**Class**

Class is a template that describes the kinds of state and behavior that objects of its type support.

**Modifiers fall into two categories:**

* **Access modifiers:** public, protected, private.
* **Non-access modifiers**:strictfp, final,static,Synchronized, native , Volatile , Transient and abstract.

ASSS N FTV

More: <http://docstore.mik.ua/orelly/java-ent/jnut/ch03_14.htm>

There are four access *controls* (levels of access) but only three access *modifiers*.

Class can have only default and public access modifiers but inner class can be private.

In class declaration you can't always mix non-access modifiers. You're free to use strictfp in combination with Final.

**Native**

The native keyword is applied to a method to indicate that the method is implemented in native code using JNI.

Native methods are currently needed when

* You need to call a library from Java that is written in other language.
* You need to access system or hardware resources that are only reachable from the other language (typically C). Actually, many system functions that interact with real computer (disk and network IO, for instance) can only do this because they call native code.

Reason to make class final

- No one can override or change the behavior of your class for security purpose.

- To make immutable class like **String**

A non abstract class is called a concrete class

Interfaces:

Short:-Interfaces provide the contract for Classes. Interface is a 100-percent abstract class.

Interfaces define the common behaviors of Objects. If any Object keeps those behaviors then it can implement that Interface and express its own behaviors by its own way.

Example: Human (Male and Female)

Need: Java does not support multiple inheritances so we use interface to implement behaviors from more than two interfaces. Interface is only way to implement multiple inheritance in java.

If a class extended two other classes, and both superclasses has a doStuff() method, which version of doStuff() would the subclass inherit?

**Abstract Class:**

Same Definition of Interface

An abstract class can never be instantiated.

If even a single method is abstract, the whole class must be declared as abstract. If you do not declare the class as an abstract instead of having abstract method then your class is illegal.

Take advantage of polymorphism.

When the sub-types behavior is totally different then you use an interface, when the sub-types behavior is partially common and different with respect to the supertype an abstract class is used.

In an abstract class the partially common behavior is given a concrete implementation.

If all the behaviours in interface is fixt and never going to change in future then we can use interface

But in future if we are going to add or remove some behaviour then we can use abstract class

Interface is a 100-percent abstract class.

Abstract class can define both abstract and non-abstract methods; an interface can have only abstract methods.

All interface methods are implicitly public and abstract if u declares or not.

All variables defined in an interface must be public, static, and final if u declare or not—in other words, interfaces can declare only constants, not instance variables.

Interface can only extend one or more interfaces but not implement anything.

Interface types can be used polymorphically.

We can declare the Interface with default and public access controls.

**Class Members access modifiers**.

Whereas a class can use just two of the four access control levels (default or public), members can use all four.

Protected member can be accessed (through inheritance) by a subclass even if the subclass is in a different package.

There is never a case where an access modifier can be applied to a local variable (method parameters and variable inside the methods are local variables) because local variable's scope is only in method not outside the method so there is not necessary. In fact, there is only one modifier that can ever be applied to local variables—final.

**Class Members non- access modifiers**.

Thread class has a method called isAlive() that is final so we cannot override it.

An abstract method is a method that's been declared (as abstract) but not implemented.

The first concrete (non-abstract class) subclass of an abstract class must implement all abstract methods of the superclass.

A method can never, ever, ever be marked as both abstract and final, or both abstract and private.

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

**Synchronized** and **native** modifier can be applied only to methods—not variables, not classes, just methods.

The **strictfp** modifier can be applied to methods and classes —not to variables.

When you are calling any method by passing value into methods then those values are called **arguments** and

When you are creating methods with values then those values are called **parameters**.

Constructor declarations can however have all of the normal access modifiers, and they can take arguments (including var-args), just like methods.

Constructors can't be marked static, final and abstract.

There are two types of variables in Java: Primitives and Reference variables and both can be declared as class variables (statics), instance variables, method parameters, or local variables.

Instance variable can use all of 4 access-controls and only final and transient non-access.

Local variables are always on the stack.

The compiler will reject any code that tries to use a local variable that hasn't been assigned a value, because—unlike instance variables—local variables don't get default values.

You can make instance and local variable with same name but to identify which is local and which instance variable you can use **this** keyword to identify instance variables.

The keyword this always, always, always refers to the object currently running.

Array itself will always be an object on the heap, even if the array is declared to hold primitive elements.

**int[5] scores;**

**The preceding code won’t compile. Remember, the JVM doesn’t allocate space until you actually instantiate the array object. That’s when size matters.**

What does it mean to have a final object reference variable? A reference variable marked final can't ever be reassigned to refer to a different object but the data within the object can be modified.

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

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.

**Static**

There will be only one copy of a static member regardless of the number of instances of that class. In other words, all instances of a given class share the same value for any given static variable.

You can mark as static: Methods, Variables, A class nested within another class, but not within a method, Initialization blocks.

You can't mark as static: Constructors, Classes (unless they are nested), Interfaces, Method local inner classes, Inner class methods and instance variables, Local variables.

A static method can't access a non-static (instance) variable, because there is no instance! That's not to say there aren't instances of the class alive on the heap, but rather that

Even if there are, the static method doesn't know anything about them. The same applies to instance methods; a static method can't directly invoke a non-static method. Think static = class, non-static = instance.

class Foo {

int x = 3;

public static void main (String [] args) {

System.out.println("x is " + x);

}

}

Understand that this code will never compile, because you can’t access a non-static (instance) variable from a static method. Just think of the compiler saying,

Static methods can't be overridden! This doesn't mean they can't be redefined in a subclass, but redefining and overriding isn’t the same thing.

Let's take a look at an example of a redefined (remember, not overridden), static method:

class Animal {

static void doStuff() {

System.out.print("a ");

}

}

class Dog extends Animal {

static void doStuff() { // it's a redefinition,

// not an override

System.out.print("d ");

}

}

**Inheritance**

Allows code defined in one class to be reused in other classes.

You express the IS-A relationship in Java through the keywords extends (for class inheritance) and implements (for interface implementation).

Has-a relationship means a class has an instance of another class as an instance variable.

**Abstraction**

Abstraction in Java is achieved by using interface and abstract class.

Abstraction is a process of hiding the implementation details and showing only functionality to the user. Abstraction lets you focus on what the object does instead of how it does it.

**Encapsulation**

One way to think about encapsulation is as a protective wrapper that prevents code and data from being arbitrarily accessed by other code defined outside the wrapper.

Encapsulated code has two features:

. Instance variables are kept protected (usually with the private modifier).

. Getter and setter methods provide access to instance variables.

**Polymorphism**

Polymorphism means more than one form, same object performing different operations according to the requirement.

By single method we can perform more than one behavior.

Polymorphism can be achieved by using two ways, those are

**Method overriding**

**Method overloading**

Example: Giraffe and Crocodile are both Animals, and animals can Move. If you have an instance of an Animal then you can call Move without knowing or caring what type of animal it is.

The reference variable's type (not the object's type), determines which methods can be called.

Polymorphic method invocations apply only to overridden instance methods.

Remember, any Java object that can pass more than one IS-A test can be considered polymorphic. Other than objects of type Object, all Java objects are polymorphic in that they pass the IS-A test for their own type and for class Object. Remember that the only way to access an object is through a reference variable,and there are a few key things to remember about references:

* A reference variable can be of only one type, and once declared, that type can never be changed (although the object it references can change).
* A reference is a variable, so it can be reassigned to other objects, (unless the reference is declared final).
* A reference variable's type determines the methods that can be invoked on the object the variable is referencing.
* A reference variable can refer to any object of the same type as the declared reference,or.this is the big one.it can refer to any subtype of the declared type!
* A reference variable can be declared as a class type or an interface type. If the variable is declared as an interface type, it can reference any object of any class that implements the interface.

Example:

Class PlayerPiece extends GameShape implements Animatable {

public void movePiece() {

System.out.println("moving game piece");

}

public void animate() {

System.out.println("animating...");

}

// more code

}

So now we have a PlayerPiece that passes the IS-A test for both the GameShape class and the Animatable interface. That means a PlayerPiece can be treated polymorphically as one of four things at any given time, depending on the declared type of the reference variable:

¡ An Object (since any object inherits from Object)

¡ A GameShape (since PlayerPiece extends GameShape)

¡ A PlayerPiece (since that's what it really is)

¡ An Animatable (since PlayerPiece implements Animatable)

The following are all legal declarations. Look closely:

PlayerPiece player = new PlayerPiece();

Object o = player;

GameShape shape = player;

Animatable mover = player;

There's only one object here.an instance of type PlayerPiece.but there are four different types of reference variables, all referring to that one object on the heap. Pop quiz: which of the preceding reference variables can invoke the

displayShape() method? Hint: only two of the four declarations can be used to invoke the displayShape() method.

**Remember that method invocations allowed by the compiler are based solely on the declared type of the reference, regardless of the object type.**

**Methods can be overridden or overloaded; constructors can be overloaded but not overridden.**

**Overriding**

If you declare two methods with same name with same argument in both of parent and child class then it is called overriding the method of parent class in child class.

**Static binding / Dynamic binding**

If you have more than two methods with same name in class hierarchy (in case of method overloading and overriding) then which one method to invoke is decided based on Static Binding and Dynamic binding.

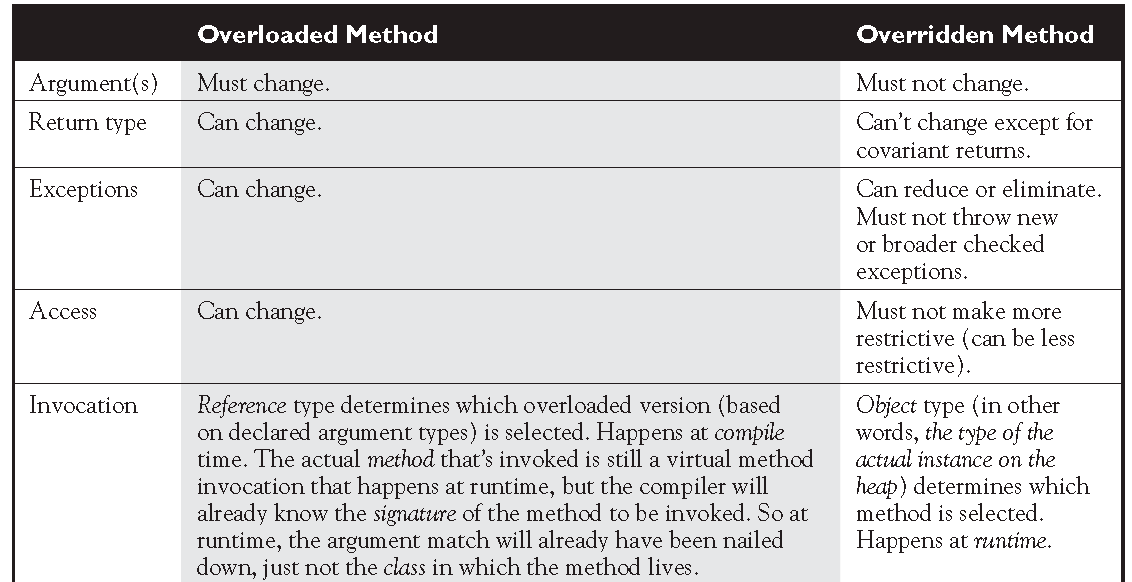
1) Static binding in Java occurs during Compile time while Dynamic binding occurs during Runtime.

2) private, final and static methods and variables uses static binding and bonded by compiler while virtual methods are bonded during runtime based upon runtime object.

3) Static binding uses Reference Type for binding while Dynamic binding uses Object Type for binding.

**Overloading**

Methods with same name but different arguments and different return types in same Class is called Method Overloading.



### Rules of Method Overloading in Java

* First and foremost rule to overload a method in Java is to change method signature. Method signature is made of number of arguments, type of arguments and order of arguments if they are of different types.
* Return type of method is not part of method signature, so just changing the return type will not overload method in Java.  See what is method overloading in Java for code example of these rules.

**Rules of Method Overriding in Java**

* a method can only be written in Sub class, not in same class
* Overriding method cannot throw checked Exception which is higher in hierarchy, than checked Exception thrown by overridden method. For example if overridden method throws IOException or [ClassNotfoundException](http://javarevisited.blogspot.sg/2011/08/classnotfoundexception-in-java-example.html), which are checked Exception, than overriding method cannot throw java.lang.Exception because it comes higher in type hierarchy (it's super class ofIOException and ClassNotFoundExcepiton).
* Overriding method cannot reduce access of overridden method. It means if overridden method is defined as public than overriding method cannot be protected or package private.
* Overriding method can increase access of overridden method. This is opposite of earlier rule, according to this if overridden method is declared as protected than overriding method can be protected or public.
* Private, static and final method cannot be overridden in Java.

You can extend only one class, but implement many interfaces.

As long as you've changed the argument list, you're overloading the method and if you are not changing the argument list and you are changing only return type then it is illegal overloading.

Polymorphism applies to overriding, not to overloading.

**Constructor**

Two key points to remember about constructors are that they have no return type and their names must exactly match the class name.

Constructors are used to initialize instance variable state.

Constructors are invoked at runtime when you say new on some class.

Every class, even an abstract class, has at least one constructor.

But what really happens when you say new Horse ()? (Assume Horse extends Animal and Animal extends Object.)

* Every constructor invokes the constructor of its superclass with an (implicit) call to super().
* Animal constructor is invoked (Animal is the superclass of Horse).
* Object constructor is invoked (Object is the ultimate superclass of all classes).
* By using new keyword at runtime we can assign values to Object instance variables.
* Object constructor completes.
* Animal instance variables are given their explicit values (if any).
* Animal constructor completes.
* Horse instance variables are given their explicit values (if any).
* Horse constructor completes.

1. If you don't type a constructor into your class code, a default constructor will be automatically generated by the compiler.

2. The only way a constructor can be invoked is from within another constructor. In other words, you can't write code that actually calls a constructor.

3. The compiler will generate a default constructor for the class if you don't define any constructors in the class. If you define a constructor in class then the compiler will not generate a default constructor for the class.

4. Constructors are never inherited. They aren't methods. They can't be overridden (because they aren't methods and only instance methods can be overridden).

5. Key Rule: The first line in a constructor must be a call to super() or a call to this().

No exceptions. If you have neither of those calls in your constructor, the compiler will insert the no-arg call to super(). In other words, if constructor A() has a call

to this(), the compiler knows that constructor A() will not be the one to invoke super().in single constructor you can never use both this() and super().

Methods with an object reference return type can return a subtype.

Instance members are accessible only after the super constructor runs.

Interfaces do not have constructors.

**Cohesion and coupling**

Cohesion refers to the degree in which a class has a single, well-defined responsibility.

All good software design will go for high **cohesion** and low **coupling**.

The most effective method of decreasing coupling and increasing cohesion is design by interface.

**Stack and Heap**

Instance variables and objects live on the heap.

Local variables and methods live on the stack.

Each thread has its own stack memory so stack is, thread safe and Heap is not thread safe.

**Variable Scope**

* Static variables have the longest scope; they are created when the class is loaded, and they survive as long as the class stays loaded in the Java Virtual Machine (JVM).
* Instance variables are the next most long-lived; they are created when a new instance is created, and they live until the instance is removed.
* Local variables are next; they live as long as their method remains on the stack. As we'll soon see, however, local variables can be alive, and still be "out of scope".
* Block variables live only as long as the code block is executing.

Default value of Char is '\u0000'

**Pass-By-value & Pass-By-Reference**

Java is always pass-by-value.

Pass by value in java means passing a copy of the value to be passed. Pass by reference in java means the passing the address itself.

With Java objects, the object reference itself is passed by value and so both the original reference and parameter copy both refer to the same Java object.

**Wrapper classes**

**Need**: Primitives values cannot participate in the object activities, such as being returned from a method as an object, and being added to a Collection of objects as an Object so by Wrapper classes we can wrap primitives’ values into object and use them as an Object.

Wrapper classes are final.

All of the wrapper classes except Character provide two constructors: one that takes a primitive of the type being constructed, and one that takes a String representation of the type being constructed—for example,

Integer i1 = new Integer(42);

Integer i2 = new Integer("42");

**valueOf()** is a static Method for all wrapper classes used to wrape primitive values into Object.

Integer i2 = Integer.valueOf("101011", 2);

Float f2 = Float.valueOf("3.14f");

When you need to convert the value of a wrapped numeric to a primitive, use

one of the many **xxxValue()** and **parseXxx()**methods

Integer i2 = new Integer(42);

byte b = i2.byteValue();

short s = i2.shortValue();

double d = i2.doubleValue();

double d4 = Double.parseDouble("3.14");

* **xxxValue() - to convert a Wrapper to a primitive**
* **parseXxx(String) - to convert a Wrapper to a primitive**
* valueOf(String) - to convert a primitive to a Wrapper

**Autoboxing**

When Java automatically converts a primitive type like int into corresponding wrapper class object e.g. Integer than its called autoboxing.

**When Happen :** Autoboxing and unboxing can happen anywhere where an object is expected and primitive type is available for example In method invocation where an object argument is expected

Like Collections deals with only Object so it will get that object unbox it and use and again box it automatically.

Boxing is when you convert a primitive type to a reference type, un-boxing is the reverse.

Note: When == is used to compare a primitive to a wrapper, the wrapper will be unwrapped and the comparison will be primitive to primitive.

The general rule is that boxing and unboxing work wherever you can normally use a primitive or a wrapped object.

**Widening v/s Autoboxing and var-args**

byte---short---int---long---float---double

If I move from left to right on the above, then I am widening my primitive types and right to left is narrowing because the number of bytes gets smaller.

class AddBoxing {

static void go(Integer x) { System.out.println("Integer"); }

static void go(long x) { System.out.println("long"); }

public static void main(String [] args) {

int i = 5;

go(i); // which go() will be invoked?

}

}

The answer is that the compiler will choose widening over boxing,

so the **output** will be

long

class AddVarargs {

static void go(int x, int y) { System.out.println("int,int");}

static void go(byte... x) { System.out.println("byte... "); }

public static void main(String[] args) {

byte b = 5;

go(b,b); // which go() will be invoked?

}

}

Because, once again, even though each invocation will require some sort of conversion, the compiler will choose the older style before it chooses the newer style, keeping existing code more robustthe.

so the **output** is

int,int

*  Widening beats boxing
*  Widening beats var-args

Remember, none of the wrapper classes will widen from one to another! Bytes won’t widen to Shorts, Shorts won’t widen to Longs, etc.

**Widening Reference Variables**

class Animal {static void eat() { } }

class Dog3 extends Animal {

public static void main(String[] args) {

Dog3 d = new Dog3();

d.go(d); // is this legal ?

}

void go(Animal a) { }

}

No problem! The go() method needs an Animal, and Dog3 IS-A Animal

**Garbage Collector**

Java's garbage collector provides an automatic solution to memory management by removing objects from memory when they're no longer needed.

The garbage collector is under the control of the JVM. The JVM decides when to run the garbage collector.

The important concept to understand for the exam is when does an object become eligible for garbage collection?

An object is eligible for garbage collection when no live thread can access it.

To make the object eligible (for GC), we set the reference variable to null.

The objects in method is not eligible to garbage collector until the method is running and after the completion of method , objects inside the methods are eligible for GC.

System.gc() allow us to request that the JVM perform garbage collection.

Java provides you a mechanism to run some code just before your object is deleted by the garbage collector. This code is located in a method named finalize() that all classes inherit from class Object.

There are a couple of concepts concerning finalize() that you need to remember.

* For any given object, finalize() will be called only once (at most) by the garbage collector.

Prefix means ++i means it will increment i before using it.

Postfix means i++ means it will use i before incrementing it.

**String**

**What is String literal pool?**

To make Java more memory efficient, the JVM has a special area of memory called the "String constant pool." When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created.

**Mutable Objects**: When you have a reference to an instance of an object, the contents of that instance can be altered

**Immutable Objects**: When you have a reference to an instance of an object, the contents of that instance cannot be altered

* By making all attributes to final of class we can make immutable object.
* String objects are immutable, and String reference variables are not.
* If you create a new String without assigning it, it will be lost to your program.

**Why String is immutable or final?**

1. Since String is immutable it can safely be shared between many threads, by which we avoid synchronization issues.

2. We can store string in string pool for example as below. If it was mutable then this would not be possible.

3. String immutability allows to cache its hash value which greatly improves String performance in Hash based collections.

Mutable objects are objects that can’t be changed programmatically. They’re especially good for multi-threaded environments or other environments where more than one process is able to alter (mutate) the values in an object.

4. Better memory management –

Consider

String s1=”hello world”;

String s2=”hello world”;

JVM automatically points both s1 and s2 to the same address in the heap memory. This decreases the usage of memory when dealing with a large number of Strings in your program.

**Difference between String,StringBuilder and StringBuffer?**

1) String is immutable while StringBuffer and StringBuilder is mutable object.

2) StringBuffer is synchronized while StringBuilder is not which makes StringBuilder faster than StringBuffer.

3) Concatenation operator "+" is internal implemented using either StringBuffer or StringBuilder.

4) **Use String if you require immutability,** use Stringbuffer in java if you need mutable + thread-safety and use StringBuilder in Java if you require mutable + without thread-safety.

**Java Serialization**

The process of saving an Object’s state into a file in the form of bytes and send it over the network is called Serialization.

We need serialization because of security purpose.

When you want to serialize an object, that respective class should implement the marker interface **serializable**. It just informs the compiler that this java class can be serialized. You can tag properties that should not be serialized as transient. You open a stream and write the object into it. Java API takes care of the serialization protocol and persists the java object in a file in conformance with the protocol. De-serialization is the process of getting the object back from the file to its original form.

Serializable is a "**marker interface**" and marker interfaces contain nothing.

If a superclass is Serializable, then according to normal Java interface rules, all subclasses of that class automatically implement Serializable implicitly.

The ObjectOutputStream.writeObject() method serializes objects, and the ObjectInputStream.readObject() method deserializes objects.

class SerializationBox implements Serializable {

private byte serializableProp = 10;

public byte getSerializableProp() {

return serializableProp;

}

}

public class SerializationSample {

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

FileNotFoundException, ClassNotFoundException {

SerializationBox serialB = new SerializationBox();

serialize("serial.out", serialB);

SerializationBox sb = (SerializationBox) deSerialize("serial.out");

System.out.println(sb.getSerializableProp());

}

public static void serialize(String outFile, Object serializableObject)

throws IOException {

FileOutputStream fos = new FileOutputStream(outFile);

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(serializableObject);

}

public static Object deSerialize(String serilizedObject)

throws FileNotFoundException, IOException, ClassNotFoundException {

FileInputStream fis = new FileInputStream(serilizedObject);

ObjectInputStream ois = new ObjectInputStream(fis);

return ois.readObject();

}

}

**I/O classes:**

* File
* FileReader
* BufferedReader
* FileWriter
* BufferedWriter
* PrintWriter
* Console

Stream classes are used to read and write bytes, and Readers and Writers are used to read and write characters.

Invoking the flush() method guarantees that the last byte of has been written successfully into the file.

When you are doing file I/O you're using operating system resources, and so invoking close() will free up all those resources.

**Difference between PrintWriter and FileWriter class**

Although both of these internally uses a FileOutputStream , the main difference is that PrintWriter offers some additional methods for formatting like println and printf.

Major Differences:

* FileWriter throws IOException in case of any IO failure.

None of the PrintWriter methods throws IOException, instead they set a boolean flag which can be obtained using checkError().

* PrintWriter automatically invokes flush after every byte of data is written. In case of FileWriter, caller has to take care of invoking flush.

**Deep Copy and Shallow Copy**

**Shallow copy** is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied but in deep object other object's reference is also copied with content.

In both case by using Object's clone () method we make copy of object

<http://www.jusfortechies.com/java/core-java/deepcopy_and_shallowcopy.php>

**Collection**

**toString()** gives you the class name followed by the @ symbol, followed by the unsigned hexadecimal representation of the object's hashcode. Like **HardToRead@a47e0**

Comparing two object references using the == operator evaluates to true only when both references refer to the same object.(both has same memory address on heap)

Comparing two object using the .equal() method evaluates to true only when their contents are meaningfully equivalent.

By default, the equals() method actually behaves the same as the “==” operator

Third difference between equals and == operator is that, You cannot change the behavior of == operator but we can override equals() method and define the criteria for the objects equality.

**Why we need to override equal() method of Object?**

Default implementation of equals() class provided by java.lang.Object compares memory location and only return true if two reference variable are pointing to same memory location .

Java recommends to override equals and hashCode method if equality is going to be define by logical way or via some business logic and many classes in Java standard library does override it e.g. String overrides equals, whose implementation of equals() method return true if content of two String objects are exactly same.

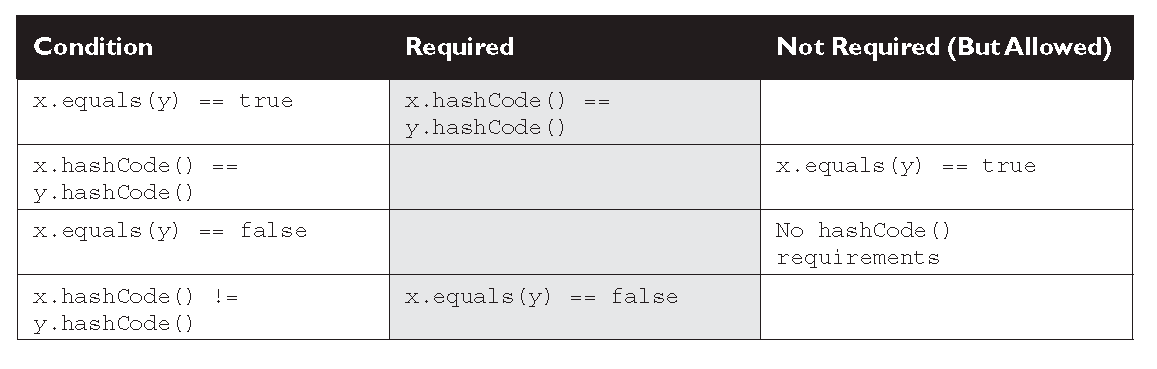
**The equals() Contract**

Pulled straight from the Java docs, the equals() contract says

* **Reflexive** : Object must be equal to itself.
* **Symmetric** : if a.equals(b) is true then b.equals(a) must be true.
* **Transitive** : if a.equals(b) is true and b.equals(c) is true then c.equals(a) must be true.
* **Consistent** : multiple invocation of equals() method must result same value until any of properties are modified. So if two objects are equals in Java they will remain equals until any of there property is modified.
* **Null compareison** : comparing any object to null must be false

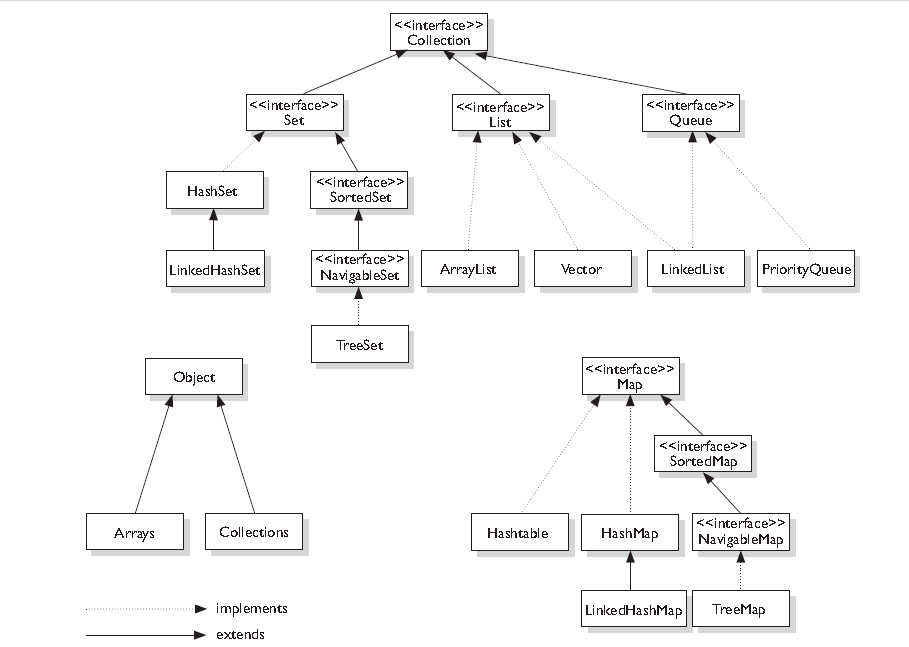
**Hashcode() Contract**

* If two objects are equal by equals() method then there hashcode must be same.
* If two objects are not equal by equals() method then there hashcode could be same or different.



***1. Find the right bucket (using*** hashCode()***)***

***2. Search the bucket for the right element (using*** equals() ***).***



Collections is a class, with static utility methods, while Collection is an interface with declarations of the methods common to most collections including add(), remove(), contains(), size(), and iterator().

**An implementation class can be unsorted and unordered, ordered but unsorted, or both ordered and sorted. But an implementation can never be sorted but unordered.**

**An ordered collection maintains the order of the elements based on insertion order or natural order into collection. A List is an example.**

**A SortedSet and TreeSet is an example.**

**A sorted collection means that the order in the collection is determined according to some rule.**

Since sorting is a type of ordering, so when a collection is sorted, it is automatically ordered.

**For Example :**

For a collection of String objects, the **natural order** is alphabetical. For Integer objects, the natural order is by numeric value.

There is **no natural order** for any Foo Objects.

The key part is that ordering doesn't depend upon the property of the element inserted but sorting does.

**List Interface**

It can contain duplicates.

All three List implementations are ordered by index position.

**1). ArrayList**

ArrayList is ordered collection (by index), but not sorted.

**2).Vector**

A Vector is basically the same as an ArrayList, but Vector methods are synchronized for thread safety. You'll normally want to use ArrayList instead of Vector because the synchronized methods add a performance hit you might not need.

**3).LinkedList**

A LinkedList ordered collection (by index) but LinkedList may iterate more slowly than an ArrayList.

**Set Interface**

It doesn't allow duplicates.

**1).HashSet**

A HashSet is an unsorted, unordered Set.

**2).LinkedHashSet**

A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements.

It maintains insertion order.

**3).TreeSet**

The TreeSet is one of two sorted collections (the other being TreeMap).

It uses a Red-Black tree structure.

TreeSet lets us define a custom sort order via a Comparable or Comparator Interface.

**Map Interface**

A Map cares about unique identifiers. You map a unique key (the ID) to a specific value.

Like Sets, Maps rely on the equals() method to determine whether two keys are the same or different.

**1).HashMap**

The HashMap gives you an unsorted, unordered Map like **HashSet.**

HashMap allows one null key and multiple null values in a collection.

**2.Hashtable**

Hashtable doesn't let you have anything that's null.

A Hashtable is basically the same as an HashMap but Hashtable methods are synchronized for thread safety.

**3).LinkedHashMap**

LinkedHashMap is an ordered version of HashMap.

It maintains insertion order.

**4).TreeMap**

The TreeMap is sorted collections like TreeSet.

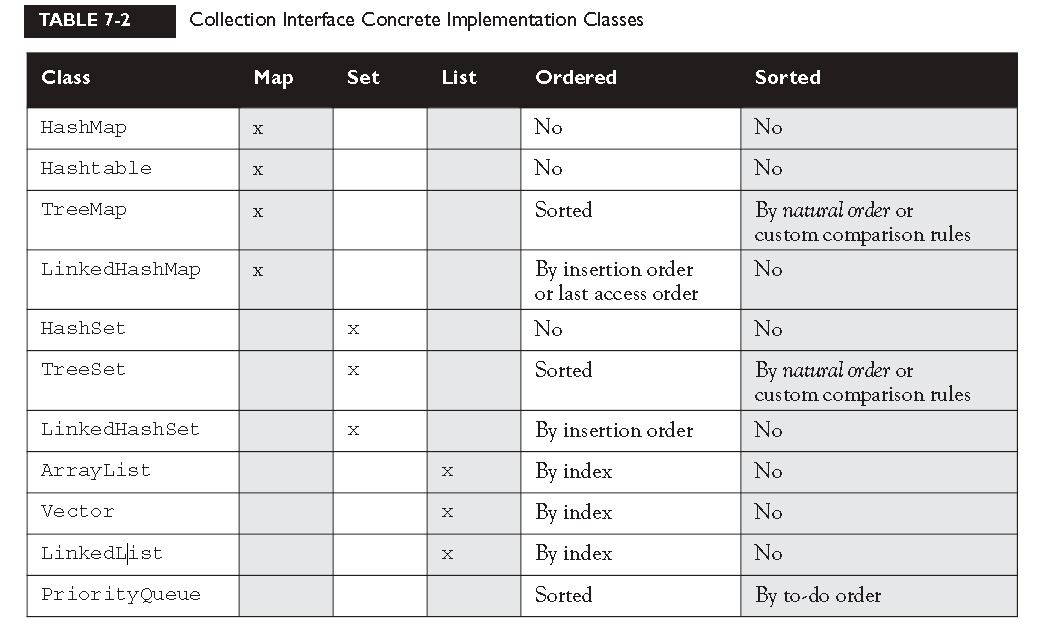
TreeMap lets us define a custom sort order via a Comparable or Comparator Interface.

**Queue Interface**

It uses FIFO first-in first-out order to insert or remove the elements.

**1).PriorityQueue**

The purpose of a PriorityQueue is to create a "priority-in, priority out" queue as opposed to a typical FIFO queue. A PriorityQueue's elements are ordered either by natural ordering (in which case the elements that are sorted first will be accessed first) or according to a Comparator. In either case, the elements' ordering represents their relative priority.



**LinkedList vs ArrayList**

1. LinkedList doesn't provide index based access and we need to iterate over linked list to retrieve any element while Arralist provide.
2. Insertion and deletion is faster in LinkedList than ArrayList.

Arralist internally uses array & when you run an "add" command a new Array of size more than old array will be created and in deletion that internal array will be modified. Whereas in linkedlist there are three parts. First part stores the pointer to previous element, second stores the data & third stores pointer to next element. So when insertion & deletion happens in linkedlist these pointers get updated. That’s why linkedlist is fast.

1. LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

**TreeSet vs LinkedHashSet vs HashSet**

1. HashSet is fastest, LinkedHashSet is second on performance or almost similar to HashSet but TreeSet is bit slower because of sorting operation it needs to perform on each insertion.
2. Both HashSet and LinkedHashSet allows null but TreeSet doesn't allow null.
3. HashSet and LinkedHashSet uses equals() method for comparison but TreeSet uses compareTo() method for comparison.

**Differences between HashMap and Hashtable in Java:**

1. [Hashtable](http://java.sun.com/javase/6/docs/api/java/util/Hashtable.html) is synchronized, whereas [HashMap](http://java.sun.com/javase/6/docs/api/java/util/HashMap.html) is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
2. Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
3. One of HashMap's subclasses is [LinkedHashMap](http://java.sun.com/javase/6/docs/api/java/util/LinkedHashMap.html), so in the event that you'd want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.

HashSet calls put(K,V) on the internal HashMap which just inserts the new object overwriting the old entry if duplicate.

================================================================================================================================================================================================================

ArrayList<DVDInfo> dvdList = new ArrayList<DVDInfo>();

ArrayList<String> stringList= new ArrayList<String>();

By using Collesctions class's **sort()** method we can sort **stringList** but when we are trying to sort **dvdList** then we will get exception because Collections.sort(List<Object>) method sort only that list of Object who implement **Comparable** Interface .

**Comparable Interface**

The Comparable interface is used by the Collections.sort() method and the java.util.Arrays.sort() method to sort Lists and arrays of objects, respectively.

To implement Comparable, a class must implement a single method **compareTo().**

int x = thisObject.compareTo(anotherObject);

The compareTo() method returns an int with the following characteristics:

* negative If thisObject < anotherObject
* zero If thisObject == anotherObject
* positive If thisObject > anotherObject

The sort() method uses compareTo() to determine how the List or object array should be sorted.

class DVDInfo implements Comparable {

public int compareTo(Object o) {

DVDInfo d = (DVDInfo)o;

return title.compareTo(d.getTitle());

} }

Now By Implementing Comparable interface , sort() method can sort dvdList.

It’s important to remember that when you override equals() you MUST take an argument of type Object, but that when you override compareTo() you should take an argument of the type you’re sorting.

Like below for dvdlist type

class DVDInfo implements Comparable<DVDInfo> {

public int compareTo(DVDInfo d) {

return title.compareTo(d.getTitle());

} }

**Comparator Interface**

While you were looking up the Collections.sort() method you might have noticed that there is an overloaded version of sort() that takes both a List AND something called a Comparator.

The Comparator interface gives you the capability to sort a given collection any number of different ways.

The other handy thing about the Comparator interface is that you can use it to sort instances of any class—even classes you can't modify —unlike the Comparable interface, which forces you to

Change the class whose instances you want to sort.

The Comparator interface is also very easy to implement, having only one method, compare().

Here's a small class that can be used to sort a List of DVDInfo instances, by genre.

class GenreSort implements Comparator<DVDInfo>

{

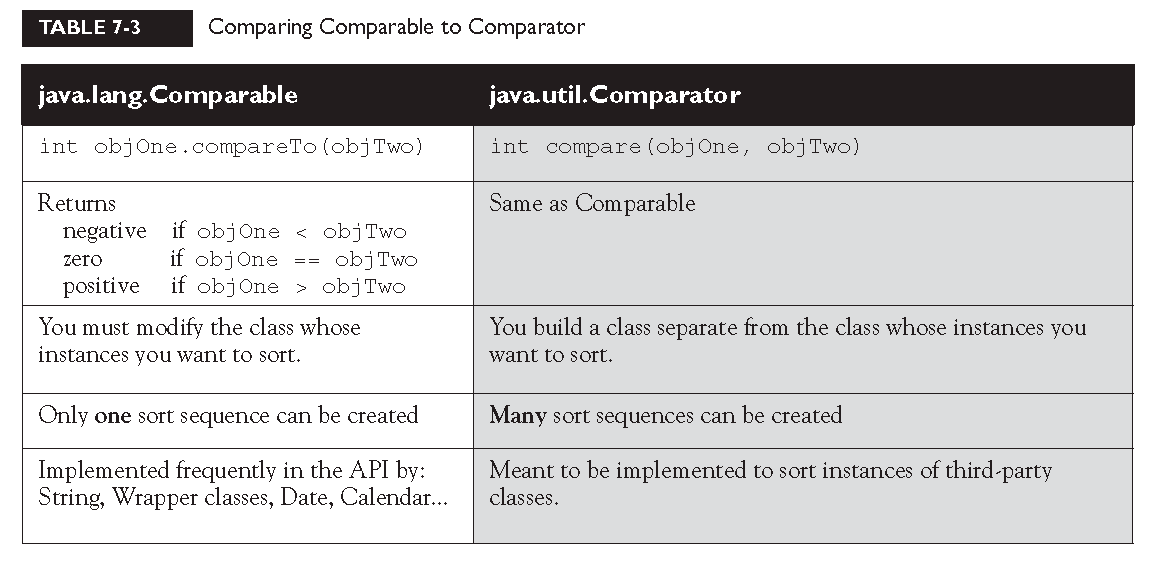
public int compare(DVDInfo one, DVDInfo two)

{

return one.getGenre().compareTo(two.getGenre());

}

}



**Whenever you want to sort an array or a collection, the elements inside must all be mutually comparable**. In other words, if you have an Object[] and you put Cat and Dog objects into it, you won’t be able to sort it. In general, objects of different types should be considered NOT mutually comparable.

**Searching Arrays and Collections**

The Collections class and the Arrays class both provide methods that allow you to search for a specific element. When searching through collections or arrays, the following rules apply:



* Searches are performed using the binarySearch() method.
* Successful searches return the index of the element being searched.
* Unsuccessful searches return an index that represents the insertion point.
* The collection/array being searched must be sorted before you can search it or it will not return a proper result.
* If the collection/array you want to search was sorted in natural order, it must be searched in natural order.
* If the collection/array you want to search was sorted using a Comparator, it must be searched using the same Comparator, which is passed as the second argument to the binarySearch() method.

**Navigating (Searching) TreeSets and TreeMaps**

1. The last ferry that leaves before 4 (1600 hours)

2. The first ferry that leaves after 8 (2000 hours)

public class Ferry {

public static void main(String[] args) {

TreeSet<Integer> times = new TreeSet<Integer>();

times.add(1205); // add some departure times

times.add(1505);

times.add(1545);

times.add(1830);

times.add(2010);

times.add(2100);

// Java 5 version

TreeSet<Integer> subset = new TreeSet<Integer>();

subset = (TreeSet)times.headSet(1600);

System.out.println("J5 - last before 4pm is: " + subset.last());

TreeSet<Integer> sub2 = new TreeSet<Integer>();

sub2 = (TreeSet)times.tailSet(2000);

System.out.println("J5 - first after 8pm is: " + sub2.first());

// Java 6 version using the new lower() and higher() methods

System.out.println("J6 - last before 4pm is: " + times.lower(1600));

System.out.println("J6 - first after 8pm is: " + times.higher(2000));

}

}

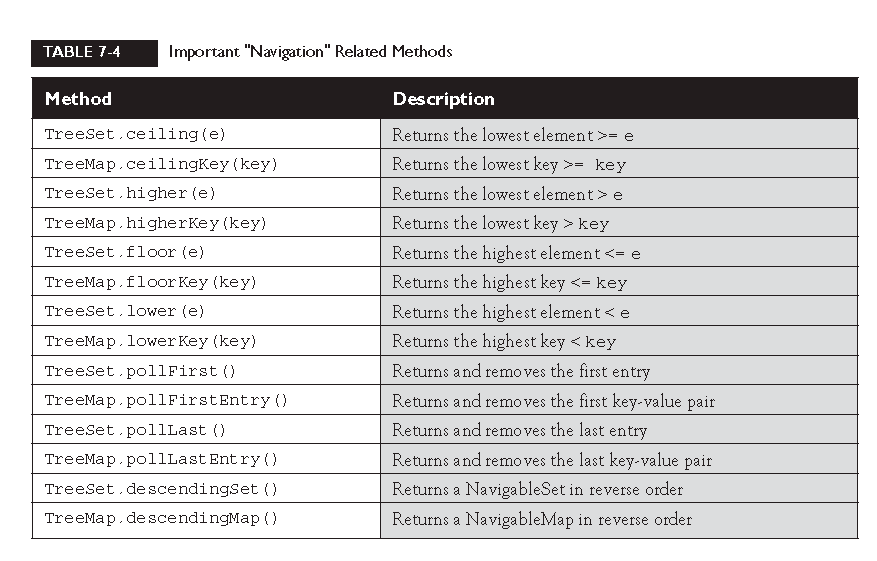
This should produce the following:

J5 - last before 4pm is: 1545

J5 - first after 8pm is: 2010

J6 - last before 4pm is: 1545

J6 - first after 8pm is: 2010



**Backed Collections**

Create backup of collections and by below method we can perform Backed Collections.

**TreeSet and the TreeMap methods. (Important Backed Collection Methods)**

The headSet() / headMap() methods create a subset that starts at the beginning of the original collection and ends at the point specified by the method's argument.

The tailSet() / tailMap() methods create a subset that starts at the point specified by the method's argument and goes to the end of the original collection.

The subSet() / subMap() methods allow you to specify both the start and end points for the subset collection you're creating.

TreeMap<String, String> map = new TreeMap<String, String>();

map.put("a", "ant");

map.put("d", "dog");

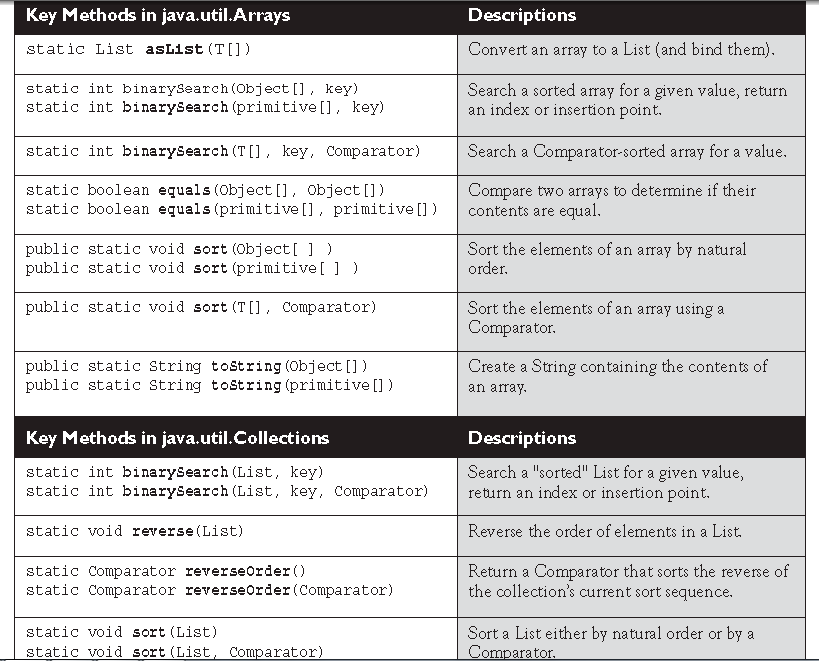
map.put("h", "horse");

SortedMap<String, String> submap;

submap = map.subMap("b", "g");

When we add key-value pairs to either the original TreeMap or the partial-copy SortedMap, the new entries were automatically added to the other collection—sometimes if that key exists in defined range of key in new copy of submap.

submap.put("p", "pig"); // If you attempt to add an out-of-range entry to the copied collection an exception will be thrown.



**Generics**

Generics add stability to our code by making more of our bugs detectable at compile time.

Code that uses generics has many benefits over non-generic code:

* Stronger type checks at compile time.
* Elimination of casts.
* We can implement generic algorithms that work on collections of different types. Type parameters provide a way for us to re-use the same code with different inputs.

**Singleton class (Singleton design pattern)**

A singleton class is a class that can have only one object at a time.

The Singleton's purpose is to control object creation, limiting the number of objects to one only.

Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields. Singletons often control access to resources such as database connections or sockets.

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

**Why we need singleton class?**

1. The Singleton pattern is used in the design of logger classes. These classes are usually implemented as singletons, and provide a global logging access point in all the application components without being necessary to create an object each time a logging operation is performed.
2. To make utility classes.

**How to create singleton class?**

1. Make constructor private.
2. Create static reference of class.
3. Create static getInstance() method that will check and create only one instance of that class.

public class ClassicSingleton {

private static ClassicSingleton instance = null;

private ClassicSingleton() {

}

public static ClassicSingleton getInstance() {

if(instance == null) {

instance = new ClassicSingleton();

}

return instance;

}

}

**Why you need Double checked Locking of Singleton Class?**

One of the common scenarios, where a Singleton class breaks its contracts is multi-threading. If you ask a beginner to write code for Singleton design pattern, there is good chance that he will come up with something like below:

private static Singleton \_instance;

public static Singleton getInstance() {

if (\_instance == null) {

\_instance = new Singleton();

}

return \_instance;

}

***With Double check follow below code.***

public static Singleton getInstanceDC() {

if (\_instance == null) { // Single Checked

synchronized (Singleton.class) {

if (\_instance == null) { // Double checked

\_instance = new Singleton();

}

}

}

return \_instance;

}

**Cloning a Singleton class**

We can clone a Singleton class instance in a scenario where the singleton class extends from a class which implements Cloneable interface and provides implementation of clone() method. So now we can clone the instance by calling the Object class's clone() method on the singleton instance.

**Prevent cloning of singleton class**

We can override the Object class's clone() method to throw the **CloneNotSupportedException** exception.

public Object clone() throws CloneNotSupportedException {

throw new CloneNotSupportedException();

}

**Immutable class**

Immutable class is a class which once created; its contents cannot be changed. Immutable objects are the objects whose state cannot be changed once constructed.

In Short: The class whose instance cannot be modified

**Example: String & all java wrapper classes.**

**Benefits of making a class immutable**

1. are automatically thread-safe and have no synchronization issues
2. allow hash Code to use lazy initialization, and to cache its return value
3. make good Map keys and Set elements (these objects must not change state while in the collection)

**Guidelines to make a class immutable**

1. **Mark the class final so no one can override it.**
2. **Mark all the fields private and final**
3. **Do not provide any setter methods**

public final class ImmutableClass

{

private final Integer immutableField1;

private final String immutableField2;

public ImmutableClass(Integer fld1, String fld2)

{

this.immutableField1 = fld1;

this.immutableField2 = fld2;

}

public Integer getImmutableField1() {

return immutableField1;

}

public String getImmutableField2() {

return immutableField2;}

}

**What is a serialVersionUID and why should I use it?**

The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that, are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in anInvalidClassException.

**What happens if an object is serializable but it includes a reference to a non-serializable object?**

Then a ‘NotSerializableException’ will be thrown at runtime.

**Are the static variables saved as the part of serialization?**

No. The static variables belong to the class and not to an object they are not the part of the state of the object so they are not saved as the part of serialized object.

**What will be the value of transient variable after de-serialization?**

It’s default value.

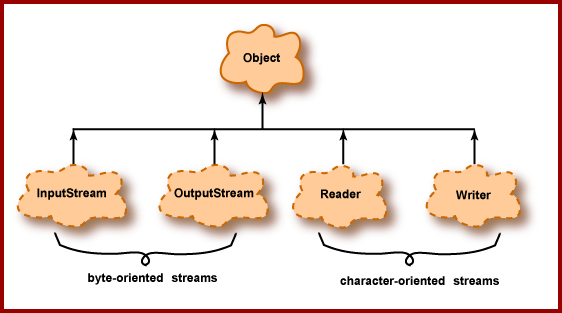
**Externalizable interface**

The **Externalizable** interface provides the necessary means for implementing a custom serialization mechanism.

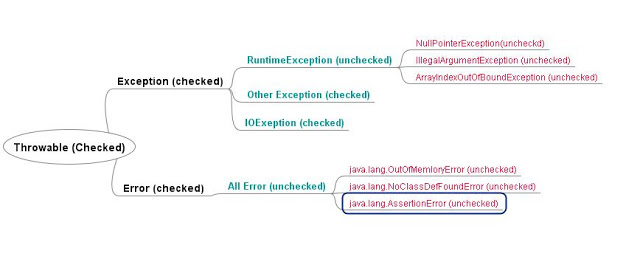
Implementing the **Externalizable** interface means that we must override some of its methods, namely the **writeExternal** and **readExternal** methods.

More here: <http://www.java2blog.com/2014/02/externalizable-in-java.html>

Static methods (in fact all methods) as well as static variables are stored in the PermGen section of the heap; since they are part of the reflection data (class related data, not instance related).



## Difference between Checked vs. Unchecked Exception in Java



In above diagram Error is unchecked exception

More: <http://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/>

**Inner Interface**

Inner interface is also called nested interface, which means declare an interface inside of another interface. For example, the Entry interface is declared in the Map interface.

public interface Map

{

interface Entry

{

int getKey();

}

void clear();

}

**Why Use Inner Interface?**

1. It is a way of logically grouping interfaces that are only used in one place.
2. It increases encapsulation.
3. Nested interfaces can lead to more readable and maintainable code.

**How to prevent your class from being sub classed?**

Make it final or make constructor private

**What are the classes used to read and write entire line from a file?**

BufferedReader and BufferedWriter.

**In which scenario finally block does not execute?**

When system.exit() method is called before finally blocks, application exits and finally won’t get called.

**Explain homogeneous and heterogeneous collections?**

Collection of objects that have a common class is called homogeneous collection. For example

Person[] p = new Person[2];

p[0] = new Person();

p[1] = new Person();

Collection of objects that have different classes but all having common parent class. For example

Person[] e = new Person[2];

e[0] = new Person();

e[1] = new Employee();

**How to place compiled Java file in a different location?**

javac -d C:\users\dac\classes MyProgram.java

Causes the class files for the classes in the MyProgram.java source file to be saved in the directory C:\users\dac\classes.

**How to create Executable Jar Archive (JAR) File?**

Let us say, we have 2 compiled classes Employee and Person inside the folder ALL which resides inside the folder Java.

Employee class contains static void main method.

Before creating a Jar file, first create a main class file in the Java folder.

Specify the name of the class in which main methods resides. Here (“Main-Class: All.Employee”) and press enter.

Open command prompt and navigate to Java Folder.

Enter the following command in the command prompt

“jar cmf mainclass myfirst.jar All/\*.class”

And press enter. Executable jar will create.

To test the newly created jar, use following command “java -jar myfirst.jar”

**Why to use an anonymous inner class in java.**

Creating anonymous class is quicker and simple.

Anonymous inner classes are useful when we need to inherit a few properties (only one method) of a superclass and this is not a good idea to take overhead of creating a separate subclass for doing things so simple.

**The reason why method local inner class cannot use local variables of the method**

The local variables of the method are kept on the stack and perish as soon as the method ends. But even after the method ends, the local inner class object may still be alive on the heap. Method local inner class can still use the local variables that are marked final,

**Can you override private or static method in Java?**

Another popular Java tricky question, as I said method overriding is a good topic to ask trick questions in Java. Anyway, you cannot override private or static method in Java, if you create similar method with same return type and same method arguments that's called method hiding.

**Why abstract class have constructor in Java**

Now if we say we cannot create instance of abstract class then why do Java add constructor in abstract class. One of the reason which make sense is, when any class extend abstract class, constructor of sub class will invoke constructor of super class either implicitly or explicitly. This chaining of constructors is one of the reasons abstract class can have constructors in Java. Here is an example Java program, which proves that abstract class can have constructors in Java:

**ClassNotFoundException**: ClassNotFoundException occurs when class loader could not find the required class in class path. So, basically you should check your class path and add the class in the classpath. Is part of Exception(checked)

**NoClassDefFoundError**: This is more difficult to debug and find the reason. This is thrown when at compile time the required classes are present, but at run time the classes are changed or removed or class's static initializes threw exceptions. It means the class which is getting loaded is present in classpath , but one of the classes which are required by this class , are either removed or failed to load by compiler .So you should see the classes which are dependent on this class .

Is part of Error.

**What is the difference between Association, aggregation and composition?**

**Association** is a relationship where all objects have their own lifecycle and there is no owner.

Let’s take an example of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers, but there is no ownership between the objects and both have their own lifecycle. Both can be created and deleted independently.

**Aggregation** is a specialized form of Association where all objects have their own lifecycle, but there is ownership and child objects can not belong to another parent object.

Let’s take an example of Department and teacher. A single teacher cannot belong to multiple departments, but if we delete the department, the teacher object will not be destroyed. We can think about it as a “has-a” relationship.

**Composition** is again specialized form of Aggregation and we can call this as a “death” relationship. It is a strong type of Aggregation. Child object does not have its lifecycle and if parent object is deleted, all child objects will also be deleted.

Let’s take again an example of relationship between House and Rooms. House can contain multiple rooms - there is no independent life of room and any room cannot belong to two different houses. If we delete the house - room will automatically be deleted.

Let’s take another example relationship between Questions and Options. Single questions can have multiple options and option cannot belong to multiple questions. If we delete the questions, options will automatically be deleted.

**Why do you get a ConcurrentModificationException when using an iterator?**

Problem: The java.util Collection classes are fail-fast, which means that if one thread changes a collection while another

thread is traversing it through with an iterator the iterator.hasNext() or iterator.next() call will throw

ConcurrentModificationException. Even the synchronized collection wrapper classes SynchronizedMap and

SynchronizedList are only conditionally thread-safe,

String str = new String(“ABC”); //Wrong. Avoid this because a new String instance

//is created each time it is executed.

**Why use factory pattern or abstract factory pattern?**

Factory pattern returns an instance of several (product Hierarchy) subclasses (like Circle, Square etc), but the calling code is unaware of the actual implementation class.

The calling code invokes the method on the interface for example Shape and using polymorphism the correct

draw() method gets invoked [Refer Q10 in Java section for polymorphism]. So, as you can see the factory pattern

Reduces the coupling or the dependencies between the calling code and called objects like Circle,

**bounded vs unbounded**

Bounded simply means that the queue has a specific capacity that cannot be exceeded. Unbounded is one who capacity can expand

**Types of OutOfMemoryError in Java**

1) Java.lang.OutOfMemoryError: PermGen space

Java.Lang.OutOfMemoryError: PermGen Space occurs when JVM needs to load the definition of a new class and there is no enough space in PermGen

Solution : -XX:PermSize=64M

-XX:MaxPermSize=128M

2) Java.lang.OutOfMemoryError: Java heap space

Java.Lang.OutOfMemoryError: Java heap Space’ occurs when there is no memory available in heap to store new object.

Solution : - Xms16m -Xmx256m

**What is Java Reflection?**

Reflection allows instantiation of new objects, invocation of methods, and get/set operations on class variables dynamically at run time without having prior knowledge of its implementation.

Daemon Thread

There are two types of threads user thread and daemon thread. The daemon thread is a service provider thread. It provides services to the user thread. Its life depends on the user threads i.e. when the entire user threads die, JVM terminates this thread automatically.

**How does ConcurrentHashMap achieves its Scalability?**   
Sometimes this multithreading + collection interview question is also asked as, the difference between ConcurrentHashMap and Hashtable in Java. The problem with [synchronized HashMap](http://java67.blogspot.com/2015/02/how-to-synchronize-hashmap-in-java-with.html) or [Hashtable](http://java67.blogspot.com/2012/08/difference-between-hashmap-and-concurrentHashMap-java-collection.html) was that whole Map is locked when a thread performs any operation with Map.   
  
The java.util.ConcurrentHashMap class solves this problem by using *lock stripping* technique, where the whole map is locked at different segments and only a particular segment is locked during the write operation, not the whole map. The ConcurrentHashMap also achieves it's scalability by allowing lock-free reads as read is a thread-safe operation.  See [here](http://java67.blogspot.com/2012/08/5-thread-interview-questions-answers-in.html) for more advanced multi-threading and concurrency questions in Java.   
  
**What is default size of ArrayList and HashMap in Java?**([answer](http://javarevisited.blogspot.sg/2014/07/java-optimization-empty-arraylist-and-Hashmap-cost-less-memory-jdk-17040-update.html))  
As of Java 7 now, default size of ArrayList is 10 and default capacity of HashMap is 16, it must be power of 2. Here is code snippet from ArrayList  and HashMap class :

// from ArrayList.java JDK 1.7

private static final int DEFAULT\_CAPACITY = 10;

//from HashMap.java JDK 7

static final int DEFAULT\_INITIAL\_CAPACITY = 1 **<<** 4; // aka 16

**Below Topics can be useful:**

1. <http://javarevisited.blogspot.in/2013/03/difference-between-singleton-pattern-vs-static-class-java.html>
2. <http://javarevisited.blogspot.in/2011/12/final-variable-method-class-java.html>
3. <http://stackoverflow.com/questions/9809074/java-difference-between-strong-soft-weak-phantom-reference>
4. <http://stackoverflow.com/questions/2399544/difference-between-inheritance-and-composition>
5. <http://javarevisited.blogspot.in/2012/02/difference-between-throw-and-throws-in.html>

We can reuse existing class functionality by extending the class or y having reference of that class

“has a” relationship also known as aggregation and composition

If container and contained object are strongly associated then its called composition

If container and contained object are weakly associated then its called aggregation

**Is a vs Has a**

If full functionality required from another class then use is a else has a

**Case 1**

**public** **class** TestClass {

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

TestClass t=**new** TestClass();

t.m1('c');

//will print int because byte-short-char-int-float-double

// we can promot char to int

t.m1(10.2);

//CE error because promotion from double to higher not possible

}

**public** **void** m1(**int** a) {

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

}

**public** **void** m1(**float** a) {

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

}

}

**Case 2**

**public** **class** TestClass {

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

TestClass t = **new** TestClass();

t.m1("dm");// print string

t.m1(**new** Object());// print object

t.m1(**null**);// will print string

// because string is child of object class so no need to call parent if

// u r satisfy with child

}

**public** **void** m1(String a) {

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

}

**public** **void** m1(Object a) {

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

}

}

**Case 3**

**public** **class** TestClass {

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

TestClass t = **new** TestClass();

t.m1("dm");// print string

t.m1(**new** StringBuffer());// print StringBuffer

t.m1(**null**);// CE error now both string and stringbuffer are child of object class but not related to each other

// so we get ambiguity problem

}

**public** **void** m1(String a) {

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

}

**public** **void** m1(StringBuffer a) {

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

}

}

**Case 4**

**public** **class** TestClass {

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

TestClass t = **new** TestClass();

Animal a = **new** Animal();

t.m1(a);// print Animal

Monkey m = **new** Monkey();

t.m1(m);// print Monkey

Animal ref = **new** Monkey();

t.m1(ref);// print Animal

// Because overloading work on reference type

}

**public** **void** m1(Animal a) {

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

}

**public** **void** m1(Monkey a) {

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

}

}

**Overriding cases**

**public** **class** TestClass {

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

P p=**new** P();

p.m1();// call parent method

C c=**new** C();

p.m1();// call child method

P ref=**new** C();

ref.m1();// call child method

// Because at compile Parent has m1 method so its compile fine

// But at runtime it will check Child has m1 method or not

// If available then call child else call parent method

}

}

**class** P {

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

System.***out***.println("parent m1");

}

}

**class** C **extends** P {

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

System.***out***.println("child m1");

}

}

If child class overriding one method from parent class but both have different return type like parent return Object but child return String then its valid and its called **covariant return type**.

**class** P {

**public** Object m() {

**return** **null**;

}

}

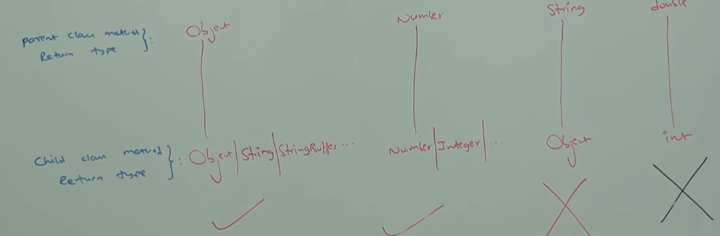
**class** C **extends** P {

**public** String m() {

**return** **null**;

}

}



Below is valid because its called method hiding

**class** P {

**private** **void** m1() {

System.***out***.println("parent m1");

}

}

**class** C **extends** P {

**private** **void** m1() {

System.***out***.println("child m1");

}

}

But below is invalid override because final cannot be overridden

**class** P {

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

System.***out***.println("parent m1");

}

}

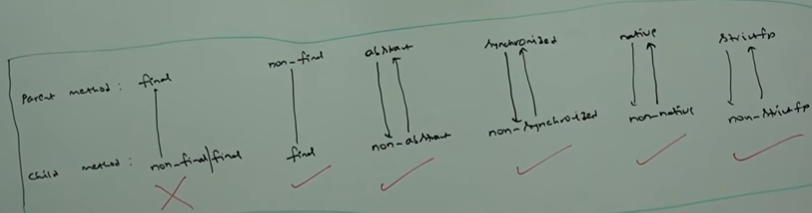
**class** C **extends** P {

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

System.***out***.println("child m1");

}

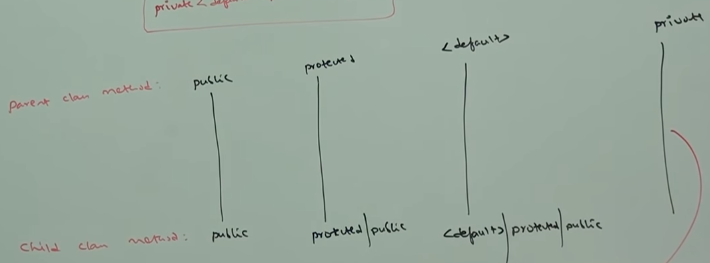
}



While overriding scop of modifiers can not be reduced

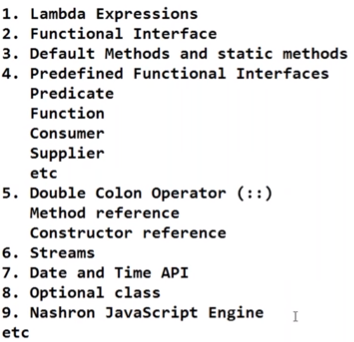
Scope range

Private<default<protected<public



**Java 8**

Now java support functional programming language to reduce line of codes



**Lambda Expression**

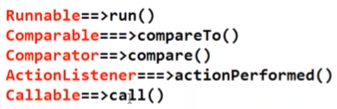
To bring benefits of functional programming lambda exp came in java

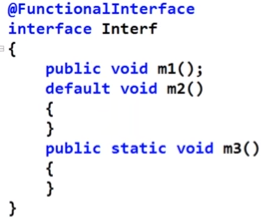
Lambda exp is an anonymous function…used to implement functional interfaces.

LE don’t generate separate .class files

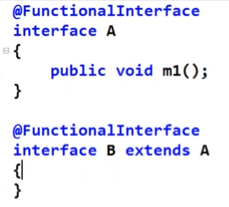
**Functional Interfaces**

An interface having single abstract method is called FI

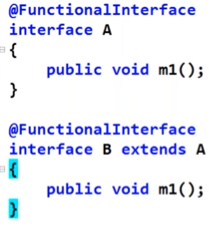




Below is valid

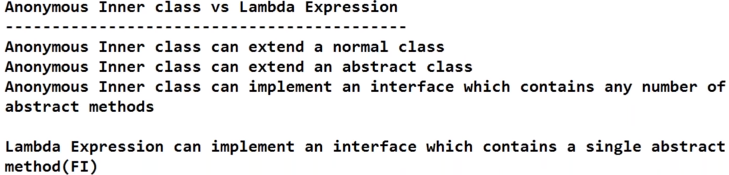


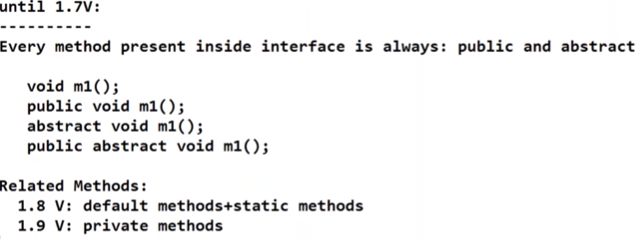
Here below we are overriding m1 of A in B …both having same method so its valid



If interface has one method then we can use both LE and anonymous inner class

But if interface has more than one method then we can use only anonymous inner class to implement those methods



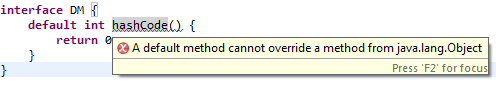


**Default methods**

Without affecting all existing implementation classes if I have to add a new method in interface then I can with help of default methods

All Object class method names below can’t be used as default methods in interface

* public Object()
* public final Class getClass()
* public int hashCode()
* public boolean equals(Object obj)
* protected Object clone() throws CloneNotSupportedException
* public String toString()
* public final void notify()
* public final void notifyAll()
* public final void wait(long timeout) throws InterruptedException
* public final void wait(long timeout, int nanoseconds) throws InterruptedException
* public final void wait() throws InterruptedException
* protected void finalize() throws Throwable



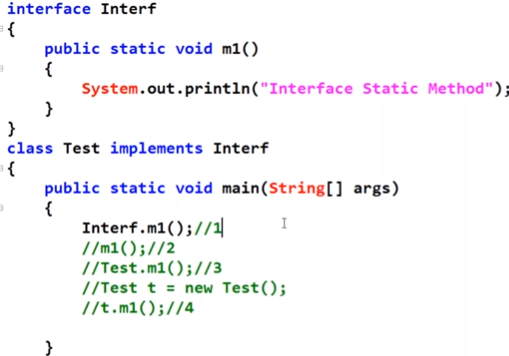
**Static Method**

Purpose is to use as utility method …before 1.8 we were declaring static method in class but static work is related to class not object … so there is no meaning of putting static methods in class …so for better performance we can declare in interface instead of class from 1.8 to improve performance

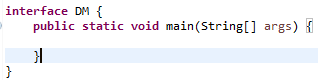
Interface static method by default will not be available to implementation classes

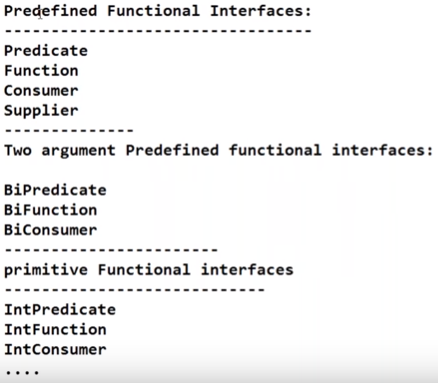
It will be called only by interface name …not by class or object

In below code only first one is valid…if you just want to call static method then we don’t even need to implement interface in class like below



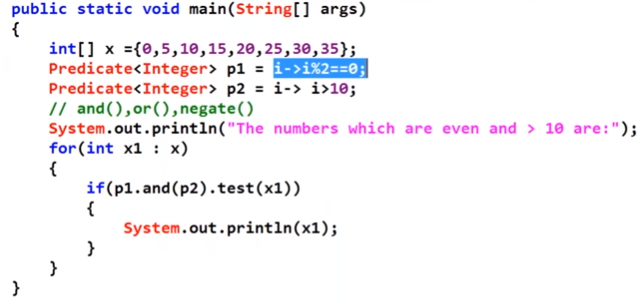
Below is valid we can declare main method in interface from 1.8



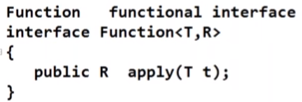


All above are available in **java.util.function** package

Predicate with multi check **and/** **or** condition



P1.negate().test(x1) means not which are not even



What is function composition?

It all has to do with creating small reusable functions that you can combine to compose new functions.

Function<Integer, Integer> times2 = e -> e \* 2;

Function<Integer, Integer> squared = e -> e \* e;

times2.compose(squared).apply(4);

// Returns 32

times2.andThen(squared).apply(4);

// Returns 64

As you can see, the difference between compose and andThen is the order they execute the functions. While the compose function executes the caller last and the parameter first, the andThen executes the caller first and the parameter last.

BiFunction<String, List<Article>, List<Article>> byAuthor =

(name, articles) -> articles.stream()

.filter(a -> a.getAuthor().equals(name))

.collect(Collectors.toList());

BiFunction<String, List<Article>, List<Article>> byTag =

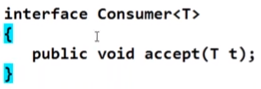
(tag, articles) -> articles.stream()

.filter(a -> a.getTags().contains(tag))

.collect(Collectors.toList());

BiFunction<String, List<Article>, List<Article>> byAuthorSorted =

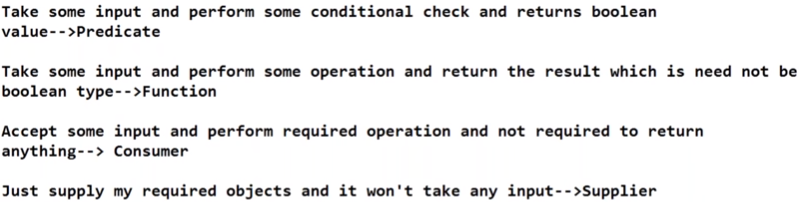
byAuthor.andThen(sortByDate);

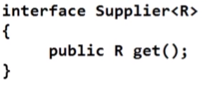


Use Predicate when you have to test or check some condition and return true or false / .test(T t)

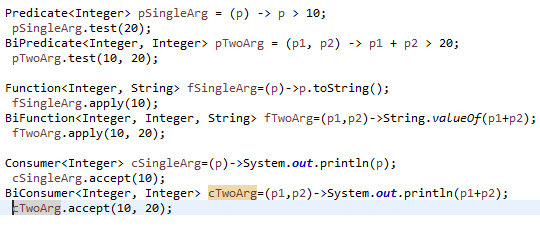
Use Function when you have input and you want some output from that function/ .apply(T t)

Use Consumer when you just want to display print or log something without an output / .accept(T t)





If you have to deal with single argument then use normal Predicate, Function and Consumer but for two arguments you have to use Bi prefix

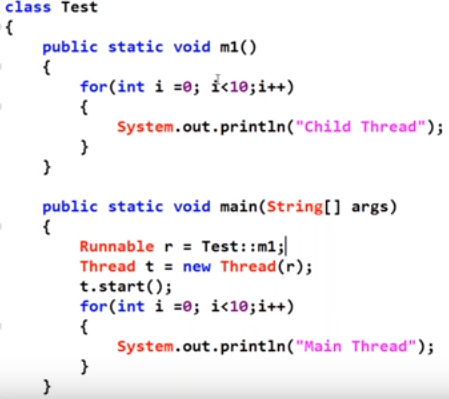


But Supplier do not accept any argument so no concept of Bi prefix

**Method and constructor reference**

Instead of LE we can also use M&C reference to implement FI

Advantage of this approach over LE is reusability of code like in below example in main method we can implement run() method with LE but if we have to use same implementation again in another place in then we had to re-implement same logic again .. so we can avoid duplication by writing separate method and now runnable interface run method implementation will refer to m1 method with :: double colon operator.



If we want to use methed reference then separate method argument parameters must match with FI method parameters like m1() is matching with run() method of Runnable interface

We don’t have to worry about access modifier and return type

### Internal Iterator (diff between normal forloop and forEach)

In internal iterator we have to define **what** to perform during iteration and compiler will do iteration for us in background…like forEach()

names.forEach(name -> System.out.println(name));

In the *forEach*method above, we can see that the argument provided is a lambda expression. This means that the method only needs to know**what is to be done** and all the work of iterating will be taken care of internally.

**External Iterator**

In this iterator we have to define **what** to perform during iteration and also **how** iteration will work

*Enumerations*, *Iterators*and enhanced *for-loop* are all external iterators (remember the methods *iterator(),* *next()* or *hasNext()*? ). In all these iterators it’s our job to specify how the iteration will be performed.

Consider this familiar loop:

for (String name : names) {

System.out.println(name);

}

**Map vs FlatMap**

Both map() and flatMap() takes a mapping function which is applied to each element of a Stream<T>, and returns a Stream<R>. The only difference is that the mapping function in the case of flatMap()produces a stream of new values, whereas for map() it produces a single value for each input element. Arrays.stream(), List.stream(), etc, are commonly used mapping function for flatMap().

Since the mapper function for flatMap() returns another stream, it should result in a stream of streams

**HashMap LoadFactor and Rehashing**

Load Factor decides when to increase the hashmap capacity(buckets) to maintain get/put operation complexity of O(1)

Default load factor of Hashmap is 0.75f (i.e 75% of current map size).

Default capacity of Hashmap is 2^4 = 16 buckets.

Let say we have well implemented hashcode() method, which make sure that key-value pair will be well distributed across 16 buckets equally.

If 16 items in hashmap, then it will distribute 1 item in each bucket. Searching for any item in this case will take only 1 look up.

If 32 items in hashmap, then it will distribute 2 item in each bucket. Searching for any item in this case will take only 2 look up.

If 128 items in hashmap, then it will distribute 1 item in each bucket. Searching for any item in this case will take only 8 look up.

If the amount of item keeps on increasing and the number of buckets are fixed(16) then at one time,

then performance of hashmap will start degrading due to large number of items in each bucket.

Problem is, keeping bucket size fixed(16)

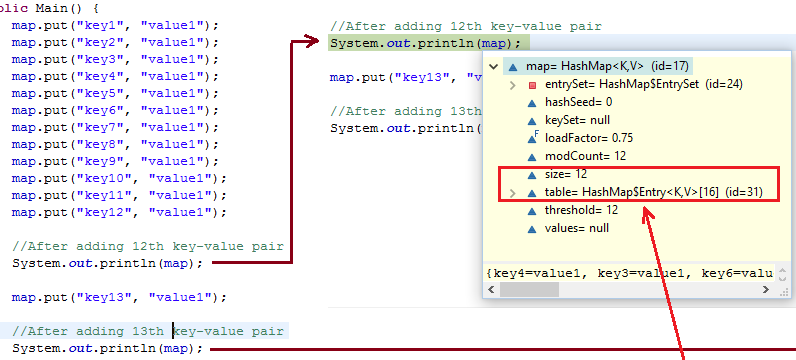
So thats way we have to increase bucket size when 75 % buckets are full

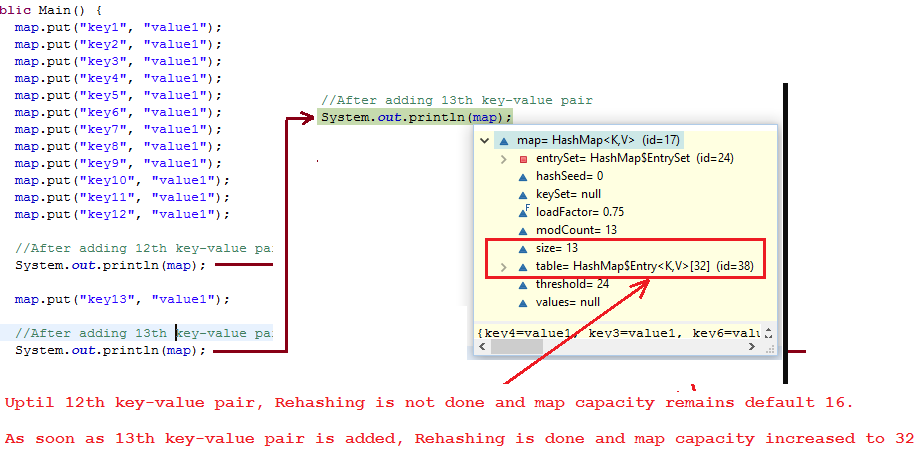
So, when to increase the hashmap size is decided by product of,

(initial capacity of hashmap \* Load factor of hashmap)=16\*0.75=12

This represents that uptil 12th key-value pair hashmap will keep its size to 16 and as soon as 13th item(key-value pair) will come into the Hashmap,

it will increase its size from default 2^4 = 16 buckets to 2^5 = 32 buckets.





**Rehashing** is the process of re-calculating the hashcode of already stored entries (Key-Value pairs), to move them to another bigger size hashmap when Load factor threshold is reached.

Why Rehashing is required?

After doubling the capacity, what to do with the key-value pairs already present in buckets?

If we keep the existing key-value pairs as it is, then doubling the capacity may not help,

because O(1) complexity will be achieved only if items are evenly distributed across all buckets.

# User-defined Custom Exception

// A Class that represents use-defined expception

class MyException extends Exception

{

    public MyException(String s)

    {

        // Call constructor of parent Exception

        super(s);

    }

}

// Throw an object of user defined exception

throw new MyException("GeeksGeeks");