/understanding-java-annotation-annotation-examples/), an enum cannot extend anything else.
* Enum in Java are type-safe: Enum has there own name-space. It means your enum will have a type for example “Company” in below [example](http://crunchify.com/java-nio-non-blocking-io-with-server-client-example-java-nio-bytebuffer-and-channels-selector-java-nio-vs-io/) and you can not assign any value other than specified in Enum Constants.
* You can specify values of enum constants at the creation time. MyEnum.values() returns an [array](http://crunchify.com/java-two-ways-to-convert-char-array-to-string/) of MyEnum’s values.
* Enum constants are implicitly static and [final](http://crunchify.com/in-java-how-to-perform-file-search-operation-using-java-nio-file-interface-tutorial-on-file-and-directory-operations/) and can not be changed once created.
* Enum can be safely compare using:

Switch-Case Statement

== Operator

[.equals()](http://crunchify.com/how-to-override-equals-and-hashcode-method-in-java/) method

* You can not create instance of enums by using new operator in Java because constructor of Enum in [Java](http://crunchify.com/category/java-web-development-tutorial/) can only be private and Enums constants can only be created inside Enums itself.
* Instance of Enum in Java is created when any Enum constants are first called or [referenced](http://crunchify.com/missing-maven-settings-xml-file-for-your-eclipse-what-if-you-need-two-settings-xml-file-for-work-personal-workspace/) in code.
* An enum specifies a list of constant values assigned to a [type](http://crunchify.com/java-tip-wherever-possible-try-to-use-primitive-types-instead-of-wrapper-classes/).
* An enum can be declared outside or inside a class, but NOT in a method.
* An enum declared outside a class must NOT be marked static, final , [abstract](http://crunchify.com/what-is-an-abstract-class-and-abstract-method-in-java-when-should-i-use-it/), protected , or private
* Enums can contain [constructors](http://crunchify.com/create-simple-pojo-and-multiple-java-reflection-examples/), methods, variables, and constant class bodies.
* enum constants can send arguments to the enum constructor, using the syntax BIG(8), where the int literal 8 is passed to the enum constructor.
* enum constructors can have arguments, and can be [overloaded](http://crunchify.com/java-method-overriding-examples-and-concepts-overriding-rules/).
* enum constructors can NEVER be invoked directly in code. They are always called [automatically](http://crunchify.com/how-to-use-expiringmap-maven-java-utility-to-remove-expired-objects-from-map-automatically-complete-java-tutorial/) when an enum is initialized.
* The semicolon at the end of an enum declaration is optional.
* **Grabage Collection**

<https://www.cubrid.org/blog/understanding-java-garbage-collection>

* **Garbage collection in java**

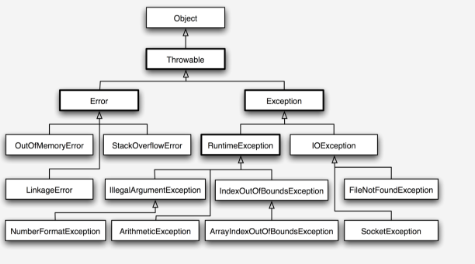
<https://www.geeksforgeeks.org/garbage-collection-java/>

* **Exception Handling:**
* An exception is an unwanted or unexpected condition or situation that alters the normal program flow. It results in abnormal program termination
* Exception handling is a mechanism to provide graceful termination of program in case if exception happens.
* With exception handling we define an alternate flow or path to be taken in case if exception occurs
* Whenever an exception occurs the method in which the exception happens creates an exception object and calls the JVM. The JVM then checks if the method has done any exception handling, if its done then it will be corresponding catch block will be called and program will to execute.
* Incase if no exception handling is done it will pop that method stack-frame from the calling stack, and check if the calling method has any exception handling and so on till main method upon which the default exception handler will be invoked and the program will terminate abnormally.
* Normal vs Abnormal termination
* Even if exception occurs close all system resources e.g. db connections is normal termination, vice-verse is abnormal.
* Try-catch
* Only cover the risky code in try catch. And void wrapping all the code in try-catch.
* Use catch from more specific(sub-class) to superclass.
* Strack Trace:

Exception in thread XXX : Name of exception : description

stackTrace ()

* Exception Class Hierarchy



* Throwable class is at the root, it has only 2 direct sub-classes Exception and Error
* Exceptions: these are mostly recoverable. And caused by user program
* Error: Are not recoverable and caused by lack of system resources.
* **Checked Vs Unchecked**
* All exception occur at run-time only. But the JVM is smart enough to perform some compile-time checks in mostly 80% of the cases. For e.g. if you code is trying to open a file for reading, the jvm will enforce you to perform compile time check of handle obvious exception cases like what if the file is not there. So it enforces you to provide exception handling for such type of exceptions know as checked exceptions or compile Time exception. Mostly handled via try-catch or throws
* But if this file is too large and we run out of memory,
* This will not be know to the compiler at run time so there is no handling can be provided, this is unchecked exceptions
* Error class and all its sub-classes also RuntimeExceptions are
* **Throw:**
* Used to throw exception object explicitly, mostly used in case of custom exceptions
* **Throws:**

delegates exception handling to the calling method. It is recommended to use try-catch insetead of throws as if you keep delegating till the end i.e. till main it results in abnormal termination. Only used for checked exceptions

# [Can we have try without catch block in java](http://www.instanceofjava.com/2016/04/can-we-have-try-without-catch-in-java.html) It is possible to have try block without catch block by using finally block

* Java supports try with finally block
* As we know finally block will always executes even there is an exception occurred in try block, Except System.exit() it will executes always.
* We can place logic like connections closing or cleaning data  in finally.
* package exceptionsInterviewQuestions;
* public class TryWithoutCatch {

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

* try {
* System.out.println("inside try block");
* } finally{
* System.out.println("inside finally block");
* }
* }
* }

Finally block executes Even though the method have return type and try block returns something

1. package exceptionsInterviewQuestions;
3. public class TryWithFinally {
5. public static int method(){

8. try {
10. System.out.println("inside try block");
12. return 10;
13. } finally{
15. System.out.println("inside finally block");
16. }
18. }
20. public static void main(String[] args) {
22. System.out.println(method());
24. }
26. }

**Output:**

1. inside try block
2. inside finally block
3. 10

**What happens if exception raised in try block?**

* Even though exception raised in try block finally block executes.



* **What is exception occues in finally block and is not handled**

An exception thrown from the finally block will replace any exception that was thrown from the try, and information about the real problem is likely to be lost.

* ARM in java7, **Automatic Resource Management**

A resource is an object that must be closed once your program is done using it. For example a File resource or JDBC resource for database connection or a Socket connection resource. Before Java 7, there was no auto resource management and we should explicitly close the resource once our work is done with it. Usually, it was done in the finally block of a try-catch statement. This approach used to cause memory leaks and performance hit when we forgot to close the resource.

try(// open resources here){

// use resources

} catch (FileNotFoundException e) {

// exception handling

}// resources are closed as soon as try-catch block is executed.

* AutoClosable interface in java,

When you have your own custom resource in that case you need to implement this interface, and override the close() method.

AutoCloseable (introduced in Java 7) makes it possible to use try-with-resources idiom:

public class MyResource implements AutoCloseable {

public void close() throws Exception {

System.out.println("Closing!");

}

}

now you can say:

try(MyResource res = new MyResource()) {

//use res here

}

* What happens when a thread throws and exception ?

<https://stackoverflow.com/questions/6546193/how-to-catch-an-exception-from-a-thread>

* What are Upper and Lower bounds in Generics? Where to choose one?

1. The question mark i.e. ? is a wild card element. It means any type If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.
2. ? extends T means **any class which extends T**. Thus, we are referring to the children of *T*. Hence, **T is the upper bound. The upper-most class in the inheritance hierarchy**
3. ? super T means **any class / interface which is super of T**. Thus we are referring to all the parents of *T*.  **T is thus the lower bound. The lower-most class in the inheritance hierarchy**

* What is type erasure
* Type Erasure. Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to: Replace all typeparameters in generic types with their bounds or Object if thetype parameters are unbounded.
* **Threads**
* **Basic concepts**

Multi-threading is a popular form of multi-tasking in java.

**Multi-Tasking**

* + **Multi-processing Multi-threading**
* **Multi – processing/ process-based multi-tasking**
* E.g. multiple proceses/applications running at the same time e.g. eclipse and windows media player
* These are 2 different processes
* Have different address space
* Inter process communication is exp
* Also called as process-based multi-tasking, as every process in running simultaneously
* **Multi -threading:**
* These are separate tasks within the same process

e.g. spell checking and typing at the same time in MS word

* these tasks are light weight processing
* these share the same address space as their parent process
* context switching is easy between threads.
* **Multi-threading in java:**
* Multi-threading in java means having 2 execution stacks, one with
* Main and other with run, and context –switching between the 2 stacks happens so fast giving an over-all illusion of running these 2 stacks parallel.



* **Thread scheduling is mostly done by the underlyning OS,**

**Mostly Unix version use preamtive scheduler.**

* **Java threads vs OS threads the**

There is no generic solution how Java threads are mapped to OS threads, if at all. Every JVM implementation can do it in a different way.

* **Runnable vs extending thread class**

Yes: implements Runnable is the preferred way to do it, IMO. You're not really specializing the thread's behavior. You're just giving it something to run. That means [composition](http://en.wikipedia.org/wiki/Object_composition) is the philosophically "purer" way to go.

In general, I would recommend using something like Runnable rather than Thread because it allows you to keep your work only loosely coupled with your choice of concurrency. For example, if you use a Runnable and decide later on that this doesn't in fact require it's own Thread, you can just call threadA.run().

**Caveat:** Around here, I strongly discourage the use of raw Threads. I much prefer the use of [Callables](http://java.sun.com/javase/6/docs/api/java/util/concurrent/Callable.html) and [FutureTasks](http://java.sun.com/javase/6/docs/api/java/util/concurrent/FutureTask.html) (From the javadoc: "A cancellable asynchronous computation"). The integration of timeouts, proper cancelling and the thread pooling of the modern concurrency support are all much more useful to me than piles of raw Threads.

**Follow-up:** there is a [FutureTask constructor](http://java.sun.com/javase/6/docs/api/java/util/concurrent/FutureTask.html#FutureTask%28java.lang.Runnable,%20V%29) that allows you to use Runnables (if that's what you are most comfortable with) and still get the benefit of the modern concurrency tools. To quote the javadoc:

If you don't need a particular result, consider using constructions of the form:

Future<?> f = new FutureTask<Object>(runnable, null)

So, if we replace their runnable with your threadA, we get the following:

new FutureTask<Object>(threadA, null)

Another option that allows you to stay closer to Runnables is a [ThreadPoolExecutor](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html). You can use the [execute](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html#execute%28java.lang.Runnable%29) method to pass in a Runnable to execute "the given task sometime in the future."

If you'd like to try using a thread pool, the code fragment above would become something like the following (using the [Executors.newCachedThreadPool()](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html#newCachedThreadPool%28%29) factory method):

ExecutorService es = Executors.newCachedThreadPool();

es.execute(new ThreadA());

* **How to run a multi-threaded code**

suspend JVM option in debug mode, will trace only the current running thread

* **How to debug a multi-threaded code**

Suspend JVM option in debug mode, lets you trace execution of the currently running thread only

* **Sleep method:**

Suspends the execution for that much millseconds of time. Needed in case of to deliberately introduce delays.

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system.

On calling sleep the current thread goes in waiting state for that much millsec and come to runnable state once the sleep time is over

It does not release any lock it holds.

runnable running dead

New/

* **Join method:** The tread on which it is called, is force to die. i.e. complete its execution, and only then other threads will resume execution. The thread on which join is called is run exclusively till completion i.e. it is not context-switched. Once this thread is dead, i.e finished execution other threads run as normal.

T1,t2,t3 are 3 threads, on

T1.start();

T2.start();

t3.start();

If called t1.join();

Now t1 will run start to finish, and only after that t2 and t3 will run.

1. **class** TestJoinMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod1 t1=**new** TestJoinMethod1();
12. TestJoinMethod1 t2=**new** TestJoinMethod1();
13. TestJoinMethod1 t3=**new** TestJoinMethod1();
14. t1.start();
15. **try**{
16. t1.join();
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }
23. **Output:1 ,2 ,3 ,4 ,5 ,1 ,1 ,2 ,2 ,3 ,3 ,4 ,4 ,5 ,5**

As you can see in the above example, when t1 completes its task then t2 and t3 starts executing.

1. **try**{
2. t1.join(1500);
3. }**catch**(Exception e){System.out.println(e);}

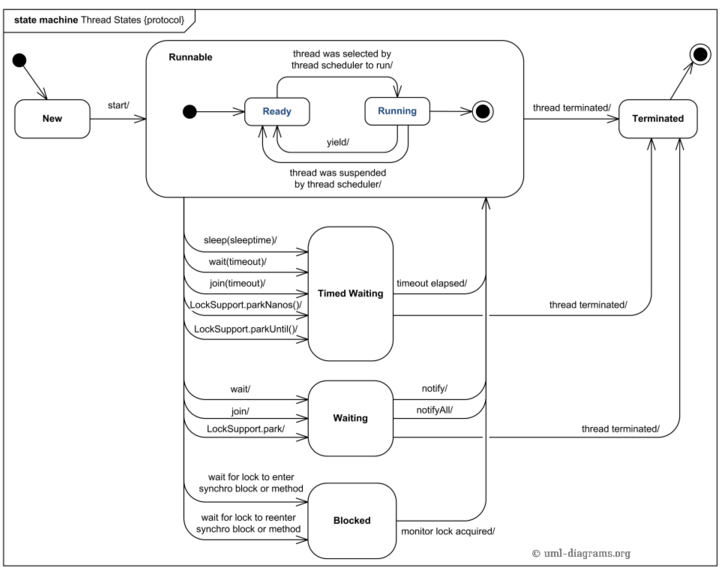
In the above example,when t1 is completes its task for 1500 miliseconds(3 times) then t2 and t3 starts executing.

* **What happens to Java thread after a join call with timeout**
* **Join vs Join(timestamp)**

Join(0) is as good as join();

join will cause the calling thread to wait for the thread join was invoked on, to die, i.e. finish it's execution. So the state of thread in your example at the moment the timeout has expired will be anything that is not [TERMINATED](http://download.oracle.com/javase/6.0/docs/api/java/lang/Thread.State.html#TERMINATED) (which is why the timeout occurred as opposed to join returning 'naturally' (in which case thread would be in the TERMINATED state) - of course, thread could almost immediately transition into the TERMINATED state, just after the timeout happened).

The state of the calling thread will become RUNNABLE immediately after the time expires and until that time, it would have in the [TIMED\_WAITING](http://download.oracle.com/javase/6.0/docs/api/java/lang/Thread.State.html#TIMED_WAITING) state.

* Thread life cycle:
* 

[**http://www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html**](http://www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html)

* **run**
* **start**
* **Join vs join(ts)**
* **yield()**

this method places, the current running thread in the runnable state, and promotes a thread with similar or higher priority from the runnable state to running state.

* Pauses the execution of the current running thread, and gives changes for similar threads with high priority to execute.
* On most JVMs call to yield() si it internally mapped to sleep(0) method
* Join: if a threads wants to wait until some other threads completes its execution, then it should call join method.
* Eg.g if thread t1 wants to wait unitll t2 completes, then t1 has to call t2.join()

If t1 executes t2.join(), then t1 will entry waiting state until t2 completes.

Once t2 completes than t1 can continue its execution

After calling join, thread t1 will entry the waiting state, and remain in it until t1 finishes execution or if its timed waiting i.e. t2.join(1000ms), it will wait for 1000ms and come to runnable state,

Also if any interrupt happens, i.e the waiting thread is interrupted then it comes to runnable state.

* If a thread calls join thread on the same thread itself, then the program will be stucked, this is something similar to deadlock. In this case thread has to wait infinite amount of time.
* E.g.
* Main(String[] args) throws IE {
* Thread.currentThread().join();
* }
* **Wait**

Causes the calling thread to release the lock held on it, and goes in the blocked/waiting state.

* **Notify**

Notifies a similar thread in the waiting state, waiting on the same lock held by the calling thread.

* **notifyAll**

notifies all threads in the waiting state, waiting on the same lock held by the calling thread

* **Demon thread:**

These are background processing threads, which the jvm create to help in the execution of the main thread, eg. Gc, finailzer are all demon threads. The JVM terminates these demon threads once the user threads finish execution.

* **Thread pool:**

represents a group of worker threads that are waiting for the job and reuse many times.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

Better performance It saves time because there is no need to create new thread.

It is used in Servlet and JSP where container creates a thread pool to process the request.

* **Thread group:**

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

ThreadGroup tg1 = new ThreadGroup("Group A");

Thread t1 = new Thread(tg1,new MyRunnable(),"one");

Thread t2 = new Thread(tg1,new MyRunnable(),"two");

Thread t3 = new Thread(tg1,new MyRunnable(),"three");

* **Thread Local**
* **What is interrupt ?**

An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate.

How is it implemented?

The interrupt mechanism is implemented using an internal flag known as the interrupt status. Invoking Thread.interrupt sets this flag. When a thread checks for an interrupt by invoking the static method Thread.interrupted, interrupt status is cleared. The non-static Thread.isInterrupted, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

* How to interrupt a thread explicitly via code
* **Synchronization in java**

In a multi-threaded env, when 2 or more therads try to access the same code, they might produce unwanted resrults e.g if 2 threads write to the same file.

So synchronization is a mechanism in java where you can control the way multiple threads access the share resource.

Process Synchronization

Synchronization Mutual Exclusion

-sync block

-sync method

- static synchronization

Thread Synchronization

Inter-thread Communication

-wait, notify, notifyAll

* **Sncynhorized keyword, is only applicable to methods and blocks, not for classes and variables**
* **Concept of Lock in Java**

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

* **Class level lock,**
* **Every class in java has a unique lock, which is also known as class level lock. If a thread wants to execute any static synchronized method it should acquire this class level lock.**
* **Under the hood, at the jvm level the class lock is also a object level lock only, because every object creates a class object.**
* **System.gc vs addShutDownHook**
* **Thread life cycle**
* **Happens before relationship in java**
* **Synch method vs synch blocks**

Here are Some more differences between synchronized method and block in Java based upon experience and syntactical rules of synchronized keyword in Java. Though both block and method can be used to provide highest degree of synchronization in Java, use of synchronized block over method is considered as better Java coding practices.

1) One significant difference between synchronized method and block is that, Synchronized block generally reduce scope of lock. As scope of lock is inversely proportional to performance, its always better to lock only critical section of code. One of the best example of using synchronized block is double checked locking in Singleton pattern where instead of locking whole getInstance() method we only lock critical section of code which is used to create Singleton instance. This improves performance drastically because locking is only required one or two times.

2) Synchronized block provide granular control over lock, as you can use arbitrary any lock to provide mutual exclusion to critical section code. On the other hand synchronized method always lock either on current object represented by this keyword or class level lock, if its static synchronized method.

3) Synchronized block can throw throw java.lang.NullPointerException if expression provided to block as parameter evaluates to null, which is not the case with synchronized methods.

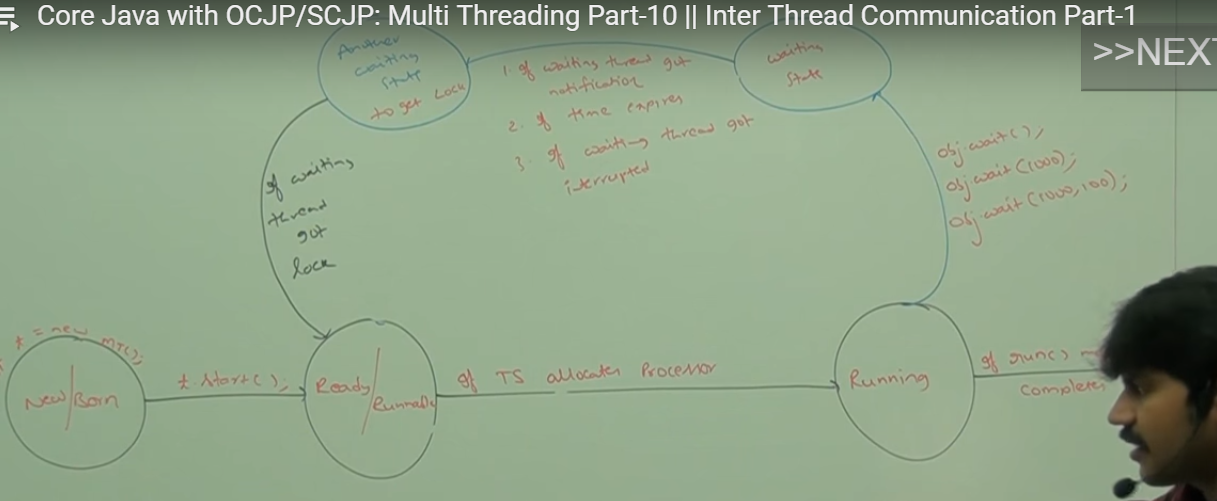
4) In case of synchronized method, lock is acquired by thread when it enter method and released when it leaves method, either normally or by throwing Exception. On the other hand in case of synchronized block, thread acquires lock when they enter synchronized block and release when they leave synchronized block.

* **Wait(), notify(), notifyAll()**
* **2 threads can communicate with each other via wati, notify, notifyAll methods**
* **The thread which is expecting updating is responsible to call wait() method, then immediately the thread will enter into waiting state,**
* **The thread which is responsible to perform updation, after performing updation it is responsible to call notify method, then waiting thread will get that notification and continue its execution with those updated object/items**
* **If a threa calls a wiat method on any object, it immediately releases the lock of that object, and goes in waiting state**
* **Same is with notify, but it may not release it immediately**
* **Except for wait, notify, notifyAll methods threads wont release a lock anywhere**
* **Why is wait(), notify(), notifyAll() defined in object class.**

**To call wait, notify, or notifyAll methds on any object, the calling thread should be owner of the object. i.e. it must acquire the lock on the object. It means the thread should be inside synch area**

* **Hence we can only call wait, notify and notifyAll methods only from synch area. Otherwise we get IllegalMoniterException as runtime exception**

**Thread state in case of wait, notify, notifyAll**

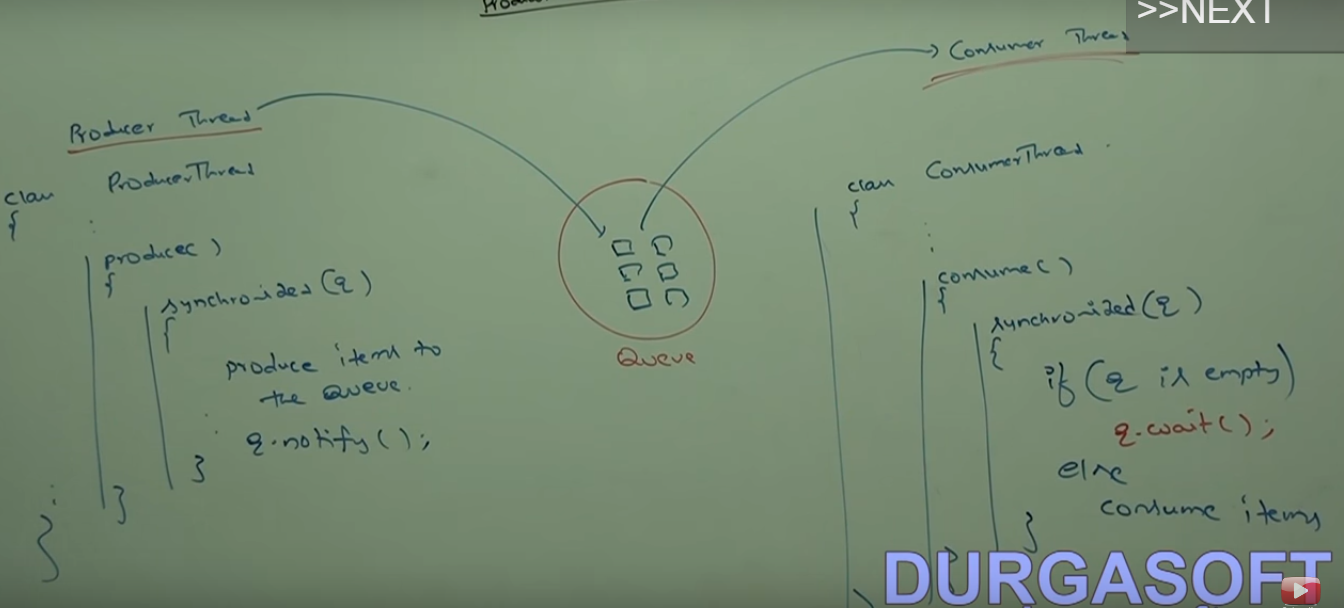


* **Incase if thread calls wait, it will release its lock held on the object and will go in waiting state**
* **It will come out of wait state only if some other thread calls notify, or it its waiting time expries, or it gets irrupted.**
* **Once it is notifyied or timed out, or inteerupted, then it agin needs to acquire the lock on the object,**
* **Thus it enters a new waiting state, jst to acquire a lock on the object, once acquired it goes to runnable state.**
* **Producer consumer problem is the best example for inter thread communication**
* **Here prodcer thread is responsible to produce items to the Q.**
* **Consumer thread is responsible to consume items from the Q**
* **So Q object is used as a common lock object ie.e**

**synchornized(Q){**

**}**

* **If Q is empty than consumer thread will call Q.wait() method and entered into waiting state**
* **After producing items to the Q, producer is responsible to call notify method then waiting consumer will get the notification and continue its execution with updated items**



* **Volatile**

New developers sometimes think volatility replaces synchronization. Although volatility, through keyword volatile, lets you assign values to long-integer or double-precision floating-point shared field variables outside a synchronized context, volatility cannot replace synchronization. Synchronization lets you group several operations into an indivisible unit, which you cannot do with volatility. However, because volatility is faster than synchronization, use volatility in situations where multiple threads must communicate via a single shared field variable

***Volatile is more suitable for varaibles, sync is more for methods or blocks.***

* ***Volatile vs Atomic***

***Volatile is about access, while atomic is about atomic operiation e.g. i++; is performed in one go.***

<https://stackoverflow.com/questions/19744508/volatile-vs-atomic>

* **Reentrant Synchronization**
* **What is object lock and class lock ?**
* No, it doesn't mean that. The "class level lock" is just a regular lock on a different object, namely SomeClass.class. The "object level lock" locks on this.
* **Edit:** Just to make sure I'm following your understanding of the terminology, you're wondering if m1 and m2 can be run concurrently as they are defined below:

public class SomeClass {

public synchronized static void m1() {

//do something

}

public synchronized void m2() {

//do something

}

}

And the answer is yes, m1 and m2 can be run concurrently. It is functionally equivalent to this:

public class SomeClass {

public static void m1() {

synchronized (SomeClass.class) {

//do something

}

}

public void m2() {

synchronized (this) {

//do something

}

}

}

Since they are synchronizing on completely different objects, they are not mutually exclusive.

**Executor Framework**

* **benefits of executor framework over raw threads**

|  |  |
| --- | --- |
|  | Prior to java 1.5 version, Thread/Runnable was designed for two separate services   1. Unit of work 2. Execution of that unit of work   ExecutorService decouples those two services by designating Runnable/Callable as unit of work and Executor as a mechanism to execute ( with lifecycling) the unit of work  The key concept is that *task submission is decoupled from the task-execution policy*, |

* **Why wait is always used inside while loop**
* **compare-and-swap (CAS) hardware instruction**
* **How threads are used in popular framework**
* **Executor interface**
* **Future interface**
* **Callable interface**
* **Callable vs runnable**
* **Runnable has run() method, callable has submit() method.**
* **Runnable return type is void, callable return type is V, often collected in the Type of FutureTask.**
* **Runnable does not have throws in its method signature, callable has throws in its method signature, so we can propogate exception to the caller mehod.**
* **run() can be called via both Thread and ExecutorService, submit only via executorSerivce**
* **Executor factory methods**
* **Fork-join framework**
* **Better ways of writing multi-threaded code**
* **Thread Executor framework in java**
* **Core pool size vs MaxPoolSize**

**-** Take this example. Starting thread pool size is 1, core pool size is 5, max pool size is 10 and the queue is 100. As requests come in, threads will be created up to 5 and then tasks will be added to the queue until it reaches 100. When the queue is full new threads will be created up to maxPoolSize. Once all the threads are in use and the queue is full tasks will be rejected. As the queue reduces, so does the number of active threads.

**-> FixedThreadPool vs cachedThreadPool**

The most important difference between a cached thread pool and a fixed thread pool in Java is that the cached thread pool has no upper limit on the number of threads it will spawn and use. Which one is preferred depends on what you want the scaling behavior to be like.

The main advantage of the cached thread pool is that threads will begin execution immediately even if you have an unanticipated large number of tasks to execute. For example, your business needs may increase over months or years, and your application may be moved to more powerful machines, and use of a cached thread pool will permit servicing the increased needs by using the increased available processing power without having to change the code. This can be an advantage since once the application has been in service for months or years, people may not remember the code well enough easily to identify the thread limit as a parameter that can be changed to improve performance.

The main advantage of the fixed thread pool is that the number of threads is more tightly controlled. This helps prevent other parts of the software installation - either within the application or in other applications - from being starved of processing power should this application receive a large number of tasks in a short period of time. In addition, the risk of running into the operating system threading limit is reduced, reducing the risks of software crashes that can result when a process needs to spawn a thread and is not able to do so.

* **What is difference between Executor.submit() and Executer.execute() methods**

**Submit() will take both callable instance and runnable instance, but execute will take only runnable. Thus submit is capable of retuning some value.**

* **Blocking Queue**
* A blocking queue is a queue that blocks when you try to dequeue from it and the queue is empty, or if you try to enqueue items to it and the queue is already full. A thread trying to dequeue from an empty queue is blocked until some other thread inserts an item into the queue. A thread trying to enqueue an item in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more items or clearing the queue completely.
* public class BlockingQueue {  
    
   private List queue = new LinkedList();  
   private int limit = 10;  
    
   public BlockingQueue(int limit){  
   this.limit = limit;  
   }  
    
    
   public synchronized void enqueue(Object item)  
   throws InterruptedException {  
   while(this.queue.size() == this.limit) {  
   wait();  
   }  
   if(this.queue.size() == 0) {  
   notifyAll();  
   }  
   this.queue.add(item);  
   }  
    
    
   public synchronized Object dequeue()  
   throws InterruptedException{  
   while(this.queue.size() == 0){  
   wait();  
   }  
   if(this.queue.size() == this.limit){  
   notifyAll();  
   }  
    
   return this.queue.remove(0);  
   }  
    
  }
* **Synchronizers**
* **Locks in java:** [**http://flex4java.blogspot.in/2015/02/lock-reentrantlock-reentrantreadwritelo.html**](http://flex4java.blogspot.in/2015/02/lock-reentrantlock-reentrantreadwritelo.html)
* **Java 8 thread features**
* **Multi-threading coding exercises**
* WAP to print even odd via 2 threads
* WAP to print nos from 1 to 10 in sequence using 2 threads
* There are three threads in a process.

The first thread prints 1 1 1 …, the second one prints 2 2 2 …, and the third one prints 3 3 3 … endlessly.

How do you schedule these three threads in order to print 1 2 3 1 2 3 …?

* Can threads in the same thread grp have different runnable interfaces

|  |  |
| --- | --- |
| * WAP to print nos from 1 to 200 using 2 or more threads. * WAP your own thread pool via executor framework * One task done by multiple threads * Multiple task done by multiple threads * WAP to create deadlock between 2 or more threads * There are two methods printA inside Class A and printB inside class B. printA is a static method and printB is a non-static method. Both the methods are synchronized. There exact 100 million threads fired on both printA and printB each. Which method execution will take less time? * There is a service which is run by a thread. Within that service, there are say 100 tasks which need to be executed sequentially(highlighted the work sequentially). How would you design your service, specifically what classes from Java framework would you use for this scenario? * Where could we use deadlock? The interviewer was very clear about his question. The use of deadlock. In which scenario could we use deadlock?   Soln: to test deadlock prevention code.  Allow me to be creative! Could we use it to enforce security? We design a system such that if certain sequence of operations are performed, there occurs a deadlock and the entities involved in the deadlock are terminated   * I have to create 5 threads where each thread has to perform the addition operation. Implemented a bounded queue:   Read:  If queue is empty, wait till it can return a value with time out  If another thread is reading from the queue then wait till that thread isdone  Remove the first element from the queue and return it  Do not block if a thread is writing into the queue  Write:  If queue is full, wait till one value is read with time out  If another thread is writing to the queue, wait till that thread is done  Write the element at the end of the queue  Do not block if a thread is reading from the queue  Thread1 - Add 1 to 10  Thread2 - Add 1 to 50  Thread3 - Add 5 to 15  Thread4 - Add 10 to 20  Thread5 - Add 15 to 20  What is the best way to accomplish this?  Also, I need 1 sec time delay between each addition operation.   * WAP to make sure main is the last thread to finish and third thread starts only when first one is dead. * Implement ReentrantLock using simple locks * Make use of an example to depict Singleton pattern. How would you make sure it works in Multithreaded environment. * What's the use of concurrency list in java? What are various locking mechanism in java? * need to implement a weather report functionality. user will provide the city name , need to return the weather report.   if weather station exists n functioning properly , will return the weather report of that station .  else ,  will return the nearest available weather station report.  interviewer looking for optimized manner.  looking for datastructures to stores the cities n algo to return the report.   * If two threads are incrementing a variable 100 times each without synchronization, what would be the possible min and maximum value. * Write a code for reader writer (multi threading) * How could you make sure that thread A ,B and C run sequentially without using join method?   [Say there are 3 array lists l1, l2 & l3 of same length. Thress threads accessing three lists. Say T1 -> l1, T2 ->l2 & T3 ->l3. It should print in the order say first element of 1st then first element of 2nd list and then first element of 3rd list. Then second element of 1st then second element of 2nd list and then second element of 3rd list.](https://www.careercup.com/question?id=16446668)   * - [rizwan.amd](https://www.careercup.com/user?id=16463662) March 26, 2013 in India | [Report Duplicate](https://www.careercup.com/reportduplicate?id=16446668) | [Flag](https://www.careercup.com/flagquestion?id=16446668)   [How do you detect deadlocks? What tools would you use? I said do “Kill -3 .<process id>, and analyse if anything is deadlocked.](https://www.careercup.com/question?id=16297665)  [Given four resources A1, A2, B1 , B2. Such that Thread T1 and Thread T2 operates on A1, A2 and B1, B2 respectively. How will you ensure the order of execution is A1-B1-A2-B2 ?](https://www.careercup.com/question?id=14755748)   * - [geekrocker](https://www.careercup.com/user?id=11999162) October 01, 2012 in India | [Repo](https://www.careercup.com/reportduplicate?id=14755748) * <https://www.careercup.com/question?id=12419684> * <https://www.careercup.com/question?id=12332725> * What is race condition |  |

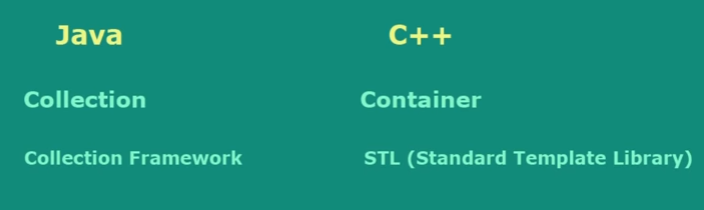
* **Restful webservice and HTTP methods**

[**https://www.restapitutorial.com/lessons/httpmethods.html**](https://www.restapitutorial.com/lessons/httpmethods.html)

* **Volative vs Atomic/CAS**

[**https://stackoverflow.com/questions/19744508/volatile-vs-atomic**](https://stackoverflow.com/questions/19744508/volatile-vs-atomic)

* **Reentrant locks in java**
* [**https://www.geeksforgeeks.org/reentrant-lock-java/**](https://www.geeksforgeeks.org/reentrant-lock-java/)
* [**https://stackoverflow.com/questions/11821801/why-use-a-reentrantlock-if-one-can-use-synchronizedthis**](https://stackoverflow.com/questions/11821801/why-use-a-reentrantlock-if-one-can-use-synchronizedthis)
* **Super and this, very imp get this in details**
* **Executor vs fork/join framework**
* **Count-down latch, cyclic barrier**
* **Fork-join vs work-stealing algo in java8**
* **Java-8**
* **invokeAll**
* **invokeAny**
* **CompletableFuture**
* **Future with timeout**
* **ScheduledExecutorService**
* **ReadWriteLock**
* **StampedLock**
* **ConcurrentMap**
* **Collections**
* **Why do we need collections**
* We normally declare variables like say int I = 10; if suppose we need 3 such variables we can say int I,j,n = 10; but what if we need 1000 such variables, in that case declaring 1000 such ints is not a partical solutions, here is were we need a collections object or any array. So that we can perform operations on these 1000 values by just using a single variable reference.
* Under the collections framework, Java programming language has given ready-made implementation of data-structure like hashtable and linkedList, unlike C where programmer had to write it yourself
* Normally that is that a framework does, it makes your life easy, it gives you thinks readymade e.g. struts, spring-batch, spring-integration, etc
* **Limitations of arrays:**
* Fixed in size
* Can only store homogenous elements
* Are not based on any data-structure, so no ready-made methods are available, like no ready-made sort or contains method is available, which are available in collections
* **Collection equivalent in C++**



* **Collection vs collections**

Collection, as its javadoc says is "The root interface in the collection hierarchy." This means that every single class implementing Collection in any form is part of the Java Collections Framework.

The Collections Framework is Java's native implementation of data structure classes (with implementation specific properties) which represent a group of objects which are somehow related to each other and thus can be called a collection.

Collections are merely an utility method class for doing certain operations, for example adding thread safety to your ArrayList instance by doing this:

List list = Collections.synchronizedList(new Arraylist());

The main difference in my opinion is that Collection is base interface which you may use in your code as a type for object (although I wouldn't directly recommend that) while Collections just provides useful operations for handling the collections.

* **There are a few basic operations you'll normally use with collections:**

**■** Add objects to the collection.

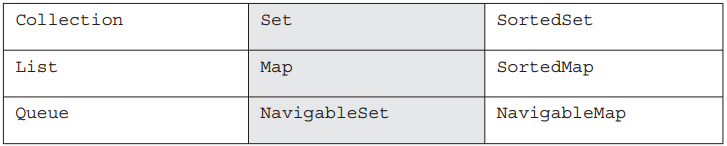
■ Remove objects from the collection.

■ Find out if an object (or group of objects) is in the collection.

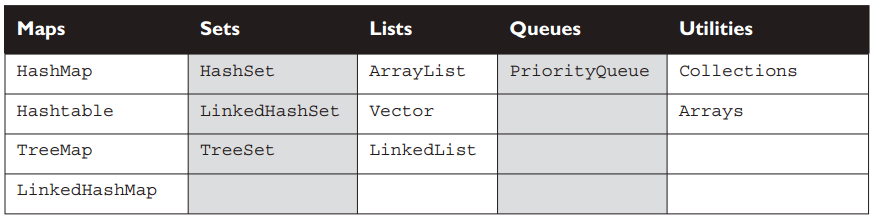
■ Retrieve an object from the collection (without removing it).

■ Iterate through the collection, looking at each element (object) one after another

* **Key Interfaces and Classes of the Collections Framework**



**The core concrete implementation classes you need to know for the exam are the following**

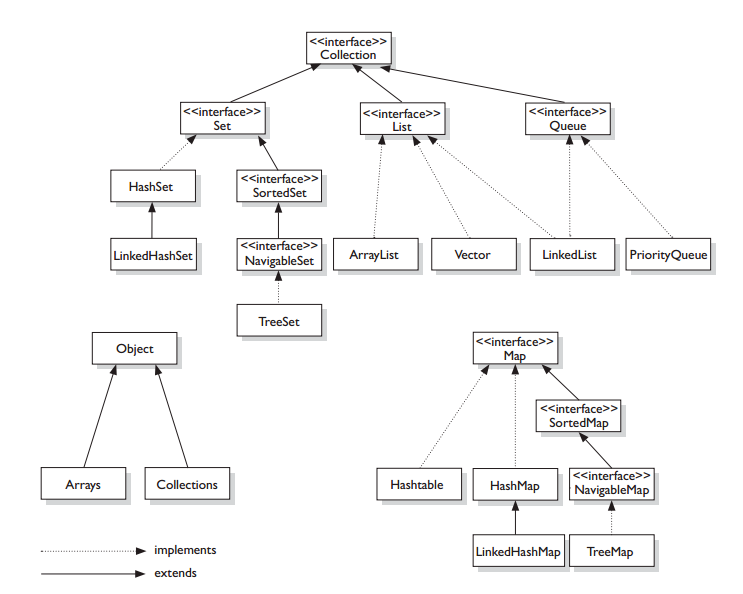


**■collection (lowercase c)**, which represents any of the data structures in which objects are stored and iterated over.

**■Collection (capital C)**, which is actually the java.util.Collection interface from which Set, List, and Queue extend. (That's right, extend, not implement. There are no direct implementations of Collection.)

**■Collections (capital Cand ends with s)** is the java.util.Collections class that holds a pile of static utility methods for use with collections.

* **Interfaces and classes hierarchy of collections framework**



**Collections come in four basic flavors:**

**■ Lists** Lists of things (classes that implement List).

**■ Sets** Unique things (classes that implement Set).

**■ Maps** Things with a unique ID (classes that implement Map).

**■ Queues** Things arranged by the order in which they are to be processed.

**But there are sub-flavors within those four flavors of collections:**

* **Sorted ⬄ unsorted**
* **Ordered ⬄ unordered**
* Sorted would imply ordering according to an implementation of Comparable or Comparator. Ordered would imply that it is following the insertion order or some other definition of order that is consistent and defined, but otherwise arbitrary.
* So a sorted list of strings would be sorted according to the String.compareTo method. A list might contain a list of strings inserted in arbitrary order, but that order will always remain the same.
* Of course there are methods on the Collections class to sort a list.

**Most commonly, the sort order used is something called the natural order. What does that mean?**

You know how to sort alphabetically—A comes before B, F comes before G, and so on. For a collection of String objects, then, the natural order is alphabetical. For Integer objects, the natural order is by numeric value—1 before 2, and so on. And for Foo objects, the natural order is…um…we don't know. There is no natural order for Foo unless or until the Foo developer provides one, through an interface (Comparable)that defines how instances of a class can be compared to one another (does instance a come before b, or does instance b come before a?). If the developer decides that Foo objects should be compared using the value of some instance variable (let's say there's one called bar), then a sorted collection will order the

Foo objects according to the rules in the Foo class for how to use the bar instance variable to determine the order. Of course, the Foo class might also inherit a natural order from a superclass rather than define its own order, in some cases. Aside from natural order as specified by the Comparable interface, it's also possible to define other, different sort orders using another interface: Comparator.

**List Interface:** it is the child interface of collection interface,

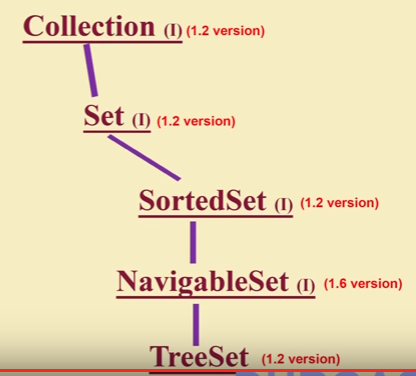
* it supports random access
* maintains insertion order,
* allows duplicates
* ArrayList, LinkedList and Vector are the direct implementations of list interface.
* Vector and its sub-class Stack are legacy classes which are from version 1.0 of java. Rest all were added in ver2.0
* **set Interface:**
* does not allow duplicates
* insertion order is not preserved
* hashSet is the direct implementation class and linkedHashSet is its direct sub-class
* When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.

**TreeSet:**

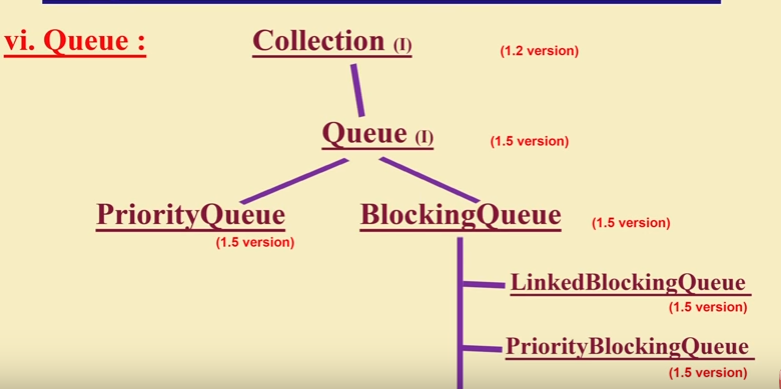
* The TreeSet is one of two sorted collections (the other being TreeMap).
* It uses a Red-Black tree structure (but you knew that), and guarantees that the elements will be in ascending order, according to natural order. Optionally, you can construct a TreeSet with a constructor that lets you give the collection your own rules for what the order should be (rather than relying on the ordering defined by the elements' class) by using a Comparable or Comparator. As of Java 6, TreeSet implements NavigableSet.

**List vs set**

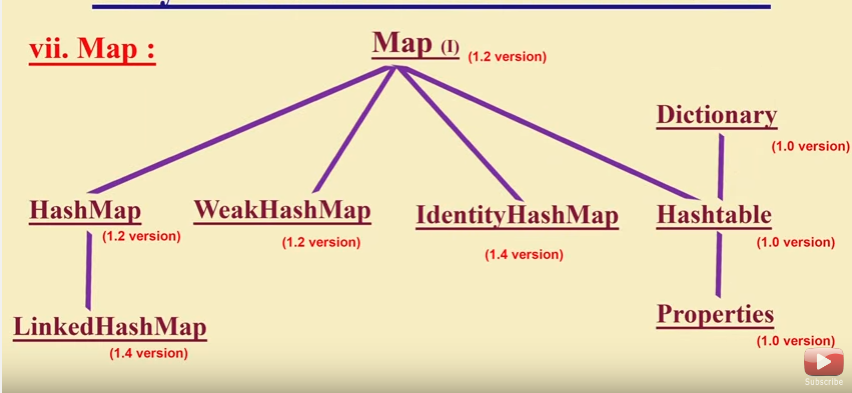
* List maintains insertion order, set does not
* List allows duplicates, set does not

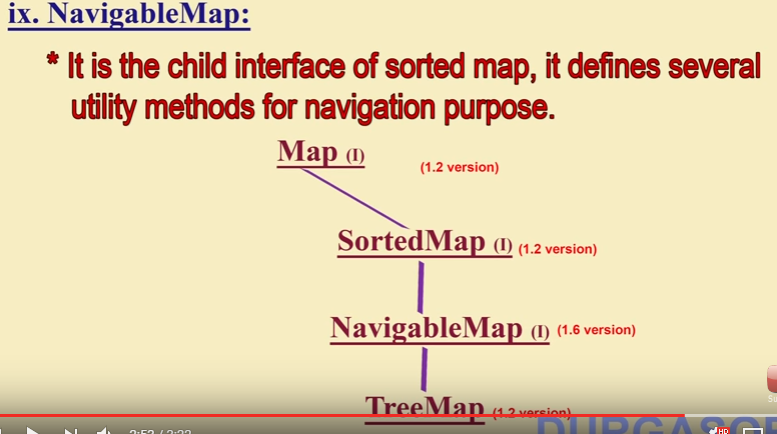


* **Queue:**

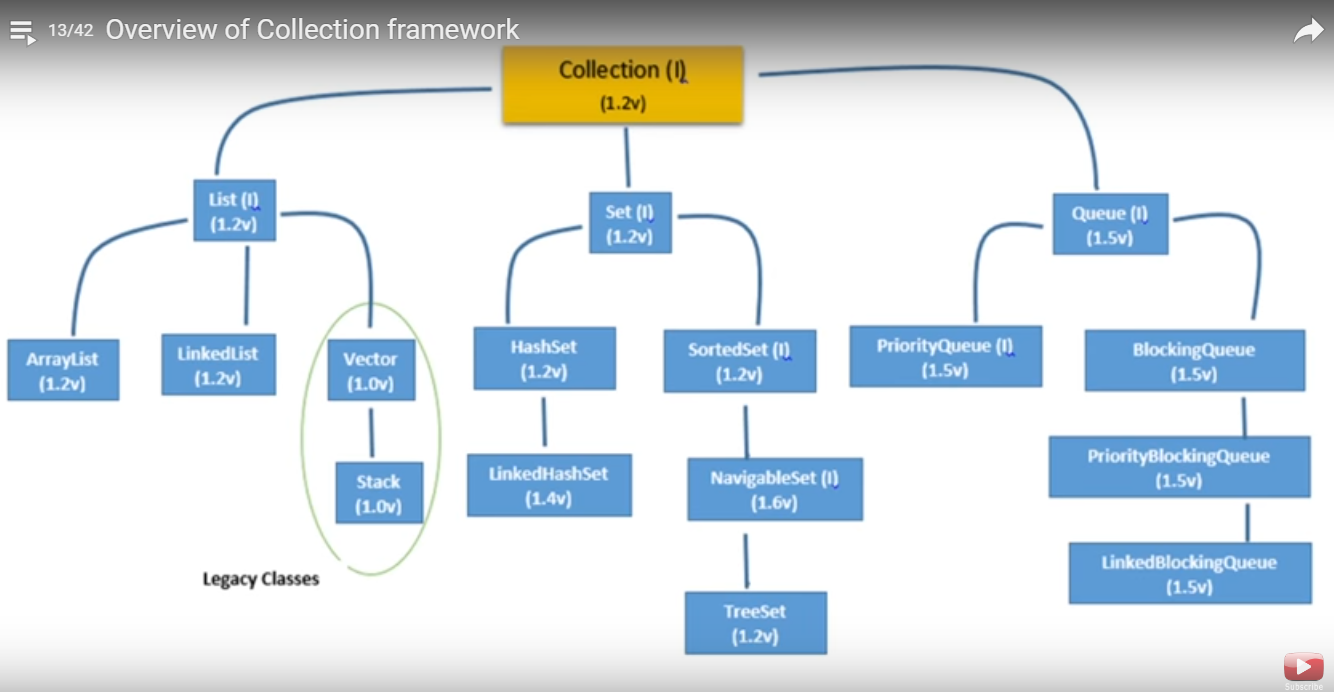


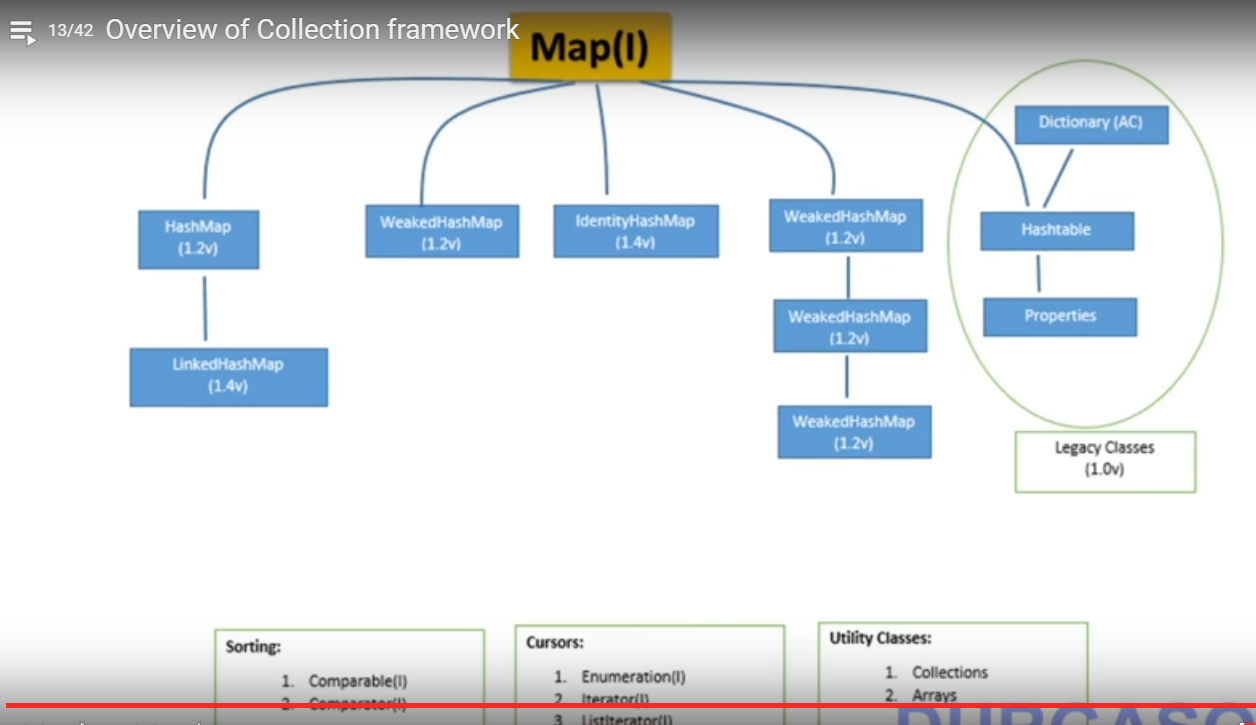
* **Map:** The Map implementations let you do things like search for a value based on the key, ask for a collection of just the values, or ask for a collection of just the keys. Like Sets, Maps rely on the equals()method to determine whether two keys are the same or different.





* **Vim:** [**https://www.youtube.com/watch?v=v9zg9g\_FbJY&index=13&list=PLd3UqWTnYXOkVR3OR9UZGyEt9RFUbaTMZ**](https://www.youtube.com/watch?v=v9zg9g_FbJY&index=13&list=PLd3UqWTnYXOkVR3OR9UZGyEt9RFUbaTMZ)





* **Every collection class implements serializable and clonable interfaces**
* **ArrayList:**
* Build on resizable array standard data structure
* Allows duplicate elements
* Insertion order is preserved
* Heterogenous objects are allowed,
* Null insertion is allowed
* Expect for treeSet and TreeMap all other collection object support heterogenoues elements, because TreeMap and TreeSet are sorted collections so to sort object we need to determine the order by comparing objects, they both the objects need to be of the same type.
* Default capacity is 10
* Resize capcity is (CC \* 3/2) + 1 here cc is current capacity i.e. 10
* From java 7 onwards, **int newCapacity = oldCapacity + (oldCapacity >> 1);**
* Arraylist and vector implements randomAccess interface it is a marker interface
* Best use-case is when the common operation is retrieval as it uses randomAccess interface
* Frequent inserts and deletions in the center of the list are slow in ArrayList, as lot of shit operations are required.
* **ArrayList vs Vector**

|  |  |
| --- | --- |
| **Arraylist** | **Vector** |
| No methods are synchronized | All public methods are synchronized |
| Not a thread-safe | It is thread-safe |
| Arraylist grows by cc\*3/2+1.  Default capacity is 10 | Grows by 2\* cc by default. There is also a constructor where u can specify the inc size. This is not in ArrayList  Default capacity is 10 |
| Added in java 1.2 ver | This is legacy class from java 1.0 ver |

* **Sync version of arraylist**

**List syncList = Collections.synchronisedList(ArrayList li);**

**Set syncSet = Collections.synchronisedSet(HashSet s);**

**Map syncMap = Collections.synchronisedMap(Map m);**

* **LinkedList**
* Based on double link list data structure
* Insertion order is preserved
* Null values are allowed
* Duplicates are allowed
* Heterogeneous objects are allowed
* Best use case is in case if insertion and deletions are in the middle of the list
* Worst choice if retierival is the main operation
* xxxFirst() and xxxLast() are 6 imp linked list specific methods
* there is no concept of default initial capacity or size of linkedList, because here a new node is created as and when you want to add data.
* Retierival operation is slow, as the entire list needs to be tranversed,
* It does not implement random access interface
* **ArrayList vs LinkedList**

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| Based on resizable or growable array data strcture | Based on double linked list data structure |
| * Best used when frequent operation is retrieval | * Best used when frequent operations in insertion and deletion from the middle |
| Worst in case of insertion and deletion from the middle | Worst in case of retierival operational as the whole list needs to be traversed |
| Implements Random Access interface | Does not implements random access interface |
| Has a default init size of 10 | No concept of default size as nodes are created on demand as data is inserted or removed |

* **Why no load factor in List, but in Maps**

Load factor is specific to hashed collections. Lists are unlikely to have a concept of load factor.

For an ArrayList this is always 1 (it won't grow until the size reaches the capacity) However the typical load factor of a HashMap is 0.75f which means the capacity will grow when the size reaches 75% of capacity.

For ArrayList, every time you put an element into it, it will check if the nested array needs to be enlarge its size. If yes, generally, its size will grow with:

newCapacity = oldCapacity + (oldCapacity >> 1);

For some special case, for example, add many or huge number of elements, things will be different. Please refer grow(int minCapacity) function in java.util.ArrayList source code.

Regarding Vector, generally, its size will grow with:

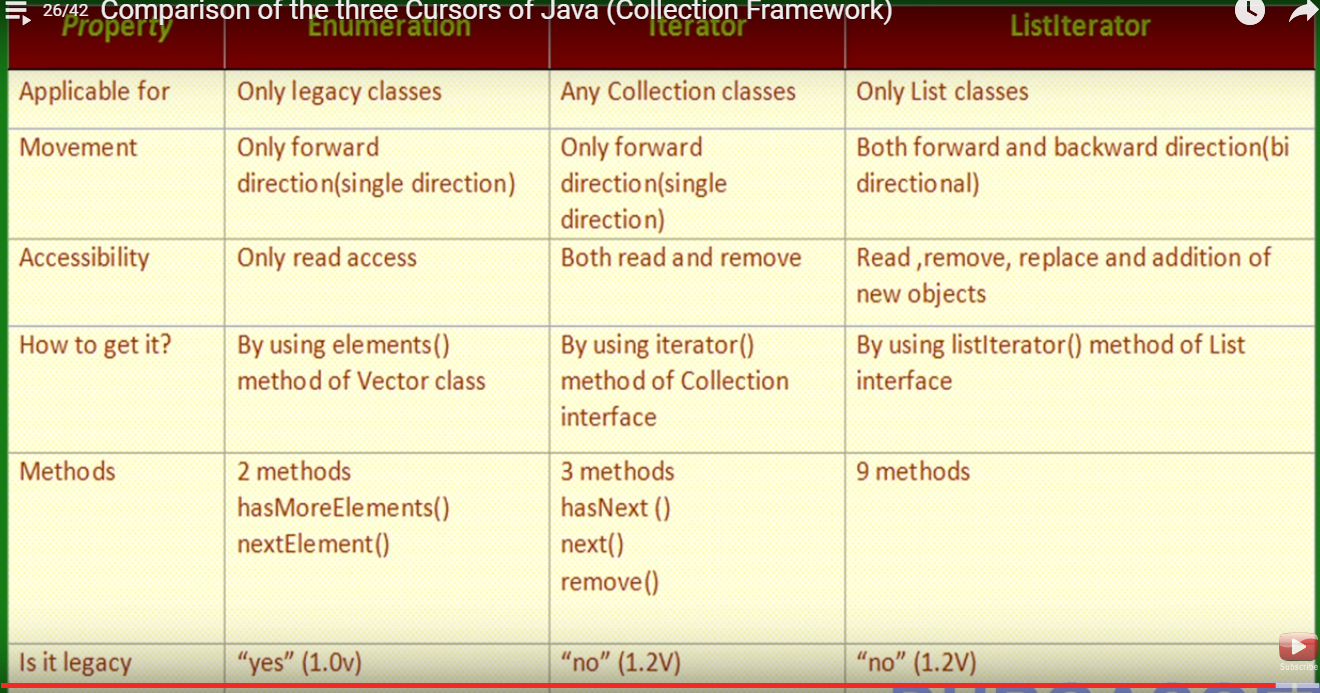
newCapacity = oldCapacity + ((capacityIncrement > 0) ?

capacityIncrement : oldCapacity);

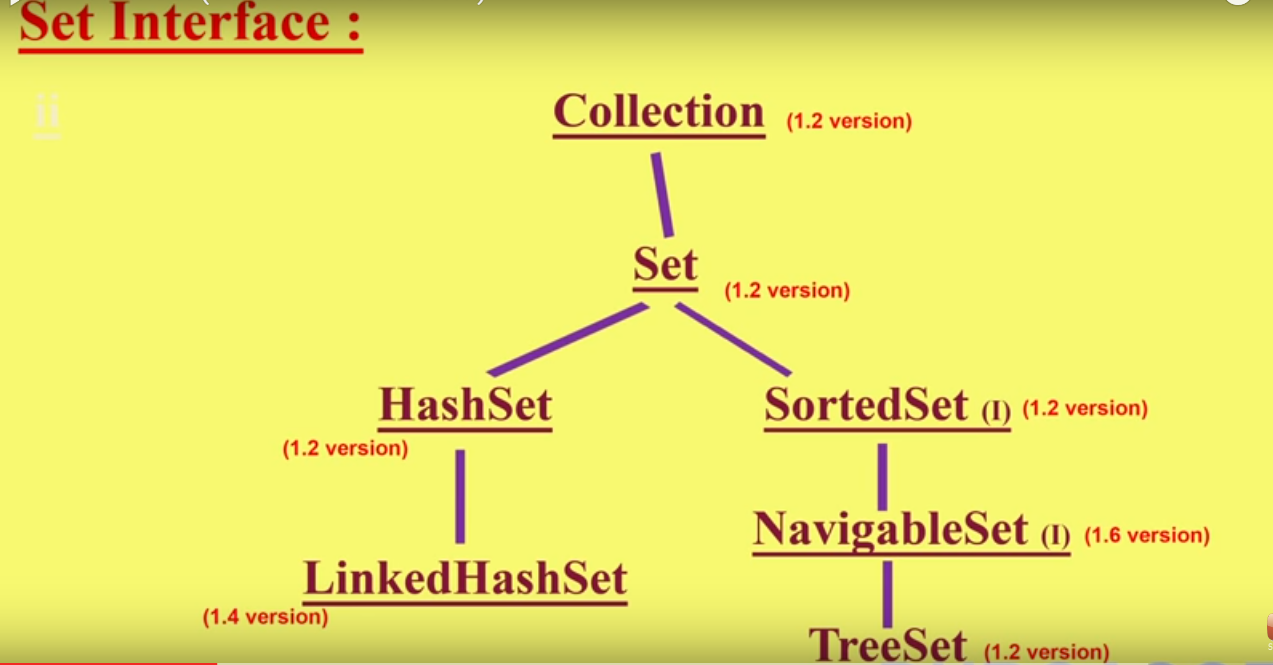
For some special cases, please refer grow(int minCapacity) in java.util.Vector.

|  |  |
| --- | --- |
|  | - Maps as such don't have a loadFactor - only implementations based on some kind of HashMap do have it (e.g. there's no loadFactor on a TreeMap).  Why is that?  A HashMap contains a number of "buckets" and when adding or retrieving an entry you take the key's hashcode and calculate which bucket you have to put it in or retrieve it from. Based on the quality of the hash-implementation two distinct objects might end up in the same bucket. When this happens the hashmap starts a Linked list that you have to go through when retrieving the entry.  The HashMap and List differ in some important points:   * The capacity of the HashMap does not say how many elemnts could be stored in it, it's the number of buckets. In theory you could store more than capacity entries in a HashMap. * Too much items ending up in the same bucket are bad for the HashMap's performance. If you have fewer buckets you have in relation to the number of entries you'll increase the number of such collisions. Enter the loadFactor: if things get "too tight" and you fear you'll get too much collisions you start to grow the number of buckets - even if there are still some empty ones left. |

* **Cursors in Java Collections**
* **Enumeration**
* **Lterator**
* **listIterator**



* **Set Interface**
* Child interface of collection interface
* Adds no new methods
* Duplicates are not allowed
* Insertion order is not preserved
* **Implementation classes are**



* **hashSet**
* based on hashtable data structure
* duplicates are not allowed, it you add duplicate the add() method returns false
* best use case in case of fast searching
* insertion order is not preserved, elements are stored/inserted based on hashCodes of objects
* heterogenous objects are allowed
* null objects are allowed
* Constructors
* Intial capacity is 16.
* Load factory is .75
* You can speicify your won init cap and custom load factor
* **LinkedHashSet**
* Child class to hashSet
* Insertion order is preserved
* It uses hashTablea and a doubly linkedlist as underlying data structure
* Introduced in java 1.4 ver
* Its is best choice while developing cache based applications where insertion order must be preserved and duplicates are not allowed
* **TreeSet**
* Underlying data structure is balanced tree
* Insertion order is preserved
* Sorted set
* By default, default natural sorting order is used,
* All element must be homogenous
* If you want to insert homogenous objects, they need to pass your implementation of comparator interface
* In case if the inserted elements are primitives or string or any class that implements the comparable interface
* In case if custom objects or classes that do not implement comparable are inserted that Class Cast exception is thrown
* Null insert will result in null pointer exception
* 1st element as null can be inserted, but subsequent elements will result in null pointer exception
* **Comparable interface**
* Present in java.lang package
* Contains only one method compareTo ()
* public int obj1.compareTo(obj2);
* Returns –ve if obj1 comes before obj2
* Return +ve if obj1 comes after obj2
* Returns 0 if obj1 equals obj2, here obj1 is the current object and obj2 is the object already in the set
* If we are not satisfied with the default natural sorting order or if the default natural sorting order is not present than we can implements our own custom sorting order using comparator
* Comparable is ment for default natural sorting order, where as compartor is ment for customized sorting order
* [Collections.sort implementation](https://stackoverflow.com/questions/14322585/collections-sort-implementation)

public static <T extends Comparable<? super T>> void sort(List<T> list) {

Object[] a = list.toArray();

Arrays.sort(a);

ListIterator<T> i = list.listIterator();

for (int j=0; j<a.length; j++) {

i.next();

i.set((T)a[j]);

}

}

* **In generics <T extends comparable> means T implements comparable**
* **here the syntax** T extends Comparable<? super T> means, to fulfill <? Super T> T must implement comparable and the type parameter to comparable method should can be any super class of T.
* **e.g.**   it means that a Student class can implement Comparable<Person>, where Student is a subclass of Person

public class Person {}

public class Student extends Person implements Comparable<Person> {

@Override public int compareTo(Person that) {

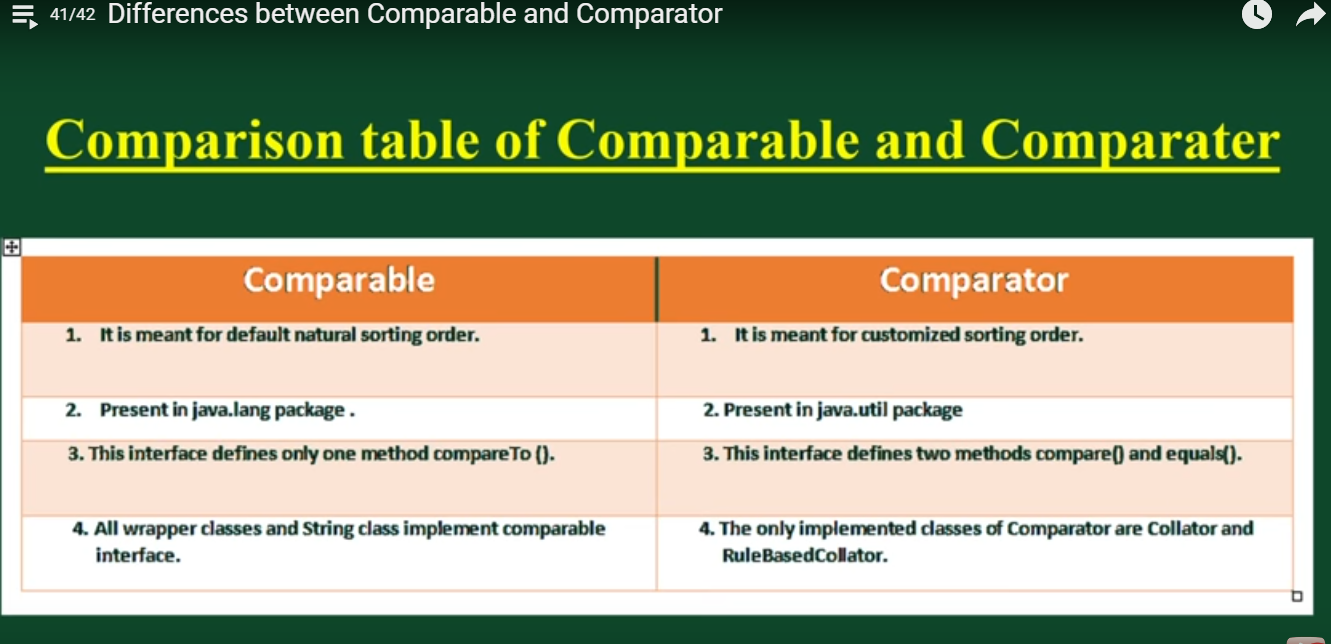
// ...

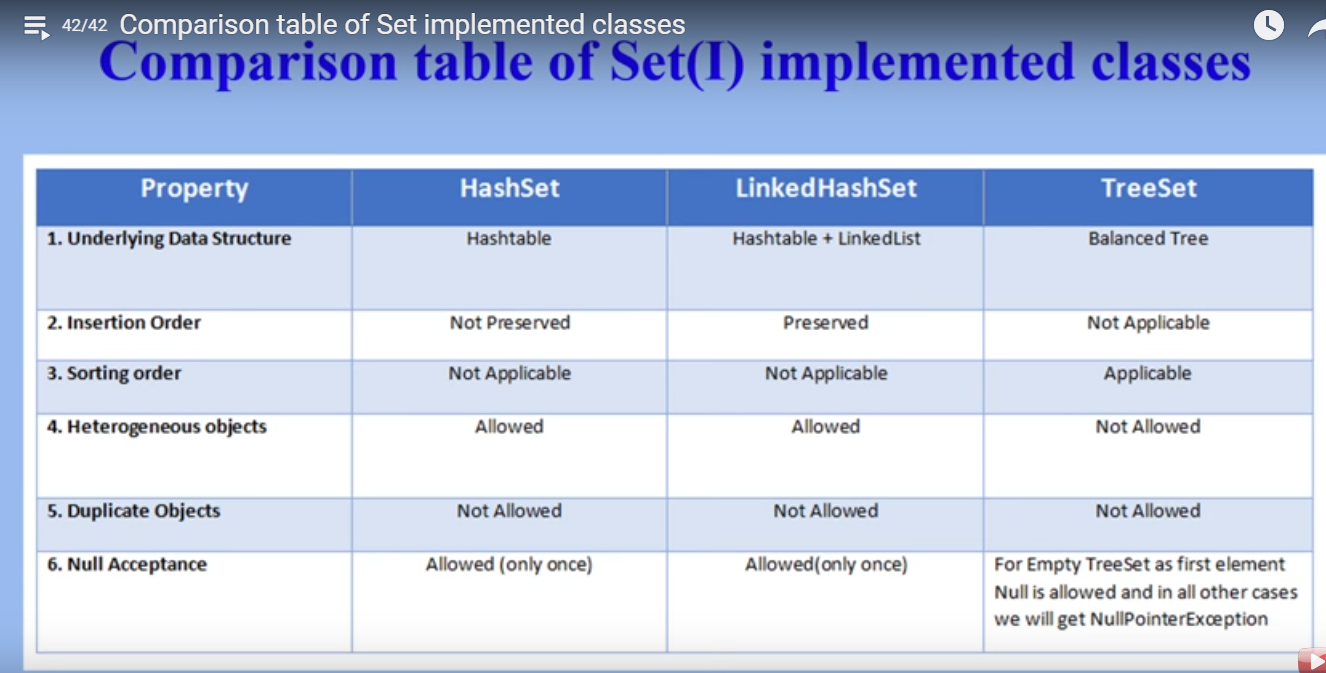
}

}

* **Comparator interface**
* Used to provide customized sorting order
* It is in java.util package
* It has 2 methods compare() and equals()
* Public int compare(obj1, obj2)
* Public boolean equals()
* It is mandatory to provide implementation for compare() method
* Equals() method is optional as it is by default avaible in all class via Object class inheritance

* **Comparator vs Comparable when to use what**
* if you are a API developer, then you should provide a default natural soring order by default via a comparable e.g. natural sorting based on empid for employee object
* if you are the client side that uses your API, if are not stisified with the default sorting, then they you need to provide your own custom sort order via comparator
* this is the similar approach used by java or Sun or Oracle peoepl, they do not provide any comparator, but have provided enough default natural sorting via comparable in classes like String.





* **Why is map interface not part of collections interface, or subclass of it.**
* collection and its subclass interfaces deal with individual objects.
* But map deals with key value pair type of objects so it’s a different interface hierarchy altogether.
* **Map interface**
* Not a child interface of collection
* If we want to represent a group of objects are key values pairs then we should go for map
* Both keys are values are objects only
* Duplicate keys are not allowed
* Duplicate values are allowed
* **Entry:**
* Each key value pair is called an Entry
* Hence map is considered as a collection of entry objects
* It’s a nested interface inside map interface
* It has methods like getKey(), getValue(), setValue(),
* getHashCode();
* **Methods in map interface**
* Object o = put(Object key, Object value);
* Returns null if key not present in map.
* Return old value if key was already present, in this case the old value will be replaced with the new value.
* Map.putAll(map m) will return void
* **Hash map**
* Underlying data structure in hash table
* Insertion order is not preserved and it is based on hashcode of keys
* Duplicate keys are not allowed
* But values can be duplicated
* Heterogenous objects are allowed for both key and value
* Null is allowed for key, but only once
* Null values are allowed, any number of times
* Hashmap implements serializable, clonnable interfaces
* Best choice in case of search specific tasks
* HashMap m = new HashMap(), creates an empty hashmap object with default size 16, and deaful init fill ratio/load-factor .75
* HashMap m = new HashMap(int capacity); HashMap m = new HashMap(int capa, float fillratio) ; these are also proviced to specify custom default capacity and custion load-factor.
* **Hashmap vs HashTable**

|  |  |
| --- | --- |
| **Hashmap** | **HashTable** |
| **Methods are no synchronized** | **Methods are synchronized** |
| **Not a thread-safe class** | **Thread-safe class, as only one thread is allowed to operate at a time** |
| **Relative performance is high, because threads are not required to wait on hashmap object** | **Relative performance is low** |
| **\* Null is allowed for both key and value** | **Null is not allowed for either key nor value** |
| **\* Introduce in ver 1.2** | **Legacy class present in ver 1.0** |

* **snchrnozied version of hashMapObject**
* **collections.synchronizedMap(m);**
* **concurrentHashMap**
* **linkedHashMap**
* **child class of hashmap**
* **exactly same as hashmap (including methods n constructors) except the following differences**
* **underlying data-strucurur is hashtable and linkedList**

|  |  |
| --- | --- |
| **Hashmap** | **linkedHashMap** |
| **Underlying data-structure is hashtable** | **Underlying data-sturcure is combition of linkedlist and hastable** |
| **Insertion order is not preserved and its based on hashcode of keys** | **Insertion order is preserved** |
| **Introduced in ver 1.2** | **Introduced in ver1.4** |

* **linkedHashSet and linkedHashMap are commonly used for developing cache-based applications**
* **Identityhashmap**
* **It is exactly same as hashmap (including methods n constructors)**
* **Differences are as below**
* **In case of normal hashmap jvm will use .equals() method to identify duplicate keys, which is ment for content comparison**
* **But in case of identityHashMap, jvm will use == operator to identify duplicate keys which is meant for reference comparison (address comparison)**
* **E.g.**

**Hashmap m = new HashMap();**

**Int i1 = 10;**

**Int i2 = 10;**

**m.put(i1, “abc”);**

**m.put(i2, “xyz”);**

**sop(m) // op: {10:xyz}, here i1 and i2 are duplicate keys because i1.equals(i2) returns true;**

* **If we replace hashmap in above e.g. in IdentidyHashMap then i1 and i2 are not duplicate keys because i1==i2 returns false;**
* **So o/p will be {10=abc, 10=xyz}**
* **when to use IdentityHashMap**
* **WeakHashMap**
* **It is exactly same as hashmap, except below difference**
* **In case of hashmap even though object doesn’t have reference it is not eligible for GC, if it is associated with hashmap i.e. hashmap dominates GC**
* **In case of weakHashMap, if object doesn’t contain any references it is eligible for GC even though object associated with weakHashMap i.e. GC dominates weakHashMap**
* **E.g.**

**HashMap m = new HashMap();**

**Temp t = new Temp();**

**m.put(t,”abc”);**

**Sop(m) // op: {t=”abc”}**

**t = null;**

**System.gc();**

**Thread.sleep(5);**

**Sop(m) // op: {t=”abc”}**

* **If we replace the above with WeakHashMap, op will be**

**WeakHashMap m = new WeakHashMap();**

**Temp t = new Temp();**

**m.put(t,”abc”);**

**Sop(m) // op: {t=”abc”}**

**t = null;**

**System.gc();**

**Thread.sleep(5);**

**Sop(m) // op: { } //temp object is GCed here…**

* **When to use weakHashMap**

Another common source of memory leaks is caches. Once you put an object reference into a cache, it’s easy to forget that it’s there and leave it in the cache long after it becomes irrelevant. There are several solutions to this problem. If you’re lucky enough to implement a cache for which an entry is relevant exactly so long as there are references to its key outside of the cache, represent the cache as a **WeakHashMap**; entries will be removed automatically after they become obsolete. Remember that **WeakHashMap** is useful only if the desired lifetime of cache entries is determined by external references to the key, not the value.

* **SortedMap**
* **It is child interface of map**
* **If we want to represent group of key values pairs, according to some sorting order of keys, then we should go for sorted map.**
* **Sorting is based on the key, but not based on value**
* **sortedMap defines the below methods**

**firstKey(), lastKey(), headmap(), tail(), subMap(),**

* **Object c = comparator(); returns null if default natural sorting order is used, returns custom comparator object if custom sorting is defined**
* **TreeMap**
* **Underlying data structure is red-black tree**
* **Insertion order is not preserved**
* **All elements will be inserted according to some sorting order of keys**
* **Duplicate keys are not allowed, but values can be duplicated**
* **Heterogeneous object insertions is not allowed for keys, if we depend on default natural sorting order**
* **If we define our own comparator, then heterogeneous key objects can be used**
* **For non empty treemap if we are inserting an entry with null key then we will get a NullPointerException**
* **For empty treemap as the 1st entry with null key is allowed, but after inserting that entry, if we try to insert any other entry then we will get runtime exception saying Null Pointer Execption**
* **The above null acceptance rule applicable until java 1.6 version only from 1.7 ver onwards null is not allowed for key**
* **There are no restrictions for null values**
* **Hashtable**
* **Underlying data structure is hash table**
* **Insertion order is not preserved n its based on hash code of keys**
* **Duplicate keys are not allowed, duplicates values are allowed**
* **Heterogenous objects are allowed for both keys and values**
* **Null is not allowed for both keys and values**
* **Its is thread-safe as most of the methods are synchronized**
* **Implements serializable, clonnable but not randomAccess**
* **Best used in case of search operations**
* **Default init capit is 11, load factor is .75**
* **properties**
* **Queue**
* **Is child interface of collection**
* **1.5 ver enhancments**
* **If we want to represent a group of individual objects prior to processing then we should go for Queue**
* **For e.g. temple queue, vip drashan etc**
* **Usually queue follows FIFO order by default, but we can provide a custom order**
* **From 1.5 ver onwards linkedList class also implements queue interface**
* **linkedList based implementation of queues always follows FIFO order**
* **Queue interface specific methods**
* **enQueue**
* **deQueue**
* **offer(object o)**
* **Object o = poll()**
* **Similar to poll is remove**
* **PriorityQueue**
* **If we want to represent a group of individual objects prior to processing according to some priority, then we should go for priority queue**
* **The priority can be either default natural sorting order or customized sorting order defined by comparator**
* **Insertion order is not preserved, and its based on some proritity**
* **Duplicate objects are not allowed**
* **If we are depending on default natural sorting order compalsary the objects should be homogenous and comparable otherwise we will get ClassCasteExection at runtime**
* **If we are defining our own sorting by comparator then objects need not be homogenous and comparable**
* **Null is not allowed**
* **ProorityQueue q = new ProdityQueue();**

**Default init capacity is 11 and default natural sorting by default**

* **Navigable Set and Navigable Map**
* **These were added as part of ver 1.6 enchanments**
* **It’s a child interface to sorted set**
* **It defines several methods for navigation purposes**
* **Sorting elements of a list**
* **List provides no sorting techniq by default**
* **So collections util class provides 2 methods**
* **We use Collections.sort(list l);**

**It will by default provide default natural sorting order**

**In case of custom objects, we need to provide a sort order by comparator interface compare method, the corresponding method to use is Collections.sort(lit l, comparator c);**

* **Searching element in a list**
* **Internally the collections**
* **Collections class defines the below binary search methods**
* **Public static int binarySearch(list l, object target)**
* **If the list is sorted according to default natural sorting order, we can use above method**
* **Public static int binarySearch(list l, object targer, comparator c);**
* **We have to use this method if the list needs to be sorted according to customized sorting order**
* **Conclusions**
* **The above search methods will internally use binary serach algo, for which we need to 1st sort the list.**
* **Successful search returns index**
* **Unsuccessful search returns insertion point**
* **Insertion point is the index where the target object can be placed in the sorted list**
* **Compulsory list should be sorted otherwise we will get unpredictable results**

* **concurrentHashMap**
* **copyonwirteArrayList**
* **copyonwriteArraySet**
* **synchronizedMap vs ConcurrentHashMap**

We can achieve thread safety by using both ConcurrentHashMap and synchronisedHashmap. But there is a lot of difference if you look at their architecture.

1. **synchronisedHashmap**

It will maintain the lock at the object level. So if you want to perform any operation like put/get then you have to acquire the lock first. At the same time, other threads are not allowed to perform any operation. So at a time, only one thread can operate on this. So the waiting time will increase here. We can say that performance is relatively low when you comparing with ConcurrentHashMap.

1. **ConcurrentHashMap**

It will maintain the lock at segment level. It has 16 segments and maintains the concurrency level as 16 by default. So at a time, 16 threads can be able to operate on ConcurrentHashMap. Moreover, read operation doesn't require a lock. So any number of threads can perform a get operation on it.

If thread1 wants to perform put operation in segment 2 and thread2 wants to perform put operation on segment 4 then it is allowed here. Means, 16 threads can perform update(put/delete) operation on ConcurrentHashMap at a time.

So that the waiting time will be less here. Hence the performance is relatively better than synchronisedHashmap.

* **hashCode and Equals**

## hashCode()

The hashCode() method of objects is used when you insert them into a HashTable, HashMap or HashSet. If you do not know the theory of how a hashtable works internally, you can read about [**hastables on Wikipedia.org**](http://en.wikipedia.org/wiki/Hashtable).

When inserting an object into a hastable you use a key. The hash code of this key is calculated, and used to determine where to **store** the object internally. When you need to lookup an object in a hashtable you also use a key. The hash code of this key is calculated and used to determine where to **search** for the object.

The hash code only points to a certain "area" (or list, bucket etc) internally. Since different key objects could potentially have the same hash code, the hash code itself is no guarantee that the right key is found. The hashtable then iterates this area (all keys with the same hash code) and uses the key's equals()method to find the right key. Once the right key is found, the object stored for that key is returned.

So, as you can see, a combination of the hashCode() and equals() methods are used when storing and when looking up objects in a hashtable.

Here are two rules that are good to know about implementing the hashCode() method in your own classes, if the hashtables in the Java Collections API are to work correctly:

1. If object1 and object2 are equal according to their equals() method, they must also have the same hash code.
2. If object1 and object2 have the same hash code, they do NOT have to be equal too.

In shorter words:

1. If equal, then same hash codes too.
2. Same hash codes no guarantee of being equal

* **How to generate/override hascode and equals**

For JDK 7 and above, you can use the new Objects class to generate the equals

and hash code values.

* import java.util.Objects;
* public class User {
* private String name;
* private int age;
* private String passport;
* //getters and setters, constructor
* @Override
* public boolean equals(Object o) {
* if (o == this) return true;
* if (!(o instanceof User)) {
* return false;
* }
* User user = (User) o;
* return age == user.age &&
* Objects.equals(name, user.name) &&
* Objects.equals(passport, user.passport);
* }
* @Override
* public int hashCode() {
* return Objects.hash(name, age, passport);
* }
* }

## 3. Apache Commons Lang

Alternatively, you can use the Apache Commons Lang EqualsBuilder and HashCodeBuilder function.

* **You must override hashCode() in every class that overrides equals(). Failure to do so will result in a violation of the general contract for Object.hashCode(), which will prevent your class from functioning properly in conjunction with all hash-based collections, including HashMap, HashSet, and Hashtable.**
* **The best use case of hashmap is that it retivees the element to be searched in constant time i.e O(n).**
* **This is only possible if you have defined your hascode perfectly, because**
* **a perfectly defined hascode will minimize collisions, and not allow duplicate entries.**
* **If you have 2 emp objects say**

**Emp e1 = new Emp(1, ‘abc’);**

**Emp e2 = new Emp(1,’abc’);**

* **Here even-though the above 2 objects are same, the default Object.equals() method will return false, because it will consider the reference equality i.e. e1==e2, which happens to be different and thereby it will return false.**
* **Here is where we need to override the equals method and tell it explicity to consider the content of the objects for comparison and not only the hashcode.**
* **So now when you call put(e1,somValue), the hashmap will first find the hashcode asso with e1 and insert it.**
* **Next when we will call put(e2,somValue), the hascode of e2 will be called, and lets say it falls on the same bucket as that of e1, then the equals will be called which will fail, there by calling e2 to be duplicate,**
* **Now if we only override the equals without override the hashcode, then e2 will land in some other bucket, thereby allowing duplicate entries in the hashtable.**
* **Thus we need to override the hashcode to because only when we will get the same hashcode for equals objects we can prevent duplicate insertion**
* **from Effective Java, by Joshua Bloch**
* **memory poolmxbeanp**
* **equals and hashCodex**
* **System.arraycopy()**

* [**What is the need of  wrapper classes?**](https://www.quora.com/What-is-the-need-of-wrapper-classes)

Java draws a strong distinction between objects and primitive types. You can't assign an **int** (or other primitive) to an Object-valued field.  
  
Fundamentally, a Collection class is implemented in terms of objects. A simple ArrayList class might be implemented like this:

1. **class** ArrayList {
2. Object[] data = **new** Object[10];
3. **int** count = 0;
5. **void** add(Object o) {
6. **if**(count < data.length)
7. data[count++] = o;
8. **else** {
9. // allocate more space and add
10. }
11. }

You can't pass an int in there because an **int** isn't an Object. You can, however make an Integer, which is an object, one that happens to hold an **int** value.   
  
Why does Java draw such a strong distinction between objects and primitives? Effectively, it's a matter of the way the JVM is implemented. An object is a pointer, which holds type information and a chunk of memory, while an **int** is just four bytes. The machine uses very different operations to work on the two. So, for example, an Object can be null; an **int** can't. The real details of the implementation are far, far more complicated than that, but that's the basic reason.

Later versions of Java try to hide the difference somewhat, automatically wrapping an **int** as an Integer whenever it can think to. But underneath, the distinction is still there.

Ans2:

Collections in the first place, for example,List<Integer>, you cannot use primitive int here. Actually any generic class / interface that can work with different object types like

public interface Callable<V> {

V call() throws Exception;

}

Note that wrapping is best done using not new Integer(i) but Integer.valueOf(i) the latter will try to use cache. Unwrapping is done as Integer.intValue(). These wrapping / unwrapping of primitives are so typical operations that Java 5 introduced autoboxing / unboxing

List<Integer> list = new ArrayList<>();

list.add(1);

int i = list.get(0);

this code is automatically converted by Java compiler into

list.add(Integer.valueIf(1));

int i = list.get(0).intValue();

Wrapper classes are designed to add more functionality to the primitive types, so that they are compatible with generic code, using the Collection Framework, and many other benefits. However, they are not mean't to **replace** primitive types.

So, you should use wrappers only when necessary, such as when dealing with generics, because creating an object adds substantial overheads to your program. So, in normal cases, you should stick to primitives.

* **What will happen if we increase the size of the array at runtime.**

ArrayList, vector and all map allocates a default initial memory of 10 objects.

In case of ArrayList, when a request of 11th object comes it increases to 1.5 times of present size i.e. 10 --&gt; 15 --&gt; 22...

In case of Vector, when a request of 11th object comes it increases to 3 times of present size i.e. 10 --&gt; 20 --&gt; 40...

In case of Map, when it is .75 (equal to load factor) full object comes it increases to 2.0 times of present size i.e. 10 --&gt; 15 --&gt; 22...

NOTE: - in case of Vector instead of increment to double you can define increment size. Then it will increase only that no of objects only not double.

* **Hash Map internal working**

some other ds e.g arrays, linke list

than maps

O(n/k) for insertion updation and deletions

associate arrays

differece between normal vs asso arrays

hash function

interall structru of hash table

table, bucket and ptr to list of entry objects

static inner class Entry<K,V>

hashMap constructors

put(key, value)

if(table is null)

inflateTable() // create new table

if(key==null)

insert for null keys

int hash = hash(key);

int bucketIndex = indexFor(hash, table.length);

//iterate over the chain of / list of entry objects corresponding to that bucket

if(key is same, hashCode is same, equalsReturnsTrue)

update with currently value and return

else

create new entry

addEntry() is called

it mainly checks if resizing is needed.

addEntry

inturn calls createEntry

corresponding indexes are updated i.e. size and all,

get(key);

if(key is null)

getForNullKey is called.

else

getEntry(key) is called,

if(table is null)

return null;

if(key is null)

return 0th index as all null keys are mapped to 0th entry;

int hash = hash(key);

int = indexFor(hash);

//iterate over the chain of entry objects

compare hashValue of every Entry object with the input

compare equals of every entry object with the input

if true

return the object

else

return null;

* **Which collision techniq does java use**

HashMap uses Separate Changing and a Supplement Hash Function to deal with hash collision problems. Java8 introduces new way to use balanced tree rather than linked list

* **Can we initialize hashmap with a default size, what if the value is negative?**

Yes we can init the hashmap with custom initial capacity, and custom loadFactor. If value is –ve a runtimeException namely illegal init capacity is thrown.

* **How shift operators work in java**

**e.g. 1<2, will add 2 zeros to the left side of the binary equivalent of 1.**

**And return the binary equivalent value.**

* **Hash table vs hashMap**
* **How hashmap works internally in java**

**Soln:** [**http://javaconceptoftheday.com/how-hashmap-works-internally-in-java/**](http://javaconceptoftheday.com/how-hashmap-works-internally-in-java/)

* **How hashSet works internally in java**

[**http://javahungry.blogspot.com/2013/08/how-sets-are-implemented-internally-in.html**](http://javahungry.blogspot.com/2013/08/how-sets-are-implemented-internally-in.html)

* **How ArrayList works internally in java**

[**http://netjs.blogspot.in/2015/08/how-arraylist-works-internally-in-java.html**](http://netjs.blogspot.in/2015/08/how-arraylist-works-internally-in-java.html)

* **Concurrent modification error in detail**
* **Deep copy vs shallow copy in ArrayList**
* **Inner/Nested classes**
* **Why we need them**

If you think that FooHelper will not at all be useful for other classes than Foo, then it makes sense to make it as private inner class of Foo.

Otherwise having it as a private instance looks good.

Yes, the advantage of using inner class is it can access members of outer class. In your case, if you think your FooHelper is not to be used by any other class, you can make it an inner class.

To check out the utility of inner class, go through the examples of AWT. Anonymous inner classes are widely used in event handlers.

One example of this kind of design can be found in HashMap where it defines a private inner class KeySet

Map.Entry is a good example of this. It's only used in conjunction with a Map, so having the definition of Entry be a part of the Map interface makes organizational sense.

For a good example of a legitimate use for member classes, see the source code for LinkedList.ListItr. This is a private inner class whose purpose is to provide an implementation of [ListIterator](http://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html) for a [LinkedList](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html). To do this, it's useful to have access to the private data inside the LinkedList. To achieve this using only top-level classes, it would have been necessary to expose more public methods in LinkedList to allow the ListIterator to get at the underlying implementation of the LinkedList. Instead, using an inner class allows LinkedList to keep its implementation private, as it should be.

* **Types of Nested classes**

<http://www.javatpoint.com/java-inner-class>

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

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

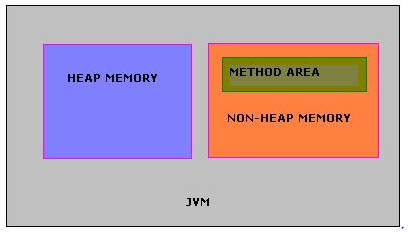
|  |  |
| --- | --- |
| **Type** | **Description** |
| [Member Inner Class](http://www.javatpoint.com/member-inner-class) | A class created within class and outside method. |
| [Anonymous Inner Class](http://www.javatpoint.com/anonymous-inner-class) | A class created for implementing interface or extending class. Its name is decided by the java compiler. |
| [Local Inner Class](http://www.javatpoint.com/local-inner-class) | A class created within method. |
| [Static Nested Class](http://www.javatpoint.com/static-nested-class) | A static class created within class. |
| [Nested Interface](http://www.javatpoint.com/nested-interface) | An interface created within class or interface. |

* **Pass by value vs. pass by reference**

**Java is always pass by value, as there is not concept of pointers or pointer arithmetic in java. Here the reference is merely a handle to the object that is created on the heap. When in case of a reference-based argument passing, there reference as passed by value. So we can modify the same object with the newly passed reference but cannot make the original reference point to a different object. A good litmus test will be a swap method which accepts 2 object reference, in java there will be not changes in the original reference, but in C++ the original reference will be swapped.**

**E.g. reference is like the controlling the same TV with multiple remotes**

* **Why is java pass by value and not pass by reference**
* **Memory Management:**



* A good rule of thumb is when the new keyword is involved the result will be on the heap.
* Local Variables are stored in Frames during runtime.
* Static Variables are stored in Method Area.
* Arrays are stored in heap memory.
* System programing
* Java heap vs premgen/metaspace(java 8)
* Fail fast vs Fail safe
* Paging segmentation and thrashing in memory management

🡺**Imperative vs. Declarative**

**Let's generalize and say that there are two ways in which we can write code: imperative and declarative.**

**We could define the difference as follows:**

**Imperative programming: telling the "machine" how to do something and as a result what you want to happen will happen.** (Ex. C, C++, Java)

**Declarative programming: telling the "machine"1 what you would like to happen, and let the computer figure out how to do it.** (Ex. SQL).

Take a query like:

SELECT **\*** from dogs

INNER JOIN owners

WHERE dogs.owner\_id **=** owners.id

Imagine trying to write the logic for this yourself imperatively:

*//dogs = [{name: 'Fido', owner\_id: 1}, {...}, ... ]*

*//owners = [{id: 1, name: 'Bob'}, {...}, ...]*

**var** dogsWithOwners **=** []

**var** dog, owner

**for**(**var** di**=**0; di **<** dogs.length; di**++**) {

dog **=** dogs[di]

**for**(**var** oi**=**0; oi **<** owners.length; oi**++**) {

owner **=** owners[oi]

**if** (owner **&&** dog.owner\_id **==** owner.id) {

dogsWithOwners.push({

dog**:** dog,

owner**:** owner

})

}

}}

}

**1 Computer/database/programming language/etc**

**Imperative** - The focus is on what steps the computer should take rather than what the computer willdo (ex. C, C++, Java).

**Declarative** - The focus is on what the computer should do rather than how it should do it (ex. SQL).

**Functional** - a subset of declarative languages that has heavy focus on recursion, Lisp, prolog.

2) There are some multiparadigm languages that kind of straddle both sides. For example, C#, Python, and JavaScript are mainly imperative languages that have some functional or declarative elements. There are also logic programming languages (like prolog) that mainly focus on satisfying constraints.

3) I believe that JQuery falls under the multiparadigm category above (like the language it's implemented in, JavaScript).

* OOPS:
* Composition vs aggregation vs association
* J2EE Servlets & Jsp
* App server vs Web Server
* <https://www.youtube.com/watch?v=BcmUOmvl1N8>
* <https://www.youtube.com/watch?v=ATObcDPLa40>

## How HTTP works

* <https://www.youtube.com/watch?v=RsQ1tFLwldY>
* <https://www.youtube.com/watch?v=emsdvKQyhl4>
* What is a servlet
* Servlet is a java class to respond a HTTP request and produce a HTTP response...... when we make a page with the use of HTML then it would be a static page so to make it dynamic we use SERVLET {in simple words one can understand} To make use of servlet is overcomed by JSP it uses the code and HTML tag both in itself..
* Servlets are programs that run on a Web or Application server and act as a middle layer between a request coming from a Web browser or other HTTP client and databases or applications on the HTTP server.
* A servlet is simply a class which responds to a particular type of network request - most commonly an HTTP request.
* Basically, servlets are usually used to implement web applications - but there are also various frameworks which operate on top of servlets (e.g. Struts) to give a higher-level abstraction than the "here's an HTTP request, write to this HTTP response" level which servlets provide.

## Jsp vs servlets

## How ajax calls work internally

* Web-services vs micro services

## Font type

It is best to avoid the use of non sans-serif fonts, such as Times New Roman.

The more ornate the font, the fewer the number of individuals who will be able to read it. 'Word Art' is not compatible with some screen readers.

The recommended fonts are:

* Arial
* Verdana
* Universe
* Helvetica
* **etc**
* It is all about knowing how to learn and
* starts with interest,
* driven by motivation,
* identifying the right resource,
* absorbing the essentials from the identified resource and
* practicing it in right way
* **Mahatma Gandhi**
* *“You must not lose faith in humanity. Humanity is an ocean; if a few drops of the ocean are dirty, the ocean does not become dirty.”*
* *“The difference between what we do and what we are capable of doing would suffice to solve most of the world’s problem.”*
* *“If I had no sense of humor, I would long ago have committed suicide.”*
* **1. Change yourself.**
* *“You must be the change you want to see in the world.”*
* *“As human beings, our greatness lies not so much in being able to remake the world – that is the myth of the atomic age – as in being able to remake ourselves.”*
* **2. You are in control.**
* *“Nobody can hurt me without my permission.”*
* **3. Forgive and let it go.**
* *“The weak can never forgive. Forgiveness is the attribute of the strong.”*
* *“An eye for eye only ends up making the whole world blind.”*
* **4. Without action you aren’t going anywhere.**
* *“An ounce of practice is worth more than tons of preaching.”*
* **5. Take care of this moment.**
* *“I do not want to foresee the future. I am concerned with taking care of the present. God has given me no control over the moment following.”*
* **6. Everyone is human.**
* *“I claim to be a simple individual liable to err like any other fellow mortal. I own, however, that I have humility enough to confess my errors and to retrace my steps.”*
* **7. Persist.**
* *“First they ignore you, then they laugh at you, then they fight you, then you win.”*
* **8. See the good in people and help them.**
* *“I look only to the good qualities of men. Not being faultless myself, I won’t presume to probe into the faults of others.”*
* *“Man becomes great exactly in the degree in which he works for the welfare of his fellow-men.”*
* *“I suppose leadership at one time meant muscles; but today it means getting along with people.”*
* **9. Be congruent, be authentic, be your true self.**
* *“Happiness is when what you think, what you say, and what you do are in harmony.”*
* *“Always aim at complete harmony of thought and word and deed. Always aim at purifying your thoughts and everything will be well.”*
* **10. Continue to grow and evolve.**
* Attributes of good developer
* No blind eyes
* Understand the overall requirements
* Ask Questions if requirements are not clear
* Is the ask justified,
* Is it missing from the existing system
* How to test it?
* Outline all the valid test cases, consider boundary conditions and all possible combinations
* Design a raw flow/draft
* code the most crucial code-flow/execution
* Unit Test
* Integration Test SIT
* Regression test - UAT
* Attributes of expert developer
* No blind eyes
* Remembers the things
* Debugs the code flow him-self
* Never breaks a code commit, also never overrides others changes
* Has in-depth analysis of the things
* Attention to details
* Does more works,
* Works smartly
* Scans logs via unix box cmds
* Amazon

 Who was your most difficult customer?  
  
- How would you introduce AWS in an elevator pitch?  
  
- What is the worst mistake you ever made?  
  
- If your direct manager was instructing you to do something you disagreed with, how would you handle it?  
  
- Describe what Human Resource means to you.  
  
- What is the angle between the hour hand and minute hand in an analog clock?  
  
- How do you detect whether or not a word is a palindrome?  
  
- Do you know our CEO? How do you pronounce his name?  
  
- Here's a string with numbers from 1-250 in random order, but it's missing one number. How will you find the missed number?  
  
- Are you willing to work on your feet for ten hours, four days a week?  
  
- Do you think you'll reach a point where you storm off the floor and never return?  
  
- Would you tell on a employee for stealing?  
  
- How would you solve problems if you were from Mars?  
  
- How do you persuade people?  
  
- Describe what happens in your browser as soon as you hit enter after writing a URL in the address bar.  
  
- Tell the story of the last time you had to apologize to someone.  
  
- Walk me through how Amazon Kindle books would be priced.  
  
- What would you do if you saw someone being unsafe at work?  
  
- What would you do if somehow you misdirected 10,000 units of something?  
  
- How would you improve Amazon's website?  
  
- You have 30 people working under you with 2 working indirect. Each employee can do 150 units/hour. Each work day has two 15 min breaks and one 30 min lunch.   
- What is the most difficult situation you have ever faced in your life? How did you handle it?  
  
- How would you tell a customer what Wi-Fi is?  
  
- You are Amazon and Samsung offers you 10,000 Samsung Galaxy S3s at a 34% discount. Is that a good deal?  
  
- Should we sell private label cleaning products?  
  
- What would you do if you found out that your closest friend at work was stealing?  
  
- Which Amazon leadership principle do you resonate most with?

* Companies –wise list
* Best BD companies globally

[Paxata](http://www.paxata.com/), [Qubole](https://www.qubole.com/), [Gainsight](http://www.gainsight.com/), [InsightSquared](http://www.insightsquared.com/), [Trifacta](https://www.trifacta.com/), [Kyvos Insights](http://www.kyvosinsights.com/), [SiSense](http://www.sisense.com/), [ThoughtSpot](http://www.thoughtspot.com/), [Google](http://www.google.com/), and [Looker](https://looker.com/)

* I further asked him what a company is actually looking for when they say they want to hire a big data engineer, he says, “Typically they are looking for people who can implement big data projects collecting logs, designing large scale warehouses, building backends for real-time and batch reporting, integrating different backend systems and so on. These are people who work with big data tooling. They are in great demand. Very few of them have to be expert in data science or actually writing code for large scale distributed systems. Their value lies in understanding and mastery of a complex and emerging basket of tools used for manipulating large data sets”
* **Best in Europe**

[**https://www.techgig.com/tech-news/10-best-countries-to-live-for-software-engineers-28210**](https://www.techgig.com/tech-news/10-best-countries-to-live-for-software-engineers-28210)

* **Best in pune 2016**
* Teredata hinjewadi, wikfield pune
* AmEx
* Capital One
* Thoughtworks
* Talentica
* Nvidia
* Symentec
* Datametica pune for cloud computing

##### [**IFC - International Finance Corporation**](https://www.linkedin.com/company/4783?trk=prof-exp-company-name)

##### [**The World Bank**](https://www.linkedin.com/company/166426?trk=prof-exp-company-name)

##### [**A.G.Technologies Pvt. Ltd**](https://www.linkedin.com/company/45801?trk=prof-exp-company-name)

##### [**Crowe Horwath LLP**](https://www.linkedin.com/company/5507?trk=prof-exp-company-name)

##### [**CPA Global**](https://www.linkedin.com/company/204705?trk=prof-exp-company-name)

##### [**Pitney Bowes**](https://www.linkedin.com/company/2503?trk=prof-exp-company-name)

* If interested please send me an updated copy of resume @ jitendra.sharma@irissoftware.com with good time to talk.
* [dineshaskhok.shiranpe@in.experis.com](mailto:dineshaskhok.shiranpe@in.experis.com)
* Highest paid software companies in Bangalore:

Product-based companies like:

* Amazon
* Microsoft
* Google
* LinkedIn
* Adobe
* Flipkart
* Cisco
* Juniper networks
* Zynga
* Nvidia
* Synopsys
* InMobi
* Cadence
* Qualcomm
* Intel

And among investment banks for software roles:

* Goldman Sachs
* Morgan Stanley
* JP Morgan & Chase
* Credit Suisse
* Bank of America
* US consutlansts for BD:

Recruiter Name:Rocky Sanghvi lodestar consultants

<https://www.linkedin.com/in/lodestarrocky>

Kumar Mangala lodestar consultants

Contact Company:Lodestar Consulting

Address:Newtown780 Newtown-Yardley RoadSuite 325PA,USA 18940

Email Address:hcooper@mylodestar.com , rsanghvi@mylodestar.com

Website:http://www.mylodestar.com

Telephone:1-609-2167474

Reference Id:BIG DATA GENERAL

**Design Patterns**

Google for: "erick banas object oriented design videos"



* **UK bank regulation**

**-Mifid**

**- mifid2**

**- sftr**

* **TIAA**

[**http://careers.tiaa.org/**](http://careers.tiaa.org/)

* **AT Kearney Inc US**
* PWC Strategy& LLC
* VMware Inc
* Splunk Inc
* Cadence Design Systems Inc
* **Data science**
* [**https://www.dataquest.io/blog/learn-data-science/?utm\_source=intercom&utm\_medium=email&utm\_campaign=signup\_5day\_inactive**](https://www.dataquest.io/blog/learn-data-science/?utm_source=intercom&utm_medium=email&utm_campaign=signup_5day_inactive)
* **Joins in sql**
* **Most complex sql queries**

[**https://bikashshaw.wordpress.com/2014/02/10/employee-id-employee-name-and-manager-id-self-join-sql-query-as-interview-question/**](https://bikashshaw.wordpress.com/2014/02/10/employee-id-employee-name-and-manager-id-self-join-sql-query-as-interview-question/)

* **LRU cache in java**

[**https://www.programcreek.com/2013/03/leetcode-lru-cache-java/**](https://www.programcreek.com/2013/03/leetcode-lru-cache-java/)

**Synchronized block vs synchronized method:**

Synchronized block is better than synchronized method, because in synchronized block only a critical section of code is locked, leaving behind the irrelevant code. This is the code which will give inconsistent data if accessed by multiple threads concurrently or simultaneously.

By using synchronized block we increase the performance, because we are not making the entrie method as synchronized, which degrades a the performance by allowing only one thread to execute the whole method at a time.

* Java 8 durga
* Key features of java-8
  1. lamda expressions
  2. functional interfaces
  3. default method in interfaces
  4. static method in interfaces
  5. pre-defined functional interfaces
     1. predicate
     2. function
     3. consumer
  6. method reference and constructor reference
  7. Optional
  8. Stream API
  9. JODA Time API
* Why do we need lamda expressions in java
* What is lamda expression in java/programming

Java is OO programming language, here everthing is either object or a class(static state or behavour). There are certain benefits of functional programming, which have been missing, so

lamda expressions are added in Java 8 and provide below functionalities of functional programming.

* Enable to treat functionality as a method argument, or code as data.
* A function that can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand/on the fly

Lamda expression is an anonymous function i.e. a function with

* No name
* No return type
* No access modifier

For e.g.

**public void m1() {**

**system.out.println(“hello lamda”);**

**}**

**()-> system.out.println(“hello lamda”);**

How to invoke/call lamda expressions?

We can only use/invoke lamda expression via functional interfaces,

Functional interfaces also called as clousers, are interfaces which have only a Single Abstract Method or SAM e.g Runnable, callable etc. Basically you need a reference ot call any method, so to call lamda expressions you need a reference of a functional interface. So no need for any implementation class, and can provide the in-line implementation of the SAM in the functional interface , as part of the calling code.

* Default method also called as defender methods or virtual extension methods.

In legacy java version, there is no provision for a interface to grow, i.e. if any new methods are introduced in the interface all the implementation classes break, So to avoid this java-8 provides new concept of default methods in interfaces, so all you get is a default method with a default implementation, and if you implementation class is not satisfied with this default implementation then it can override this implementation.

* Static methods in interface, Private is also allowed via java-9
* Built-in functional interfaces in java-8
* Predicate: uses lamda expression to test a condition returns Boolean, has **boolean test(T t) method**
* **Function takes 2 args, input and output and return output**

**Need to call the R apply(T t, R r) method**

* **Consumer, take only a single arg, and does not return any value, need to call the accept(T t) method**
* Function chaining

andThen vs Compose: in f1().andThen(f2), f1 is evaluated 1st and then f2.

In compose, f1().compose(f2), f2 is evaluated 1st and then f1

* Stream,
* Filter vs map

Filter is simple Boolean based filtering of the underlyging collection, if we need to apply and processing on each element of the underlyting collection we need to use map

* Map vs flatMap
* CompletableFuture , overcomes the limitations of the Future
* Micro-services

<https://www.youtube.com/watch?v=hVVG3WZgT84>

<https://www.youtube.com/watch?v=rBkD-OBeL0s>

<https://github.com/arunpa0206/microservices>

<https://www.javacodegeeks.com/2017/12/microservices-implementation-example-spring-boot.html>

in olden days, applications were written as a single monothic piece of software, wherein all the modules where tighly coupled and shared the same infra namely

* All services deployed on the same app-sever, dbase instance etc…

As new functions and users get added, this monolithic piece grows ever bigger, and slows down performance, and increases complexity of the code base.

To achieve scalability the only means was to add more severs, add more database nodes

* To counter the above problems, companies like Amazon, Netfix

Design their applications as a collections of independent services, that share nothing, yeah, nothing among themselves

The point is to break this monolicth piece of s/w into smaller logical modules/business areas

e.g. customer registration can be independent deployable & testable unit of code having its own database, and other infra.

e.g. payments can be another one.

Pricing/ order management etc

These micro-services can be build using different tech-stack, the one which best fits that particular business problem. E.g. customer registration can be build in java, payments in python etc.

These micro-services can communicate with each-other

Syn way - via rest(via http) or Thrift

Asyn way – via message broker, Kafka, MQBrowser

Beneftis of this approcach.

* Smaller team sizes(each team can finish its lunch in 2 pizzas)
* Smaller code base to deal with

Disadvantages of this approach

* Lot more communication & co-ordination needed to launch a new product/feature
* Need to have Std development and deployment methods processes in place
* Invesment on platform components like AWS etc, costly

If you are a large organization facing the complexity of scaling, then micro-services is the right way,

Spring-boot is a framework, that offer good support to build micro-servces by managing a lot of 3rd party dependencies, else you will need to do-it you-self.

* Eliminates a lot of boiler plate code
* Replaces configurations with opinionated default values to make it fast
* Easy modules to get stared with like web dev, and databse access
* All other spring related benefits

Disadvantges

* Migrate a legacy application to spring-boot, micro-services architecture for that matter

<https://www.edureka.co/blog/microservices-with-spring-boot/>

<https://github.com/koushikkothagal>

<https://www.javacodegeeks.com/2017/12/microservices-implementation-example-spring-boot.html>

<https://www.youtube.com/watch?v=F3uJyeAyv5g>

<https://javabrains.io/courses/spring_bootquickstart/>

<https://github.com/koushikkothagal/spring-boot-microservices-workshop>

serarch for below youtube videos

“Spring Boot Microservices java brains”

Traditional way vs micro-services

* The key factor is what gets deployed, you may have independent modules, etc but if its still a mono getting deployed that
* Spring boot vs spring cloud: in my understanding, spring boot is application centric i.e. it lets you write spring based applications very quickly, and on the micro-services architecture front, the communication between these different application or services is best done via cloud based infra like service discovery, load balancing, fault tolerance etc that again spring wraps up very easily
* Different ways to create a spring boot application
* Old and manual way, put all dependencies in the pom.xml do it yourself approach
* Using spring cli
* Using spring intilizer web interface
* restTemplate vs web-client

-restTemplate is an util object that does all the heavy-lifting while calling communicating between other micro-service. But it’s a synchronous way, it can be done asynch via web-client.

Web-client on the other hand is an asyn way of communication its based on the reactive mode or paradigm of programming where-in you as web-client to call a service end point and do the processing and you move ahead with other processing, once the call is returned the web-client will be notified.

* Reactive programing in java, flux and mono objects
* Web-client is based on the reactive programming font
* Service discovery: its one of the micro-services patterns, like when you build micro-services you need them to talk to each other, how to know what to talk to , its done via Eureka server
* Client side vs server side service discovery
* In client side each call goes through the discovery client, in server side all the process is done via a eureka server
* Eureka server, it the technology that help with service discovery, its

Basically is a server itself running on a standard port namely 8761(can also be changed), all the clients then need to register themselves with this server via application.properties and adding dependcies in their pom.xml

@loadBalance on the restTemplate bean

* Resilience and fault tolerance:

e.g. you need to connect to an external web-service

e.g. what if that particular service instance is slow

- set timeouts with restTemplate,

- timeouts,only help in partly solving the problem,

* Circuit Breaker pattern:
* Detect the slow/not responding component
* Hold-off sending request to it
* Resume sending the request, once the slow/not-responding component is recovered
* This pattern can be applied to any micro-service that call another micro-service
* TODO
* Circuit breaker parameters
* Fallback
* Hystrix
* Bulk-ahead pattern
* Convey the concept as it applies to micro-services
* Pattern vs technology
* Part-2

NIFI

<https://www.youtube.com/watch?v=VVnFt54jUQ8&list=PL55symSEWBbMBSnNW_Aboh2TpYkNIFMgb>

Kafka Streams

<https://www.youtube.com/watch?v=MCwhYP2uJ2s&list=PLkz1SCf5iB4cRM57IxhsNFHVR5JItXfDe>

* Open Questions to HR and hiring mangers

The questions are:

1. What qualities will a person in this role need to be successful in your company culture — as an individual and as a worker?

2. What's the company's position on education and development, including student loan reimbursement and tuition assistance?

3. How does the company keep employees excited, innovative, and motivated?