# Declarations, Access Control and Object Orientation.

### Explain basic terms of java like Class, Interface, Object, State, Behaviour.

* Java programs are basically a **collection** of **objects** talking to other objects by **invoking** each others **methods**. Every object is of certain type and that type is determined by the class or interface.
* A **CLASS** is a template that describes the **state** and **behavior** that objects of its type support. When JVM encounters new keyword, it uses appropriate class to make an object (instance of that class).
* Each object has its own set of **instance variable** as defined in the class. The **values** assigned to instance variables of the object constitutes objects **state**.
* An objects **behavior** is determined by its methods. This is where the logic is stored, the real work is done, algorithms executed and data gets manipulated.
* All the java components (class, variables, methods, keywords etc) need names, and these names are called **identifiers**.
* **Inheritance is** one of the core concepts of java that promotes re-usability. A more abstract super class can be extended by specific subclasses.
* **Interfaces** are *100% abstract super classes* that define the methods a subclass must support but not how they must be supported.

### What is the importance organizing your code into packages ?

Java is a **package-centric** language. For good **organization** and name **scoping**, we must put all our classes into packages. If not done...Imagine this nightmare: Three different programmers, in the same company but working on different parts of a project, write a class named Utilities. If those three Utilities classes have not been declared in any explicit package and are in the classpath, there is no way to tell the compiler or JVM which of the three we are trying to reference. Oracle recommends that developers use reverse domain names, appended with division and/or project names. For example, if your domain name is geeksanonymous.com and you’re working on the client code for the TwelvePointOSteps program, you would name your package something like com.geeksanonymous.steps.client. That would essentially change the name of your class to com.geeksanonymous.steps.client.Utilities. You might still have name collisions within your company if you don’t come up with your own naming schemes, but you’re guaranteed not to collide with classes developed outside your company (assuming they follow Oracle’s naming convention, and if they don’t, well, Really Bad Things could happen).

### Explain rules to create legal Identifiers ?

* Legal identifiers must compose of only Unicode characters, numbers, currency symbols, and connecting characters.
* Identifiers must start with a letter, a currency character ($) or a connecting character such as the underscore (\_). Identifiers cannot start with a number.
* There is no limit to number of characters an identifier can contain. You cannot use java keyword as identifier. Identifier in java is case-sensitive.

**Ex : legal Identifiers**

*int \_a; int $c; int \_\_\_\_\_\_2\_w; int \_$;*

*int this\_is\_a\_very\_detailed\_name\_for\_an\_identifier;*

**Ex: Illegal identifiers**

*Int :b; int -d; int e#; int .f; int 7g;*

To summarize

* Identifiers can begin with a letter, an underscore, or a currency character.
* After first character , identifiers can also include digits.
* Identifiers can be of any length.
* Javabeans methods must be named using camelCase, and depending on the methods purpose, must start with set, get, is, add, or remove.

### Explain oracle’s recommended Java code conventions ?

* **Classes and Interfaces** -> the first letter should be capitalized, and if several words are linked together to form the name, first letter of inner words should be uppercase (camelCase). Ex : PrintWriter.
* **Methods** -> first letter should be lowercase and then normal camel case rule should be used. Name should be typical verb-noun pair. Ex: getBalance, doCalculation etc
* **Variables** -> like methods camel case format should be used with first letter being lowercase.
* **Constants** -> java constants are created by marking variables **static** and **final**. They must be named with uppercase letters with underscore character as separator.

**Ex** : MIN\_HEIGHT , MAX\_LENGTH etc

### Explain oracle’s recommended JavaBean standards ?

* Java Bean spec is intended to help java developers create java components that can be easily used. By using naming rules java spec helps guarantee that tools can recognize and use components built by different developers.
* **JavaBeans** are java classes that have **properties**. The methods that change value of the properties are called **setter** methods and the methods that retrieve a properties value are called **getter** methods.
  + If the property is not a Boolean , the getters method prefix must be **get**. Ex, getSize() is valid javabean getter for property named size.
  + If the property is a Boolean, the getter methods prefix is either **get** or **is**. Ex, getStopped() or isStopped() are both valid javabean names for Boolean property.
  + Setter methods prefix must be **set**. Ex, setSize()
  + To complete name of a getter or setter method, change the first letter of property name to uppercase and append it to appropriate prefix (get, set, is)
  + Setter method signature must be marked **public**, with a void return type and an argument that represents the property type.
  + Getter signature must be marked public, take no arguments, and have a return type that mathches the type of the property.
* Java Bean **Listener** naming rules.
  + Javabean specs support events, which allow components to notify each other when something happens.
  + The objects that receive information that a event occurred are called **listeners**.
  + Listener method names used to “register” a listener with an event source must use the prefix add, followed by listener type. Ex, addActionListener()
  + Listener method names used to “unregister” a listener with an event source must use the prefix remove, followed by listener type. Ex, removeActionListener().
  + The type of listener to be added or removed must be passed as the argument to the method.
  + Listener method names must end with the word **Listener**.
  + Examples of valid JavaBean method signatures are

public void setMyValue(int v)

public int getMyValue()

public boolean isMyStatus()

public void addMyListener(MyListener m)

public void removeMyListener(MyListener m)

* + Examples of invalid JavaBean method signatures are

void setCustomerName(String s) // must be public

public void modifyMyValue(int v) // can't use 'modify'

public void addXListener(MyListener m) // listener type mismatch

### Explain source file declaration rules ?

* There can be only **one** public class per source file. Comments can appear anywhere in the source file.
* If there is a public class in the file then the **name of the file** must match name of the public class. If no public class is present then file can have any name.
* **Package** statements must be the first line in source code file(if present), then followed by **import** statements (if present), then the **class** declaration. If no package or import statements present then class declaration must be the first line in the source code file.
* **Import** and **package** statements apply to all classes in the source code file. There is no way to declare multiple classes in one source code file and have them in different package, or use different import.
* Afile can have **multiple** non-public classes.

### What do you mean by Acessing a Class ? what modifiers can be applied to a Class ?

* Access mean **visibility**. If class A has access to class B , it can do one of the things

1. Create instance of class B
2. Extend class B(become subclass of class B)
3. Access certain methods or variables within class B, depending on their access control.

* ***Default Access*** -> A class with default access has no modifier preceding it. A class with default access can be seen only by other class in **same** package.
* ***Public Access*** -> A class with public keyword gives all classes from all packages access to the public class.
* **YOU WILL NEED TO IMPORT** the package of a class, if the class you are trying to access is in a different package , ***even though*** it is a public class.
* NON-Access class modifiers
  + **strictfp –** marking a class as strictfp means that any method code in the class will conform to the IEEE 754 standart rules for floating points. Without that modifier floating point used in methods might behave in a platform dependant way. If a class is not declared strictfp, you can still get strictfp behavior on a method-by-method basis by declaring a method as strictfp.
  + **final** – when used in a class declaration final keyword means the class cannot be subclassed. No other class can ever extend a final class.
  + **abstract –** a abstract class is created with sole intention to subclass. It can never be instantiated. You can compile and execute an abstract class as long as you do not create a instance of it. If a single method in a class is abstract, the whole method becomes abstract.
* You can never mark a class abstract & final at the same time. They both have opposite meaning. A final class can never be subclassed where as an abstract class must be subclasses.

### What constitue class members ?

**Methods** and **instance**(non-local) variables are collectively known as members of a class. You can modify the members with both access & non-access modifiers. Access Modifiers can **never** be applied on **local** variables. But non-access modifier **final** can be applied for local variables.

### What are different categories of Modifiers ?

Modifiers fall into 2 categories

* Access Modifiers (public , private , protected )

In java there are **4** access controls but **3** access modifiers. The fourth access control level (called default or package access) is what you get when none of the acess modifiers are used. Every class , method, instance variable has a access control. A class can be declared with only **public** or **default** access, the other two does not make sense for a class.

* Non-Access Modifiers( strictfp , final , abstract )

### What is an Interface ?

When you declare a interface, you are defining a contract for what a class can do, without saying anything about how the class will do it. An interface is a **contract**. Any class that **implements** an interface must write the code for the interface methods. Interface can be implemmented by any class from any inheritance tree.

* An interface is a 100% abstract class.
* An abstract class can have both abstract and non-abstract methods , where as an interface can have **only** abstract methods.
* You need to use **public** modifier if you want *interface* to have public access rather than default access.
* All interface *methods* are *implicitly* **public** and **abstract**. In other words you need not type abstract in method declaration.
* Interface methods must **not** be static.
* All variables defined in a interface must be **public**, **static** and **final.** Interfaces can declare only constants, not instance variables.
* Because interface methods are abstract they **cannot** be marked with **final**, **strictfp** or **native**.
* An interface can **extend** one or more interfaces. And an interface can extend only other interfaces and nothing else.
* An interface cannot implement another interface or class.
* An interface must be declared with the keyword **interface**. Interface types can be used polymorphically.

### Explain default and static method features in Interface since Java8 ?

**Default Methods** : as of java8, interface can include inheritable methods with concrete implementation.

* Default methods are declared using the **default** keyword. This keyword can only be used with interface method signatures, and not class method signatures.
* Default methods are **public** by definition and public modifier is optional.
* Default methods cannot be marked private, protected, static, final or abstract.
* Default methods must have a concrete method body.

**Static Methods** : as of java8, interface can include static methods with concrete implementation.

* Static methods are declared using the **static** keyword.
* Static methods are **public** by definition and public modifier is optional.
* Static interface method must have concrete method body.
* When **invoking** a static interface method, methods type (interface name) **MUST** be included in invocation.

### Explain various Access modifiers and access levels in Java ?

While a class can use just two of the four access controls(public and default), members can use all the four (public , protected , default, private). If Class A has access to a member of Class B, it means that class B’s member is visible to Class A.

**There are 2 access variations.**

1. Whether method code in one class can access a member of another class ?

(using [**.**] dot operator to invoke a method or retreive a variable)

1. Whether a subclass can inherit a member of its superclass ?

If a subclass inherits a member of its superclass , it is exactly as if subclass actually declared the member itself(the subclass has the member).

Although a instance variable can be marked as public, in practice it is always best to keep it **private** or **protected**. If variable need to be changed, set, or read, you must use public accessor methods(getters & setters).

Rules to Consider to determine access Levels between 2 classes

1. Much of Access control centers on whether the two classes involved are in the **same or different** package. If class A itself cannot be accessed by class B, then no members of class A can be accessed by class B. so make sure the class is visible or not.
2. If a class is visible, when a member is declared **public**,

* it means all other classes, regardless of the package they belong to can access the member.
* For a subclass, it can inherit the member regardless of whether both classes are in same package.

1. If a class is visible, when a member is declared **private**,

* It cannot be accessed by code in any other class other than the class in which it is declared. A private member is invisible to any code outside the member’s own class.
* For a subclass, it cannot inherit the private member of its superclass. It can declare a matching method but it will not be considered **overriding**. It is simple a method that happens to have a same name as a private method(that you are not supposed to know) . this newly-declared-but-just-happens-to-match method can declare new exceptions, return type or any thing else you want it to do.

1. If a class is visible, when a member is declared with no access modifier(**default**)

* It cannot be accessed by code in any other class other than classes in the same package.
* For a subclass, it can inherit the member if it belongs to same package as the parent class.

1. If a class is visible, when a member is declared with **protected**,

* It is same as the default modifier with one difference, a protected member can be accessed(through) ***inheritance*** by a subclass **even if** the subclass is in a **different** **package**.
* Once the sub-class outside the package inherits the protected member it becomes a private member, which means other classes in the same package as the sub-class cannot access this inherited method.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **same class** | **same package-**  **subclass** | **same package-**  **other class** | **other package-**  **subclass** | **other package-**  **other class** |
| **Public** | Yes | Yes | Yes | Yes | yes |
| **Protected** | Yes | Yes | Yes | Yes(only via Inheritance) | No |
| **Default** | Yes | Yes | Yes | No | No |
| **private** | Yes | No | No | No | No |

### Explain various Non-Access Member Modifiers applied to class members ?

non-Access modifiers that can be applied to class members are,

1. ***final***

* when final is applied to **class** , class cannot be subclassed or inherited.
* When final is applied to a **method**, it cannot be overridden in a subclass and is often used to enforce API Functionality of a method(many of methods in core class libraries are marked as final).
* When final is applied to **arguments of a method,** it means that it cannot be modified further within the method. A final argument must keep the same value that the parameter had when it was passed into the method.
* Declaring a variable with final keyword makes it impossible to reinitialize that variable once it has been initialized with an explicit value.
* A reference variable marked final can’t ever be reassigned to refer to a different object. The data within the object can be changed but the reference variable cannot be changed. There are no final objects only final references.

1. ***abstract***

* an abstract method is a method that’s been declared(as abstract) but **not** **implemented**. The method does not contain any funtional code. If a class contains atleast one abstract method the class also must be declared abstract. You can however have an abstract class with no abstract methods.
* The first concrete subclass(means non-abstract) of a abstract class must implement all abstract methods of the superclass.
* A method can never be marked as both (**abstract** and **final**) or both (**abstract** and **private**) or both (**abstract** and **static**).
* An abstract designation means superclass doesnot know anything about how the subclasses should behave in that method, whereas a final designation means the superclass knows everything about how all subclasses(however far down the inheritance tree they mab be) should behave in that method.

1. ***Transient***

* If you mark a instance variable **transient ,** you are telling JVM to skip this variable when you attempt to serialize the object containing it. It can be applied **only** to instance variable.

1. ***synchronized***

* a synchronized keyword indicates that a method can be accessed by **only one thread** at a time. synchronized keyword can be applied only to methods, not classes, not variables, only methods.

1. ***Native***

* **native** modifier also can be applied only to methods, not classes or variables. native modifier indicates that a method is implemented in platform-dependent code, often in C. native methods body must be( ; ) indicating that implementation is omitted(like abstract methods).

1. ***Strictfp***

* **Strictfp** modifier can be applied to class or methods. Strictfp forces floating point to adhere to IEEE754 standard. A variable can never be declared strictfp.

1. ***Static***

A static modifier is used to create variables and methods that will exist **independantly** of any instances created for the class. All static members exist before you ever make a new instance of a class, and there will be **only one copy** of a static member regardless of the number of instances of that class. All instances of a given class **share** the same value for any given static variable.

You can mark following with static

* Methods
* Instance Variables
* A class nested within another class, but not within a method.
* Initialization blocks.

Following things cannot be marked static

* Constructors
* Classes(unless they are nested)
* Interfaces
* Method local inner classes
* Inner class methods and instance variables
* Local variables

Variables and methods marked static belong to the class, rather than to a particular instance. We can use a static method or a variable without having any instance of that class. A static variable will be **shared** among all the instaces of the class, which means there is only **one** copy for that class.

* A static method cannot access a non-static (instance) variable , and cannot directly invoke a non-static method. This is because it is **unaware** of the actual object to which this variable or method belong to. Hence you need to refer to instance variable or method using the specific object and the (.) operator. Think static = class and nonstatic = instance.
* Making the method called by the JVM (**main**()) a static method means the JVM doesn't have to create an instance of your class just to start running code.
* In-order to access a static method or static variable, we use the dot (.) operator on the **class** **name**. Java also allows you to use an object reference variable to access a static member.
* Static methods cannot be overridden.

1. ***volatile***

* 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. It can be applied only to instance variable.

### Explain what is Variable Argument List ?

Java allows you to create methods that can take a variable number of arguments.

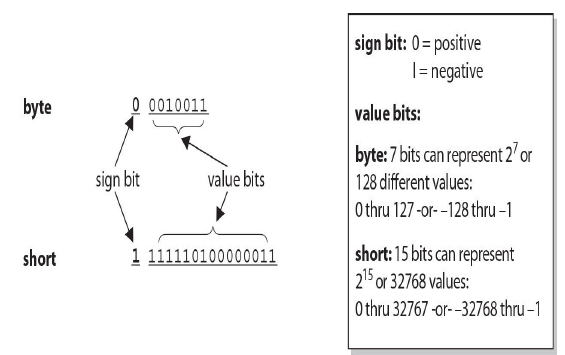
This capability is referred as "variable-length argument lists," "variable arguments," "var-args," "varargs", "variable arity parameter."

* When you declare a var-arg parameter, you must specify the type of argument(s) this parameter of your methods can receive.
* To declare a method using a var-arg parameter, you follow the type with an ellipsis(…) , a space, and then the name of the array that will hold the parameters passed.
* You can have other parameters in a method that uses var-args.
* The var-args must be the last parameter in the methods signature, and you can have only one var-arg in a method.

### What are various types of Variables in java?

There are 2 types of variables in Java.

* **Primitives :** a primitive variable can be one of eight types : char, byte, short, int, long, float, double, boolean. All number types (integers, floating-point types) are signed. All six number types are made up of a certain number of 8-bit bytes. The leftmost bit(most significant digit) is used to represent the sign. 1 means negative and 0 means positive.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Bits** | **Bytes** | **Min Range** | **Max Range** |
| byte | 8 | 1 | -2(8-1) | 2(8-1)-1 |
| short | 16 | 2 | -215 | 215-1 |
| Int | 32 | 4 | -231 | 231-1 |
| Long | 64 | 8 | -263 | 263-1 |
| Float | 32 | 4 | n/a | n/a |
| double | 64 | 8 | n/a | n/a |

* **Reference variable :** a reference variable is used to refer to (or access) an object. A reference variable is declared to be of a specific type and that type can never change.

### What is difference between instance and local variables ?

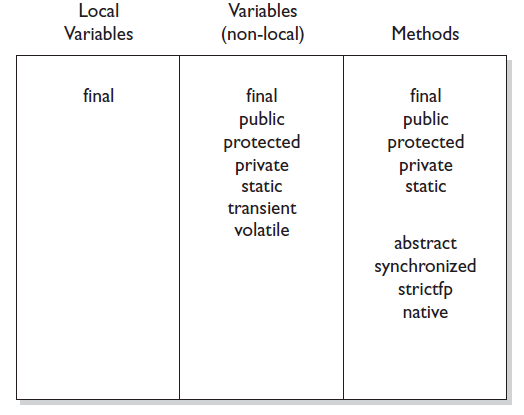
**Instance variables** are defined inside the class, but outside of any method, and are initialized only when class is instantiated. Instance variables are fields that belong to each object.

Following are key points to remember, Instance variables

* Can use any of the 4 access levels.
* Can be marked with final.
* Can be marked transient.
* Cannot be marked abstract.
* Cannot be marked synchronized.
* Cannot be marked strictfp.
* Cannot be marked native.
* Cannot be marked static, because then they would become class variables.

**Local** **variables** are variables declared within a method. Local variable starts its life when method starts and ends when method finishes executing. Local variables are always on the stack not the heap. Even if local variable is an object reference, the object itself will still be created on the heap. There is no such thing as stack object only stack variable.

It is possible to to declare a local variable with the same name as an instance variable. It is called as **shadowing**.



### What is an Enum ?

you can create a list of pre-defined values (or enumerated list) to which you can restrict your variable to be assigned. The items in the enumerated list are called **enums**.

Ex: enum CoffeeSize {BIG, HUGE, OVERWHELMING};

CoffeeSize cs = CoffeeSize.BIG;

* Enums can be declared as their **own** separate class, or as a class **member**, but **not** within a member.
* The enum that isn’t enclosed within a class can be declared with **only** the public or default modifier.
* Every enum has a static method called values(), which returns an array of the enum’s values in the order they are declared.
* You can never invoke a enum constructor directly. You can define more than one argument to the enum constructor or overload a enum constructor.



# Object Orientation, JVM and Java memory model.

### What is Encapsulation ?

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interferance and misuse. It is like a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and the data inside the wrapper is tightly controlled through a well defined interface.

Encapsulation is a must if you want your java programs to be maintainable, flexible and extensible.

In order to achieve encapsulation :

* Keep instance variables **protected**(with an access modifier, often **private**)
* Make public **accessor** methods, and force calling code to use those methods rather than directly accessing the instance variable. Access methods are called with different names like getters and setters , accessors and mutators.
* For the methods, use the javaBeans naming conventions of set<someProperty> and get<someProperty>.

### Explain Inheritance(Is-A) and object composition (Has-A) ?

Inheritance is everywhere in java. It is impossible to write even the tiniest java program without using Inheritance. Every java class is a subclass of class **Object(except class object itself)**.

The two important reasons to use inheritance are

* To promote code reuse
* To use polymorphism

*Is-A*

In OO the concept of IS-A is based on class inheritance or interface implementation. IS-A is a way of saying, “*this this is a type of that thing*”.

Ex : mustang IS-A type of horse, Broccolli IS-A vegetable, Subaru IS-A car.

IS-A relationship in java is expressed through the keyword **extends** (for class inheritance) and **implements**(for interface implementation).

*Has-A*

Has-A relationships are based on usage, rather than inheritance. In other words , class A HAS-A B if code in class A has a reference of class B.

Ex : humans IS-A type of animal, HAS-A car.

### Explain polymorphism ? What are object references ?

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.

* 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 re-assigned to other objects,(unless the object is declared final).
* A reference **variables type determines** that 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.
* Polymorphic method invocations apply only to **instance** methods. You can always refer to an object with a more general reference variable type (a superclass or interface), but at runtime, the **ONLY** things that are dynamically selected based on the actual object (rather than the reference type) are **instance methods**. Not static methods. Not variables. **Only** **overridden** instance methods are dynamically invoked based on the real object's type.

### What is Reference Variable casting ?

There are two types of reference variable casting: downcasting and upcasting.

* **Downcasting**: If you have a reference variable that refers to a subtype object, you can assign it to a reference variable of the subtype. You must make an explicit cast to do this, and the result is that you can access the subtype's members with this new reference variable.
* **Upcasting**: You can assign a reference variable to a supertype reference variable explicitly or implicitly. This is an inherently safe operation because the assignment restricts the access capabilities of the new variable.

### Explain the difference between method overriding and overloading ?

*Overriding*

During inheritance you can override a method from superclass(unless method is declared final). The key benefit of overriding is the ability to define behavior that’s specific to a particular subclass type.

* Compiler looks at the reference type, not the instance type. Polymorphism lets us use a more abstract supertype(including a interface) reference to refer to one of its subtypes(including interface implementers).
* The overridden method cannot have a more **restrictive** access modifier than the method being overridden.
* Java Runtime Polymorphism, is where java uses virtual method invocation to dynamically select the actual version of the method that will run, based on the actual instance.
* You can invoke the superclass version of the overridden method using **super** keyword.
* **@Override** annotation can be used to help catch error with overriding or implementing at compile time. If you intend to override a method in a superclass or implement a method in an interface, you can annotate that method with @Override.

The rules for overriding a method are :

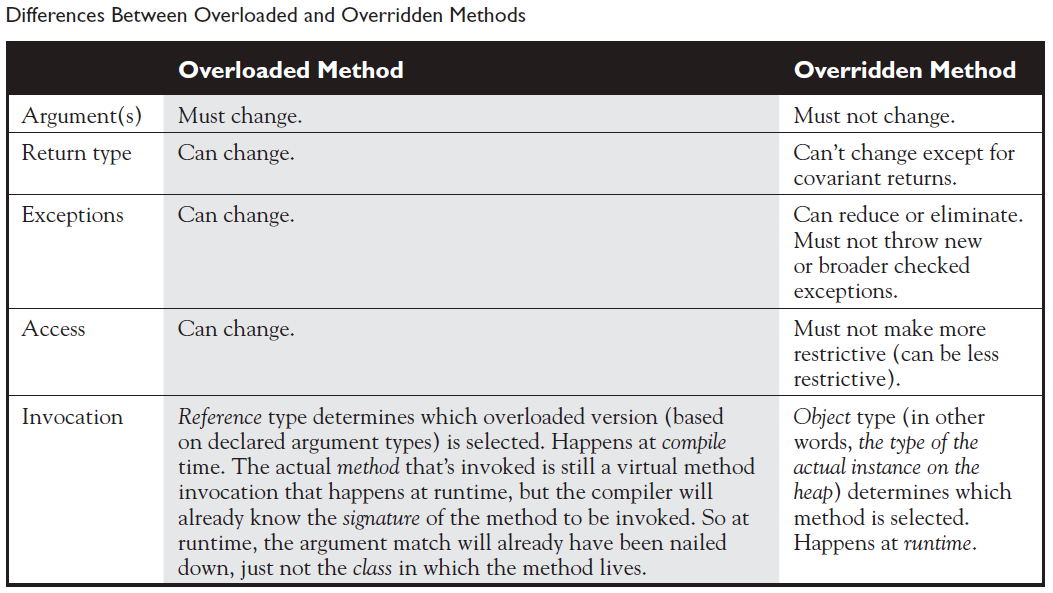
* The **argument list** must exactly match that of the overridden method. If they don't match, you can end up with an overloaded method you didn't intend.
* The **return** type must be the same as, or a subtype of, the return type declared in the original overridden method in the superclass.
* The access level can't be **more** restrictive than the overridden method's.
* The access level CAN be **less** restrictive than that of the overridden method.
* Instance methods can be overridden only if they are inherited by the subclass. A subclass within the same package as the instance's superclass can override any superclass method that is not marked private or final. A subclass in a different package can override only those non-final methods marked public or protected (since protected methods are inherited by the subclass).
* The overriding method **CAN** throw any unchecked (runtime) exception, regardless of whether the overridden method declares the exception.
* The overriding method must **NOT** throw checked exceptions that are new or broader than those declared by the overridden method. For example, a method that declares a FileNotFoundException cannot be overridden by a method that declares a SQLException, Exception, or any other non-runtime exception unless it's a subclass of FileNotFoundException.
* The overriding method can throw **narrower** or fewer exceptions. Just because an overridden method "takes risks" doesn't mean that the overriding subclass' exception takes the same risks. Bottom line: an overriding method doesn't have to declare any exceptions that it will never throw, regardless of what the overridden method declares.
* You cannot override a method marked **final** or **static**.
* If a method **cannot** be inherited , you **cannot** override it.

*OverLoading*

Overloaded methods let you **reuse** the same method **name** in a class, but with different arguments. The rules are..

* Overloaded methods MUST change the argument list.
* Overloaded methods CAN change the return type.
* Overloaded methods CAN change the access modifier.
* Overloaded methods CAN declare new or broader checked exceptions.
* A method can be overloaded in the same class or in a subclass. In other words, if class A defines a doStuff(int i) method, the subclass B could define a doStuff(String s) method without overriding the superclass version that takes an int. So two methods with the same name but in different classes can still be considered overloaded, if the subclass inherits one version of the method and then declares another overloaded version in its class definition.
* Deciding which of the overloading methods to invoke is based on the arguments. And the reference type of the argument(not the object type) determines which overloaded method is invoked.

*To summarize, which* ***overridden*** *version of the method to call (in other words, from which class in the inheritance tree) is decided at* ***runtime*** *based on* ***object*** *type, but which* ***overloaded*** *version of the method to call is based on the* ***reference*** *type of the argument passed at* ***compile*** *time.*



### What is a constructor ? what are the rules for instantiating an object ?

In java objects are constructed. Everytime a new object is made atleast one constructor is invoked.

* A constructor must have **same name** as the class in which they are declared.
* Constructors can never, ever, ever ,ever have a return type.
* Constructors can have all of the normal access modifiers.
* Constructors cannot be marked as static, final or abstract.

Objects are constructed. You can’t make a new object without invoking a constructor. In fact, you can't make a new object without invoking not just the constructor of the object's actual class type, but also the constructor of each of its superclasses! Constructors are the code that runs whenever you use the keyword **new**.

* Every class **including** abstract classes must have a constructor.
* Constructors can use any access modifier, including private. (A private constructor means only code within the class itself can instantiate an object of that type, so if the private constructor class wants to allow an instance of the class to be used, the class must provide a static method or variable that allows access to an instance created from within the class.)
* The constructor name must match the name of the class.
* Constructors must not have a return type.
* It's legal (but stupid) to have a method with the same name as the class,but that doesn't make it a constructor. If you see a return type, it's a method rather than a constructor. In fact, you could have both a method and a constructor with the same name—the name of the class—in the same class, and that's not a problem for Java. Be careful not to mistake a method for a constructor—be sure to look for a return type.
* if you don't type a constructor into your class code, a default constructor will be automatically generated by the compiler.
* The default constructor is ALWAYS a no-arg constructor.
* If you want a no-arg constructor and you've typed any other constructor(s) into your class code, the compiler won't provide the no-arg constructor (or any other constructor) for you. In other words, if you've typed in a constructor with arguments, you won't have a no-arg constructor unless you type it in yourself!
* Every constructor has, as its first statement, either a call to an overloaded constructor (this()) or a call to the superclass constructor (super()), although remember that this call can be inserted by the compiler.
* If you do type in a constructor (as opposed to relying on the compiler-generated default constructor), and you do not type in the call to super() or a call to this(), the compiler will insert a no-arg call to super() for you, as the very first statement in the constructor.
* A call to super() can be either a no-arg call or can include arguments passed to the super constructor.
* A no-arg constructor is not necessarily the default (i.e., compiler-supplied) constructor, although the default constructor is always a no-arg constructor.
* The default constructor is the one the compiler provides! While the default constructor is always a no-arg constructor, you're free to put in your own noarg constructor.
* You cannot make a call to an instance method, or access an instance variable, until after the super constructor runs.
* Only static variables and methods can be accessed as part of the call to super() or this(). (Example: super (Animal.NAME) is OK, because NAME is declared as a static variable.)
* Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated.
* Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree.
* The only way a constructor can be invoked is from within another constructor. In other words constructors cannot be called from any other methods apart from constructors.
* The first call in the constructor must be a call to **super()** or a call to **this()** (calling another constructor in same object). Constructor can never have both calls together.

### What is an interface ? how are rules around implementing interface ?

* Interfaces are contracts for what a class can do, but they say nothing about the way in which the class must do it. Interfaces can be implemented by any class, from any inheritance tree. One class can implement many interfaces.
* An interface is like a 100-percent abstract class, and is implicitly abstract whether you type the abstract modifier in the declaration or not.
* An interface can have only abstract methods, no concrete methods allowed.
* Interface methods are by default public and abstract—explicit declaration of these modifiers is optional.
* Interfaces can have constants, which are always implicitly public, static, and final.
* Interface constant declarations of public, static, and final are optional in any combination.
* A legal nonabstract implementing class has the following properties:

❑ It provides concrete implementations for the interface's methods.

❑ It must follow all legal override rules for the methods it implements.

❑ It must not declare any new checked exceptions for an implementation method.

❑It must not declare any checked exceptions that are broader than the exceptions declared

in the interface method.

❑ It may declare runtime exceptions on any interface method implementation regardless

of the interface declaration.

❑ It must maintain the exact signature (allowing for covariant returns) and return type

of the methods it implements (but does not have to declare the exceptions of the

interface).

* A class implementing an interface can itself be abstract.
* An abstract implementing class does not have to implement the interface methods (but the first concrete subclass must).
* A class can extend only one class (no multiple inheritance), but it can implement many interfaces.
* Interfaces can extend one or more other interfaces.
* Interfaces cannot extend a class, or implement a class or interface.
* When taking the exam, verify that interface and class declarations are legal before verifying other code logic.

### Does java allow multiple inheritance ?

Prior to java8.. NO !!

But now post java8 as interfaces can have concrete methods and classes can implement multiple interfaces, this might lead to the deadly **Diamond of Death** as interfaces can have duplicate concrete method signatures. But good news is that compiler will handle this by **forcing you to provide concrete implementation by overriding** conflicting method.

### Explain rules around valid return types in Java ?

- Overloaded methods can change return types; overridden methods cannot, except in the case of covariant returns. Object reference return types can accept **null** as a return value.

- An array is a **legal** return type, both to declare and return as a value.

- For methods with primitive return types, any value that can be **implicitly** **converted** to the return type can be returned.

- Nothing can be returned from a **void**, but you can return nothing. You're allowed to simply say return, in any method with a void return type, to bust out of a method early. But you can't return nothing from a method with a non-void return type.

- Methods with an object reference return type, can return a subtype (**CoVariant** return).

- Methods with an interface return type, can return any implementer.

### What is an Immutable class and how to create one ?

Immutable class means that once an object is created, we **cannot** change its **content (object variables)**. In Java, all the wrapper classes (like String, Boolean, Byte, Short) and String class is immutable.

They are inherently **thread-safe** and hence are userful in concurrent/parallel applications.

We can create our own immutable class as well following below guidelines,

* Mark your **class final** so that it cannot be extended and modified.
* Mark the **variables** private and **final**.
* If the constructor takes any mutable object as arguments, make **new copies** of those objects and assign the new object within the constructor.
* **Do not** provide any **setter** method.
* If the getter methods return a mutable object reference, make a copy of actual object and **return** reference of the **copy**.

### What is Coupling and Cohesion ? what is the importance of these ?

coupling and cohesion, have to do with the quality of an OO design. In general, good OO design calls for loose coupling and shuns tight coupling, and good OO design calls for high cohesion, and shuns low cohesion. As with most OO design discussions, the goals for an application are

* Ease of creation
* Ease of maintenance
* Ease of enhancement

**Coupling**

Coupling is the **degree** to which one class knows about another class. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be **loosely** coupled…that's a good thing. If, on the other hand, class A relies on parts of class B that are not part of class B's interface, then the coupling between the classes is tighter…not a good thing. In other words, if A knows more than it should about the way in which B was implemented, then A and B are **tightly** coupled. Poor encapsulation causes tight coupling. An aspect of good class and API design is that class should be well encapsulated.

**Cohesion**

Contrary to coupling, cohesion is all about how a single class is designed. The term cohesion is used to indicate the degree to which a class has a single, well-focused purpose. The more focussed the class is **higher** its cohesiveness(good thing). The key benefit of higher cohesiveness is that such classes are much easier to maintain(and less frequently changed) than classes with low cohesion.

### What are Initialization Blocks ?

two places in a class where you can put code that performs operations: methods and constructors. Initialization blocks are the third place in a Java program where operations can be performed. Initialization blocks run when the class is first loaded (a **static** initialization block) or when an instance is created (an **instance** initialization block).

* A static initialization block runs when the *class is first loaded*.
* An instance initialization block runs once *every time a new instance is created*.

The order of execution

* Init blocks execute in the **order** they appear.
* Static init blocks run once, when the class is **first** loaded.
* Instance init blocks run every time a class instance is **created**.
* Instance init blocks run **after** the constructor's call to super().

### What is the effect of using static modifier and list its benefits ?

* Use static methods to implement behaviors that are **not affected** by the state of any instances.
* Use static variables to hold **data that is class specific** as opposed to instance specific—there will be only one copy of a static variable.
* All static members belong to the class, not to any instance. A static method can’t access an instance variable directly. Use the dot operator to access static members, but remember that using a reference variable with the dot operator is really a syntax trick, and the compiler will substitute the class name for the reference variable; for instance:

d.doStuff();

becomes

Dog.doStuff();

* To invoke an interface’s static method, use **MyInterface.doStuff()** syntax.
* static methods can’t be overridden, but they can be redefined.

### What is Java Byte Code ?

|  |  |
| --- | --- |
| **Virtual machine** is a software implementation of physical machine. Java was developed with concept of WORA(**W**rite **O**nce and **R**un **A**nywhere).  **Java compiler** compiles java source file(.java) into ***bytecode***(.class) files, which is the instruction set for the java virtual machine. **JRE** (Java Runtime Environment) is the implementation of java virtual machine(JVM), which analyzes the bytecode, **interprets** the code and executes it.  JVM is platform specific and shipped independently and installed to local system. Same bytecode is interpreted to execute in platform specific way, which essentially is the trick behind achieving platform independence. | Java Bytecode |

### Explain JVM Architecture ?

|  |  |
| --- | --- |
| **JVM** is divided into 3 main subsystems.   1. ***Class Loader*** : java’s dynamic class loading functionality is handled by the class loader subsystem. It loads, links and initializes the class file when it refers to a class for the **first** **time** at **runtime** (not compile time).  * Loading :   classloaders follow **Delegation** **Hierarchy** algorithm.   * **Bootstrap** class loader is responsible for loading classes from the bootstrap classpath (classes in rt.jar), and highest priority is given to this loader. * **Extension** class loader is responsible for loading classes from inside ext folder (jre\lib). * **Application** class loader is responsible for loading application level classpath, path mentioned in environment variable etc. * Linking : * **Verify** bytecode verifier will verify whether the generated bytecode is proper or not and throw appropriate error if not. * **Prepare** for all static variables memory will be allocated and assigned with default values. * **Resolve** all symbolic memory references are replaced with the original references from method area. * Initialization : in this phase of class loading all static variables will be **assigned** with original values and static **block** will be executed. | JVM Architecture Diagram |

1. ***Runtime Data Area*** : this area is divided into 5 major components,

* **Method Area** all class level data is stored here, including static variables. There is only one method area per JVM and it is a shared resource.
* **Heap Area** all the objects and their corresponding instance variables and arrays will be stored here. There is also one heap area per JVM. Since method and heap area share memory for multiple threads, the data stored is not thread safe.
* **Stack Area** for every Thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called as stack frame. All local variables will be create in the stack memory. The stack area is thread-safe since it is not a shared resource. The stack frame is divided into 3 sub entities.
  + *Local variable array – R*elated to the method, how many local variables are involved and the corresponding values will be stored.
  + *Operand stack –* If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
  + *Frame data –* All symbols corresponding to the method is stored here. In the case of any exception, the catch block information will be maintained in the frame data.
* **PC Registers** each threads will have separate PC registers, to hold the address of current executing instruction, and once instruction gets executed, it will be updated with the next instruction.
* **Native method stacks** this holds native method information. For every thread, a separate native method stack will be created.

1. ***Execution engine :*** the bytecode that is assigned to the runtime data area will be executed by the Execution engine. It reads the bytecode and executes it piece by piece.

* **Interpreter** it translate a .class file into code that can be executed natively on the underlyingmachine**.** the interpreter interprets the bytecode faster, but executes slowly. Disadvantage of interpreter is that when one method is called multiple times, everytime a new interpretation is required.
* **JIT Compiler** the JIT compiler neutralizes the disadvantage of the interpreter. Execution engine uses interpreter in converting bytecode but when it finds repeated code it uses the **JIT Compiler**, which compiles the entire bytecode and changes it to **native code**. This native code will be used directly for repeated method calls which improves performance of system. JIT compiler composes of,
  + *Intermediate code generator –* Produces intermediate code.
  + *Code optimizer –* responsible for optimizing the intermediate code generated.
  + *Traget code generated –* responsible for generating machine code or native code.
  + *Profiler –* a special component responsible for finding hotspots (ex : whether method is called multiple times or not).
* **Garbage collector** collects and removes unreferenced objects from Heap area. It can be triggered by calling *System.gc()* but the execution is not guaranteed.
* ***Java Native Interface (JNI) :*** JNI will be interacting with the native method libraries and provides the native libraries required for the execution engine.
* ***Native method libraries :*** this is a collection of the native libraries which is required for execution engine.

### Explain how java class loading works?

|  |  |
| --- | --- |
| Class loaders are responsible for loading java classes during **runtime** **dynamically** to the JVM memory. Java classes aren’t loaded into memory all at once, but when required by an application. This is where class loaders come into picture.  They are part of JRE, hence JVM doesn’t need to know about underlying files or file systems inorder to run java programs. We can also create custom class loaders. |  |

* ***Built-in ClassLoaders***
* ***Bootstrap or Primodial***

java classes are loaded by an instance of java.lang.**ClassLoader**. But the question is who loads java.lang.ClassLoader itself ? this is where bootstrap class loader comes into picture. Bootstrap or Primodialclass loader is the **parent of all classloaders** and is written in native code (and different platforms might have different implementations) hence doesn’t showup as java class (displayed as null above). Its main job is to load **JDK internal** classes, typically rt.jar and other core libraries located under $JAVA\_HOME/jre/lib directory.

* **Extenstion**  class loader is a child of the bootstrap clas loader and loads classes that are an extension of standard core java classes. It loads classes from JDK extensions directory, usually $JAVA\_HOME/lib/ext directory or any other directory mentioned in **java.ext.dirs** system property.
* **Application or System** class loader is child of Extensions class loader and it takes care of loading all application level classes into the JVM. It loads files found in the classpath environment variable, –classpath or –cp command line option.
* ***How do class loaders work***

Class loaders are part of java runtime environment. When the JVM requests a class, the class loader tries to locate the class and load the class definition into the runtime using fully classfied class name. class loader in java works on 3 principles : **delegation**, **visibility** and **uniqueness**.

* *Delegation* principle forward request of class loading to parent class loader and only loads the class, if parent is not able to find or load class.
* *Visibility* principle allows child class loader to see all the classes loaded by parent ClassLoader, but parent class loader can not see classes loaded by child.
* *Uniqueness* principle allows to load a class exactly once, which is basically achieved by delegation and ensures that child ClassLoader doesn't reload the class already loaded by parent.

The *java.lang.ClassLoader.loadClass()* method is responsible for loading class definition at runtime. It tries to load class based on a fully qualified name, and if class isn’t already loaded, it **delegates** the request to the parent class loader, and this process happens **recursively**.

|  |  |
| --- | --- |
| Class Loader in Java BootStrap Extension and Application | https://1.bp.blogspot.com/-0gOWex7Pb2E/USTOh2K7zpI/AAAAAAAAAjc/_viQADzxrsk/s1600/Java+classloader+working.PNG |

### How to load class explicitly in java ?

|  |  |
| --- | --- |
| Java provides API to explicitly load class.  -> Class.forName(classname) and  -> Class.forName(classname, initialized, classloader).  For eg. In the simple JDBC code we use to load JDBC drives to Connect MySql database. As shown you can pass name of ClassLoader which should be used to load that particular class along with binary name of class. |  |

### Explain Java memory model and how garbage collection works ?

|  |  |
| --- | --- |
| The various pieces of java program (methods, variables and objects) live in one of two places in memory, the **STACK** or the **HEAP**.   * Instace variables and objects live on the heap. * Local variables live on the stack. * Method calls live on the stack.   The heap is where Java objects live, and it's the one and only part of memory that is in any way involved in the garbage collection process.  ***Garbage collection*** | Java Memory Model, JVM Memory Model, Memory Management in Java, Java Memory Management |

*the words "construct", "create", and "instantiate" are used interchangeably. They all mean, “An object is built on the heap.” This also implies that the object’s constructor runs, as a result of the construct/create/instantiate code. You can say with certainty, for example, that any code that uses the keyword* **new***, will (if it runs successfully) cause the class constructor and all superclass constructors to run.*

Java Garbage Collection is the process to **identify and remove** the unused objects from the memory and free space to be allocated to objects created in the future processing. One of the best feature of java programming language is the **automatic** garbage collection, unlike other programming languages such as C where memory allocation and deallocation is a manual process. **Garbage Collector** is the program running in the background that looks into all the objects in the memory and find out objects that are not referenced by any part of the program. All these unreferenced objects are deleted and space is reclaimed for allocation to other objects.

One of the basic way of garbage collection involves three steps:

***Marking***: This is the first step where garbage collector identifies which objects are in use and which ones are not in use.

***Normal Deletion***: Garbage Collector removes the unused objects and reclaim the free space to be allocated to other objects.

***Deletion with Compacting***: For better performance, after deleting unused objects, all the survived objects can be moved to be together. This will increase the performance of allocation of memory to newer objects.

There are two problems with simple mark and delete approach.First one is that it’s not efficient because most of the newly created objects will become unused, Secondly objects that are in-use for multiple garbage collection cycle are most likely to be in-use for future cycles too. The above shortcomings with the simple approach is the reason that Java Garbage Collection is **Generational** and we have Young Generation and Old Generation spaces in the heap memory. At broad level JVM Heap memory is physically divided into two parts – **Young** Generation and **Old** Generation.

***Young Generation***

* This is the place where all the new objects are created. It is divided into 3 parts, **Eden** memory and **2** **Survivor** memory.
* Most of newly created objects are located in Eden memory space. When Eden space is filled **Minor GC** is performed, and all survivor objects are moved to one of the survivor space. Minor GC also check for older survivor space members and move them aswell. So at a given time one of the survivor space is **always empty**.
* Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it is done by setting a threshold for the age of young generation objects before they become eligible to promote to old generation.

***Old Generation***

* Old generation memory contains the objects that are long lived and survived many rounds of minor GC. Usually garbage collection is performed in old generation when its memory is full. This is called **Major GC** and usually takes longer time.
* All garbage collection events are ***stop the world*** events, as all application threads are stopped until operation completes.
* Since young generation keep short lived objects, Minor GC is fast and application doesn’t get affected much. But Major GC takes longer time as it checks all the live objects and application becomes unresponsive for the duration, and might notice timeout errors. The duration taken by GC depends on the strategy used, hence it is necessary to monitor and tune the garbage collector to avoid timeout errors in highly responsive applications.

***Perm Generation (prior to Java8)***

* Permanent Generation or “Perm Gen” contains the application metadata required by the JVM to describe the classes and methods used in the application. Note that Perm Gen **is** **not part of** Java Heap memory.
* Perm Gen is populated by JVM at runtime based on the classes used by the application. It also contains Java SE library classes and methods.
* Perm Gen objects are garbage collected in a full garbage collection.

***MetaSpace***

* In Java8 permgen is replace with Metaspace. The JDK 8 HotSpot JVM is now using native memory for the representation of class metadata and is called **Metaspace**.

### What is garbage collection ? when does it run ?

Memory management is a crucial element in many types of applications. Consider a program that reads in large amounts of data, say from somewhere else on a network, and then writes that data into a database on a hard drive. A typical design would be to read the data into some sort of collection in memory, perform some operations on the data, and then write the data into the database. After the data is written into the database, the collection that stored the data temporarily must be emptied of old data or deleted and recreated before processing the next batch. This operation might be performed thousands of times, and in languages like C or C++ that do not offer automatic garbage collection, a small flaw in the logic that manually empties or deletes the collection data structures can allow small amounts of memory to be improperly reclaimed or lost. Forever. These small losses are called **memory** **leaks**, and over many thousands of iterations they can make enough memory inaccessible that programs will eventually crash. Creating code that performs manual memory management cleanly and thoroughly is a nontrivial and complex task, and while estimates vary, it is arguable that manual memory management can double the development effort for a complex program.

Java's garbage collector provides an **automatic** solution to memory management. In most cases it frees you from having to add any memory management logic to your application. The downside to automatic garbage collection is that you can't completely control when it runs and when it doesn't.

Whenever a software program executes(any program language), it uses memory in several different ways. Memory will be used to create a stack, a heap, constant pools, and method areas. The **heap** is the part of memory where java objects live, and it is the **one and only part** of memory that is in any way involved in the garbage collection process. *There is one and only one heap*. So garbage collection revolves around making sure the heap has a much free space as possible.

When the garbage collector runs **its purpose** is to find and delete objects that cannot be reached by the running program. The garbage collector is under the **control of the JVM**. The JVM decides when to run the garbage collector. From within your Java program you can ask the JVM to run the garbage collector, but there are ***no*** guarantees, under any circumstances, that the JVM will comply. Left to its own devices, the JVM will typically run the garbage collector ***when it senses that memory is running low***.

Experience indicates that when your Java program makes a request for garbage collection, the JVM will usually grant your request in short order, but there are no guarantees. Just when you think you can count on it, the JVM will decide to ignore your request. Garbage collection implementation is JVM Specific. It can be implemented in many ways, ex : mark and sweep algoritm, refence counting, etc. **Important** thing to know is when does Object become eligible for garbage collection ? Every java program has from one to many threads. Each thread has its own execution stack. When a program runs **atleast one** thread with the main() method at bottom of stack runs. The initial thread can further launch further additional threads. These threads will have their own little execution stack, and its own lifecycle. A given thread can be alive or dead. ***So, an Object is eligible for garbage collection when no live thread can access it or reach it.*** When we talk about reaching an object, it means a reachable reference variable that refers to the object in question. If our Java program has a reference variable that refers to an object, and that reference variable is available to a live thread, then that object is considered reachable.

Garbage collection cannot ensure that there is enough memory, only that the memory that is available will be manage efficiently. If the program consists of too many live objects(objects referenced from other live objects) the system can run out of memory.

### How to make objects eligible for GC ? Can you force GC ?

* **Nulling** a refence variable. Set the reference variable that refers to the object to **null**.
* **Re-assigning** reference variable. Decouple a reference variable from an object by setting the reference variable to refer to another object.
* Objects that are created in a method will be eligible for GC once the **method returns** and there are no references existing for the object outside the method.
* **Isolating a reference**. There can be cases where multiple objects refer to each other but there is no way for any of the active thread to reach to any of these objects. In such case objects become eligible for GC. This scenario is called as “islands of isolation”.
* Garbage collection can never be forced. However there are certain methods that allow you to request the JVM to perform garbage collection. The GC routines that java provides are members of the Runtime class. The Runtime class is a special class that has a single object for each main program. The Runtime object provides a mechanism for communicating directly with the virtual machine. To get the Runtime instance, you can use the method **Runtime.getRuntime(),** which returns the Singleton. Once you have the Singleton you can invoke the garbage collector using the **gc()** method. Alternatively, you can call the same method on the System class, which has static methods that can do the work of obtaining the Singleton for you. The simplest way to ask for garbage collection (remember-just a request) is **System.gc();**
* **finalize()** method.

Java provides a mechanism to run some code just before the object is deleted by the garbage collector. This can be achieved by overriding the method finalize() that all classes inherit from class Object. But the problem is the timing of the GC deleting the object being not guaranteed, which means finalize() method may not run, putting important code withing finalize() method is not recommended.

* For any given object, finalize() will be called only once (at most) by the garbage collector.
* Calling finalize() can actually result in saving an object from deletion.

First of all, remember that any code that you can put into a normal method you can put into finalize(). For example, in the finalize() method you could write code that passes a reference to the object in question back to another object, effectively *uneligiblizing* the object for garbage collection. If at some point later on this same object becomes eligible for garbage collection again, the garbage collector can still process this object and delete it. The garbage collector, however, will remember that, for this object, finalize() already ran, and it will not run finalize() again.

# Literals, Assignments and Variables

### What are the default values and ranges of primitive variables in java ?

|  |  |  |  |
| --- | --- | --- | --- |
| **Data type** | **Size** | **Default value** | **Range** |
| byte | 8 bit | 0 | min [-128] to max [127] (inclusive) |
| short | 16 bit | 0 | min [-32,768] to max [32,767] (inclusive) |
| int | 32 bit | 0 | min [-2,147,483,648] to max [2,147,483,647] (inclusive) |
| long | 64 bit | 0L | min [-9,223,372,036,854,775,808] to  max [9,223,372,036,854,775,807] (inclusive) |
| float | 32 bit | 0.0f | n/a |
| double | 64 bit | 0.0d | n/a |
| char | 16 bit | ‘\u0000’ | n/a |
| String or any Object | n/a | null | n/a |
| boolean | 1 bit | false | n/a |

|  |  |
| --- | --- |
| All six number types in Java are made up of a certain number of 8-bit bytes, and are **signed**, meaning they can be negative or positive. The **leftmost bit** (the most significant digit) is used to represent the sign, where a **1 means negative and 0 means positive**, and the **rest of the bits** represent the value using **two’s complement** notation.  with a byte, for example, there are 256 possible numbers (or 28). Half of these are negative, and half-1 are positive. The positive range is one less than the negative range because the number zero is stored as a positive binary number. We use the formula **-2(bits-1)** to calculate the negative range,  and we use **2(bits-1)-1** for the positive range. |  |

### what is a primitive literal ?

A primitive literal is merely a source code representation of the primitive data types - in other words, an integer, floating-point number, boolean, or character that you type in while writing code.

*'b' // char literal*

*42 // int literal*

*false // boolean literal*

*2546789.343 // double literal*

* ***Integer literals***

Numeric literals can be declared using underscore characters (**\_**), ostensibly to improve readability, but it cannot be used at the beginning or end of the literal.

int pre7 = 1000000; // pre Java 7 – we hope it's a million

int with7 = 1\_000\_000; // much clearer!

There are four ways to represent integer numbers in the Java language,

* + **Decimal** literal (base 10)

This is the default representation when used without any prefix or suffix

*int length = 343;*

* + **Binary** literal (base 2)

Binary literal can only contain only the digits 0 or 1, and must start with either 0b or 0B

*int b1 = 0B101010; // set b1 to binary 101010 (decimal 42)*

*int b2 = 0b00011; // set b2 to binary 11 (decimal 3)*

* + **Octal** literal (base 8)

Octal integers use only the digits 0 to 7, and an integer is represented in octal form by **prefixing 0(zero)** in front of the number.

*int six = 06; // Equal to decimal 6*

*int seven = 07; // Equal to decimal 7*

*int eight = 010; // Equal to decimal 8*

*int nine = 011; // Equal to decimal 9*

You can have up to 21 digits in an octal number, not including the leading zero.

* + **Hexadecimal** literal (base 16)

Hexadecimal (hex for short) numbers are constructed using 16 distinct symbols. Because we never invented single-digit symbols for the numbers 10 through 15, we use alphabetic characters to represent these digits. Counting from 0 through 15 in hex looks like this: 0 1 2 3 4 5 6 7 8 9 a b c d e f. You are allowed up to 16 digits in a hexadecimal number, not including the **prefix 0x (or 0X)** or the optional suffix extension L.

*int x = 0X0001;*

*int y = 0x7fffffff;*

*int z = 0xDeadCafe;*

All four integer literals (binary, octal, decimal, and hexadecimal) are defined as int by default, but they may also be specified as **long by placing a suffix of L or l** after the number:

*long jo = 110599L;*

*long so = 0xFFFFl; // Note the lowercase 'l'*

* ***Floating point literals***

Floating-point numbers are defined as a number, a decimal symbol, and more numbers representing the fraction. Floating-point literals are defined **as double (64 bits) by default**, so if you want to assign a floating-point literal to a variable of type float (32 bits), you must attach the **suffix F or f** to the number. If you don't do this, the compiler will complain about a possible loss of precision.

*Double d = 11301874.9881024;*

*float f = 23.467890; // Compiler error, possible loss of precision*

*float g = 49837849.029847F; // OK; has the suffix "F"*

* ***Boolean literals***

Boolean literals are the source code representation for boolean values. A boolean value can be defined only as true or false.

*boolean t = true; // Legal*

*boolean f = 0; // Compiler error!*

* ***Character literals***

A char literal is represented by a single character in single quotes.You can also type in the Unicode value of the character, using the Unicode notation of prefixing the value with \u.

*char a = 'a';*

*char b = '@';*

*char letterN = '\u004E'; // The letter 'N'*

Characters are just 16-bit unsigned integers under the hood. That means you can assign a number literal, assuming it will fit into the unsigned 16-bit range (0 to 65535).

*char a = 0x892; // hexadecimal literal*

*char b = 982; // int literal*

*char c = (char)70000; // The cast is required; 70000 is out of char range*

*char d = (char) -98; // Ridiculous, but legal*

*char e = -29; // Possible loss of precision; needs a cast*

*char f = 70000; // Possible loss of precision; needs a cast*

*char c = '\"'; // A double quote*

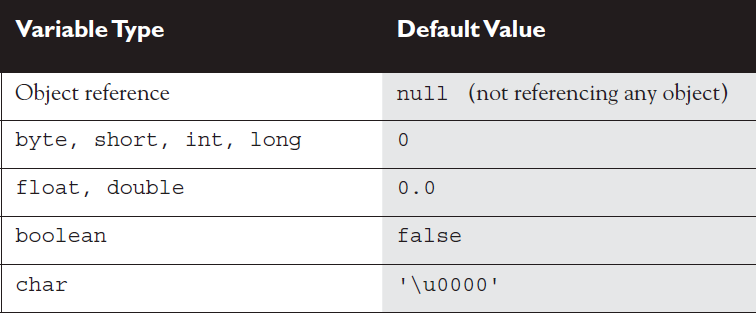
*char d = '\n'; // A newline*

*char tab = '\t'; // A tab*

### What are the default values of variables in java ?

Variables are just **bit holders**, with a designated type. You can have an int holder, a double holder, a Button holder, and even a String[] holder. Within that holder is a bunch of bits representing the value. For primitives, the bits represent **a numeric value**. A byte with a value of 6, for example, means that the bit pattern in the variable (the byte holder) is 00000110, representing the 8 bits.

* Instance variables are initialized to a default value each time a new instance is created, although they may be given explicit values after objects super constructors complete.



* An array is an object; thus, an array instance variable that's declared but not explicitly initialized will have a value of **null, but** if initialized all array elements are given their default values, the same default values that elements of that type get when they're instance variables. The bottom line: ***Array elements are always, always, always given default values, regardless of where the array itself is declared or instantiated.***
* One exception to the way object references are assigned is **String**. In Java, String objects are given special treatment. For one thing, String objects are **immutable**; you can't change the value of a String object. *any time we make any changes at all to a String, the VM will update the reference variable to refer to a different object.*

### What are the various scopes in which variables can be and how long they last?

There are 4 basic scopes

* **Static** variables have the **longest scopes**. They are created when the class gets loaded in JVM and survive as long as the class stays loaded in the JVM.
* **Instance** variables are the next most long lived. They are created when a **new** instance is created and they live until the instace is **removed**.
* Local variables are next, they live as long as their methods remains **on the stack**.
* Block variables live only as long as the **code block** is executing.

### What is the role of casting during variable assignments ?

* A variable referring to an object is just a reference variable. A reference variable bit holder contains bits *representing* ***a way to get to the object***.
* Casting lets you convert primitive values from one type to another. Casts can be ***implicit*** or ***explicit***.
* An implicit cast means you don't have to write code for the cast; the conversion happens automatically. Typically, an implicit cast happens when you're doing a ***widening*** conversion. In other words, putting a smaller thing (say, a byte) into a bigger container (like an int).
* The large-value-into-small-container conversion is referred to as narrowing and requires an explicit cast, where you tell the compiler that you're aware of the danger and accept full responsibility.
* every floating point literal is implicitly **double**.
* When you assign one primitive/reference variable to other, the content of the right hand side is copied to left hand side. Means the bit pattern **get copied**.

### What is an Array ? How it works ?

* Arrays can hold primitives or objects, but the array itself is always an **object**.
* When you declare an array, the brackets can be to the left or right of the name.
* It is never legal to include the **size** of an array in the **declaration**. You must include the size of an array when you construct it (using new) unless you are creating an anonymous array.
* Elements in an array of objects are not automatically created, although primitive array elements are given default values.
* You'll get a NullPointerException if you try to use an array element in an object array, if that element does not refer to a real object.
* Arrays are indexed beginning with **zero**. An ArrayIndexOutOfBoundsException occurs if you use a bad index value. Arrays have a **length** attribute whose value is the number of array elements. The last index you can access is always one less than the length of the array.
* Multidimensional arrays are just arrays of arrays. The dimensions in a multidimensional array can have different lengths.
* An array of primitives can accept any value that can be promoted implicitly to the array's declared type—for example, a byte variable can go in an int array.
* An array of objects can hold any object that passes the IS-A (or instanceof) test for the declared type of the array. For example, if Horse extends Animal, then a Horse object can go into an Animal array.
* If you assign an array to a previously declared array reference, the array you're assigning must be the **same** dimension as the reference you're assigning it to.
* You can assign an array of one type to a previously declared array reference of one of its supertypes. For example, a Honda array can be assigned to an array declared as type Car (assuming Honda extends Car).

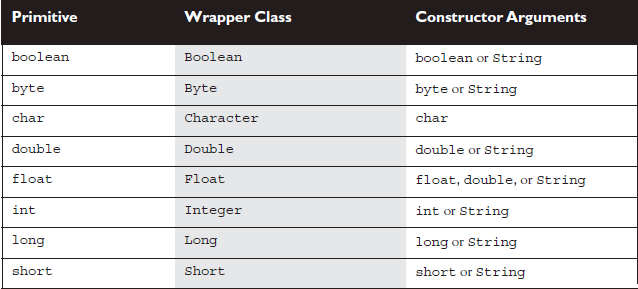
### Explain how ArrayList works ?

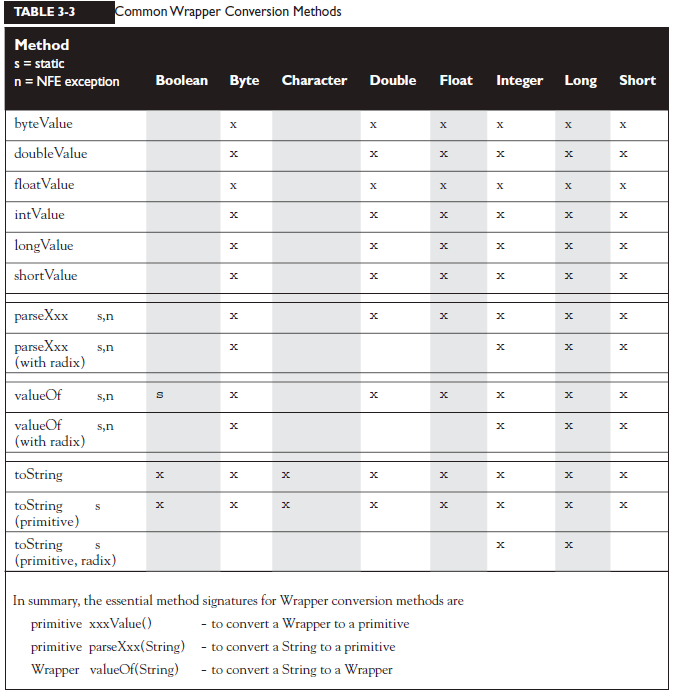
* ArrayList class in java.util package, and it implements **List** interface. Similar to Array, whey you build an ArrayList you have to declare what kind of object it can contain.
* Like Arrays indexes of ArrayList are zero based. Benefit of ArrayList comparitively is that we can add/insert Objects to the collection on the fly.

### What are Wrapper Classes and how does Boxing work here ?

The wrapper classes in the java API serve two primary purposes:

* To provide a mechanism to "wrap" primitive values in an object so that the primitives can be ***included in activities reserved for objects***, like being added to Collections, or returned from a method with an object return value.
* To provide an assortment of utility functions for primitives. Most of these functions are related to various conversions: converting primitives to and from String objects, and converting primitives and String objects to and from different bases (or radix), such as binary, octal, and hexadecimal.
* There is a wrapper class for each primitive in java. Table below lists them.



* Wrapper objects are **immutable**. Once they have been given a value, that value cannot be changed.

### What is AutoBoxing and Unboxing ?

* Feature added in Java 5, Boxing and unboxing make using wrapper classes more convenient.
* (==) operator when applied to primitive or reference variables looks for ***the equality of the bit patterns*** of the variables. so from object reference point of you (==) will return true only if both reference variables are pointing to same object. In order to check whether the object’s are meaningfully equal, use **equals()** method.
* There is an exception to above rule..

In order to save memory, two instances of the following wrapper objects (created through boxing), will always be == when their primitive values are the same:

■ Boolean

■ Byte

■ Character from \u0000 to \u007f (7f is 127 in decimal)

■ Short and Integer from -128 to 127 (in the bytes range)

so because of above, see below example

Integer num1 = 1000;

Integer num2 = 1000;

(num1==num2) will return false

But..

Integer num1 = 10;

Integer num2 = 10;

(num1==num2) will return true

### Does Java Use Pass-By-Value or pass-by-reference Semantics?

when Java passes objects by passing the reference variable instead, does that mean Java uses pass-by-reference for objects? **Not exactly**, although you'll often hear and read that it does. Java is ***actually pass-by-value*** for all variables running within a single VM. Pass-by-value means pass-by-variable-value. And that means, ***pass-by-copy-ofthe-variable*** (There's that word copy again!)

It makes no difference if you're passing primitive or reference variables, you are **always passing a copy of the bits** in the variable.

### Explain how Overloading work with Boxing and var-args ?

three factors that can make overloading a little tricky:

* Widening
* Autoboxing
* Var-args

When a class has overloaded methods, one of the compiler's jobs is to determine which method to use whenever it finds an invocation for the overloaded method.

* *when an exact parameter match isn't found, the JVM uses the method with the smallest argument* ***that is wider*** *than the parameter.*
* *On top of above below rules apply from Java 5 onwards.*
  + Widening beats boxing
  + Widening beats var-args
  + Boxing beating var-args

Reference parameter widening happens if the object passes **IS-A** test for reference variable. For ex. You can pass a horse object to method that is expecting animal reference.

Wideing will not happen in case of wrapper classes, because the wrapper classes are peers to one another.

To summarize the rules

* Primitive widening uses the "smallest" method argument possible.
* Used individually, boxing and var-args are compatible with overloading.
* You CANNOT widen from one wrapper type to another. (IS-A fails.)
* You CANNOT widen and then box. (An int can't become a Long.)
* You can box and then widen. (An int can become an Object, via Integer.)
* You can combine var-args with either widening or boxing.

### Explain various Java Operators ?

#### Relational Operators

* Relational operators always result in a boolean value (true or false).
* There are six relational operators: >, >=, <, <=, ==, and !=. The last two (== and !=) are sometimes referred to as equality operators.
* When comparing characters, Java uses the Unicode value of the character as the numerical value.
* Equality operators

❑ There are two equality operators: == and !=.

❑ Four types of things can be tested: numbers, characters, booleans, and reference variables.

❑ When comparing reference variables, == returns true only if both references refer to the same object.

#### instanctof operator

* **instanceof** is for reference variables only, and checks for whether the object is of a particular type.
* The instanceof operator can be used only to test objects (or null) against class types that are in the same class hierarchy.
* For interfaces, an object passes the instanceof test if any of its superclasses implement the interface on the right side of the instanceof operator.

#### Arithmetic Operators

* There are four primary math operators: add, subtract, multiply, and divide.
* The remainder operator (%), returns the remainder of a division.
* Expressions are evaluated from left to right, unless you add parentheses, or unless some operators in the expression have higher precedence than others.
* The \*, /, and % operators have higher precedence than + and -.

#### String Concatenation

* If either operand is a String, the + operator concatenates the operands.
* If both operands are numeric, the + operator adds the operands.

#### Increment/Decrement and ternary operator

* Prefix operators (++ and --) run before the value is used in the expression.
* Postfix operators (++ and --) run after the value is used in the expression.
* In any expression, both operands are fully evaluated before the operator is applied.
* Variables marked final cannot be incremented or decremented.
* Returns one of two values based on whether a boolean expression is true or false.

❑ Returns the value after the ? if the expression is true.

❑ Returns the value after the : if the expression is false.

#### Logical Operators

* six main "logical" operators: &, |, ^, !, &&, and ||.
* Logical operators work with two expressions (except for !) that must resolve to boolean values.
* The && and & operators return true only if both operands are true.
* The || and | operators return true if either or both operands are true.
* The && and || operators are known as short-circuit operators.
* The && operator does not evaluate the right operand if the left operand is false.
* The || does not evaluate the right operand if the left operand is true.
* The & and | operators always evaluate both operands.
* The ^ operator (called the "logical XOR"), returns true if exactly one operand is true.
* The ! operator (called the "inversion" operator), returns the opposite value of the boolean operand it precedes.

# Flow Control, Exceptions and Assertions

### Explain how If and switch statements are used for flow control ?

* The only legal expression in an if statement is a boolean expression, in other words an expression that resolves to a boolean or a Boolean variable.
* Watch out for boolean assignments (=) that can be mistaken for boolean equality (==) tests:

boolean x = false;

if (x = true) { } // an assignment, so x will always be true!

* Curly braces are optional for if blocks that have only one conditional statement. But watch out for misleading indentations.
* switch statements can **evaluate** only to enums or the byte, short, int, char, and string data types. You can't say,

double s = 30.23;

switch(s) { }

* The case constant must be a literal or final variable, or a constant expression, including an enum. You cannot have a case that includes a nonfinal variable, or a range of values.
* If the condition in a switch statement matches a case constant, execution will run through all code in the switch following the matching case statement **until a break** statement or the end of the switch statement is encountered. In other words, the matching case is just the entry point into the case block, but unless there's a break statement, the matching case is not the only case code that runs.
* The **default** keyword should be used in a switch statement if you want to run some code when none of the case values match the conditional value.
* The default block can be located anywhere in the switch block, so if no case matches, the default block will be entered, and if the default does not contain a break, then code will continue to execute (fall-through) to the end of the switch or until the break statement is encountered.

### Explain various Loop construts in java ?

* The **while loop** will execute the body as long as test condition is met.
* The **do loop** will enter the body of the loop at least once, even if the test condition is not met.
* A basic **for statement** has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.
* If a variable is incremented or evaluated within a basic for loop, it must be declared before the loop, or within the for loop declaration.
* A variable declared (not just initialized) within the basic for loop declaration cannot be accessed outside the for loop (in other words, code below the for loop won't be able to use the variable).
* You can initialize more than one variable of the same type in the first part of the basic for loop declaration; each initialization must be separated by a comma.
* An **enhanced for** statement (new as of Java 6), has two parts, the declaration and the expression. It is used only to loop through arrays or collections.
* With an enhanced for, the expression is the array or collection through which you want to loop.
* With an enhanced for, the declaration is the block variable, whose type is compatible with the elements of the array or collection, and that variable contains the value of the element for the given iteration.
* An **unlabeled break statement** will cause the current iteration of the innermost looping construct to stop and the line of code following the loop to run.
* An **unlabeled continue statement** will cause: the current iteration of the innermost loop to stop, the condition of that loop to be checked, and if the condition is met, the loop to run again.
* If the break statement or the continue statement is labeled, it will cause similar action to occur on the labeled loop, not the innermost loop.

### Explain java Exception handling mechanism ?

The term "exception" means "exceptional condition" and is an occurrence that alters the normal program flow. A bunch of things can lead to exceptions, including hardware failures, resource exhaustion, and good old bugs.

When an exceptional event occurs in Java, an exception is said to be "thrown." The code that's responsible for doing something about the exception is called an "exception handler," and it "catches" the thrown exception.

Exception handling works by transferring the execution of a program to an appropriate exception handler when an exception occurs. For example, if you call a method that opens a file but the file cannot be opened, execution of that method will stop, and code that you wrote to deal with this situation will be run. Therefore, we need a way to tell the JVM what code to execute when a certain exception happens. To do this, we use the **try and catch** keywords. The try is used to define a block of code in which exceptions may occur. This block of code is called a guarded region (which really means "risky code goes here"). One or more catch clauses match a specific exception (or group of exceptions) to a block of code that handles it.

#### finally

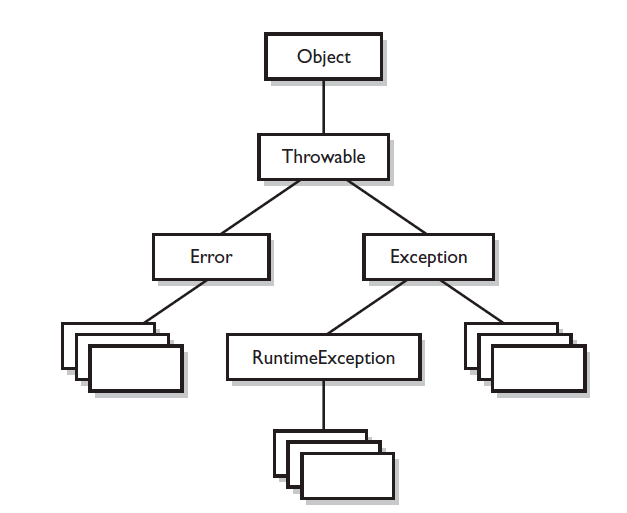
A **finally** block encloses code that is **always executed** at some point after the try block, whether an exception was thrown or not. **Even if there is a return** statement in the try block, the finally block executes right after the return statement is encountered, and before the return executes.

This is the right place to close your files, release your network sockets, and perform any other cleanup your code requires. If the try block executes with no exceptions, the finally block is executed immediately after the try block completes. If there was an exception thrown, the finally block executes immediately after the proper catch block completes. finally always runs.

#### Exception class hierarchy

All exception classes are **subtypes** of class Exception. This class derives from the class **Throwable** (which derives from the class Object). there are two subclasses that derive from Throwable: **Exception** and **Error**.

Classes that derive from Error represent unusual situations that are not caused by program errors, and indicate things that would not normally happen during program execution, like JVM running out of memory. Generally, your application won't be able to recover from an Error, so you're not required to handle them. If your code does not handle them (and it usually won't), it will still compile with no trouble. Although often thought of as exceptional conditions, Errors are technically not exceptions because they do not derive from class Exception.



The **order** in which the catch clauses are placed matters if the exception‘s handled fall in same **inheritance** hierarchy. The handlers for the most specific exceptions must always be placed above those for more general exceptions. The order does not matter if exceptions are siblings, i.e one Exception class is not a subtype or supertype of the other. Just as a method must specify what type and how many arguments it accepts and what is returned, the exceptions that a method can throw must be declared (unless the exceptions are subclasses of RuntimeException). The list of thrown exceptions is part of a method's public interface. The throws keyword is used as follows to list the exceptions that a method can throw:

*void myFunction() throws MyException1, MyException2*

*{*

*// code for the method here*

*}*

Any method that might throw an exception (**unless** it's a subclass of RuntimeException) must **declare** the exception. That includes methods that aren't actually throwing it directly, but are "ducking" and letting the exception pass down to the next method in the stack. If you "duck" an exception, it is just as if you were the one actually throwing the exception. RuntimeException subclasses are exempt, so the compiler won't check to see if you've declared them.

But all non-Runtime Exceptions are considered "**checked**" exceptions, because the compiler checks to be certain you've acknowledged that "bad things could happen here." Remember this:

*Each method must either handle all checked exceptions by supplying a catch clause or list each unhandled checked exception as a thrown exception*. This rule is referred to as Java's "***handle or declare***" requirement. (Sometimes called "*catch or declare*.")

#### Common Exception and Errors

There are 2 broad categories of exceptions and errors.

1. **JVM Exceptions** : those exceptions or errors that are either exclusively or most logically thrown by the JVM. Eg.

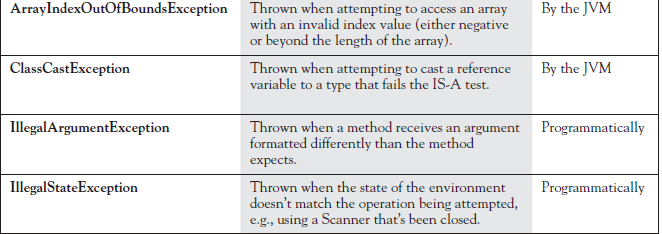
*NullPointerException* -> this exception is thrown when you attempt to access an object using a reference variable with a current value of null. There is no way a compiler can hope to find these problems before runtime.

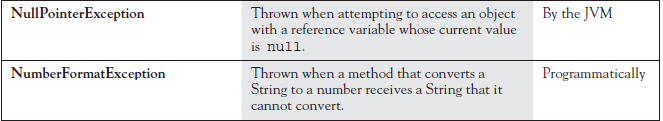
*StackOverFlowError* -> the call stack is the chain of methods that your program executes to get to the current method. main() would be at the bottom of the call stack, and that as main() invokes another method, and that method invokes another, and so on, the stack grows upward. the stack resides in memory, and even if OS gives a gigabyte of RAM for program, it's still a finite amount. It's possible to grow the stack so large that the OS runs out of space to store the call stack. When this happens you get StackOverflowError. Only JVM knows when this moment occurs and the JVM will be the source of this error.

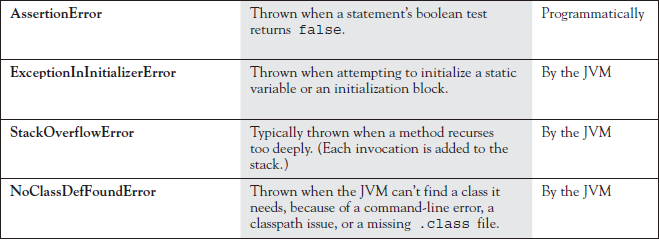
1. **Programmatic Exceptions**: Those exceptions or errors that are thrown explicity by application and /or API Programmers.

*NumberFormatException* -> many classes in the Java API have methods that take String arguments, and convert these Strings into numeric primitives. A good example of these classes are the "wrapper classes", that have parseInt() , valueOf() methods. If a string is passed to these methods which cannot be converted to a number, a NumberFormatException() is thrown.

*IllegalArgumentException***,** *AssertionError*are some other examples.





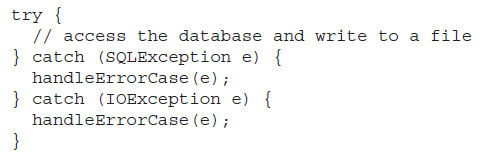


#### Exception Summary

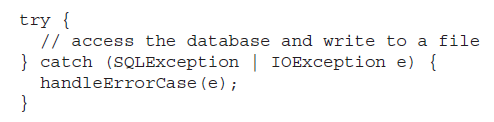
* Exceptions come in two flavors: **checked** and **unchecked**.
* Checked exceptions include all subtypes of Exception, excluding classes that extend RuntimeException.
* Checked exceptions are subject to the **handle or declare** rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using throws, or handle the exception with an appropriate try/catch.
* Subtypes of **Error** or **RuntimeException** are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them, or to declare them, but the compiler doesn't care one way or the other.
* If you use an optional finally block, it will always be invoked, regardless of whether an exception in the corresponding try is thrown or not, and regardless of whether a thrown exception is caught or not.
* The only exception to the finally-will-always-be-called rule is that a finally will not be invoked if the JVM **shuts** down. That could happen if code from the try or catch blocks calls System.exit().
* Just because finally is invoked does not mean it will complete. Code in the finally block could itself raise an exception or issue a System.exit().
* Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to main(), and main() is "ducking" the exception by declaring it).
* You can create your own exceptions, normally by extending Exception or one of its subtypes. Your exception will then be considered a checked exception, and the compiler will enforce the handle or declare rule for that exception.
* All catch blocks must be ordered from most specific to most general. If you have a catch clause for both IOException and Exception, you must put the catch for IOException first in your code. Otherwise, the IOException would be caught by catch(Exception e), because a catch argument can catch the specified exception or any of its subtypes! The compiler will stop you from defining catch clauses that can never be reached.
* Some exceptions are created by programmers, some by the JVM.

### Explain multi-catch exception handling feature of java7 ?

Sometimes we want to handle different types of exceptions the same way i.e reuse the exception handler. Prior to java7 best way of doing this was to have multiple catch blocks as below,



Since Java-7 this becomes easier with the multi-catch feature..



But there are certain rules to keep in mind..

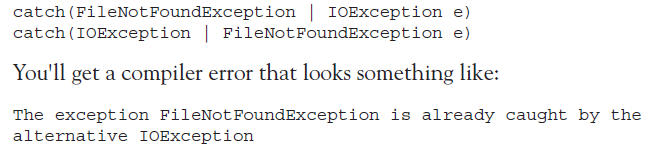
* You can’t use variable name multiple times in a multi-catch, below will not compile,



* With multi-catch order of Exception’s handled doesn’t matter



* But with multi-catch a given exception should match only one type. Multi-catch is only for exceptions in **different inheritance hierarchies**. In below case This is compilation error because FileNotFoundException is a subclass of IOException.



* Don’t assign new value to the catch parameter.

It is legal java code to assign a new value to the catch blocks parameter when there is only one type listed and it will compile, but this is not allowed in multi-catch as it uses multiple types.

|  |  |
| --- | --- |
| Legal but bad practice !! | this throws compilation error :  *“the parameter e of multi-catch block cannot be assigned”* |

* **Rethowing Exceptions**

Sometimes we want to do something with the Exception thrown (logging etc) and rethrow them (handle and declare - pattern). There are two ways to write this,

|  |  |
| --- | --- |
|  |  |

The second version is the simple version, it is not actually catching any Exception subclass, but *only those* covariant types of the **declared ones in throws** clause.

This was done to avoid a issue, when lets say, suppose the API developers of couldThrowAnException() decide the method will never throw a SQLException and remove it from signature of method, the code would stop compiling with error like..



It is reasonable for code to stop compiling if we add exceptions to a method. But we don't want our code to break if a method's implementation gets LESS brittle. And that's the advantage of using:



Java **infers what we mean** here and doesn't say a peep when the API we are calling **removes** an exception. Below is an example..

|  |  |
| --- | --- |
|  |  |

### Explain try-with-resource feature of Java7 ?

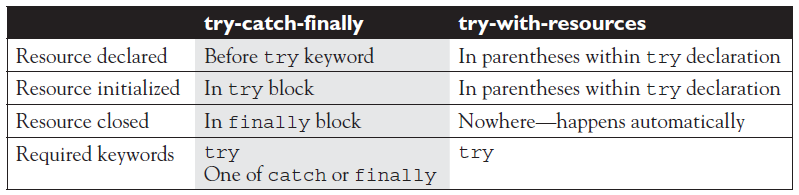
Finally block is a good place to close files/other resources used, but this clean up code can be verbose.

|  |  |
| --- | --- |
|  | In this example it is lot of code to just close one file. At line 10 we can get exception while we try to close the file and we then have to ignore the exception. |

Java7 onwards we have a new feature called Automatic Resource management using “try-with-resources”, that help remove the verbose code.

|  |  |
| --- | --- |
| The syntax is inspired from for loop, we can use semicolon to declare multiple resources. | In here we start by declaring reader inside try declaration. The reader is scoped to just the try block (not even the catch block).  Even though the finally block is not present, the try with resource statement is logically calling a finally block to close the reader. You can also add a finally block as well if needed. |

* the type used in a try-with-resource statement must implement **AutoCloseable**.
* try-with-resource automatically calls close() on any resource declared in try as *try(Resource r = new Foo())*
* A try must have atleast one catch or finally unless it is a try-with-resource, where it can have neither, one or both the keywords.



* try-with-resource are closed in **reverse order** of creation and **before going on to catch or finally**.
* If more than one exception is thrown in a try-with-resources block, the exception thrown in the try block gets priority and others gets added as a **suppressed** exception array.

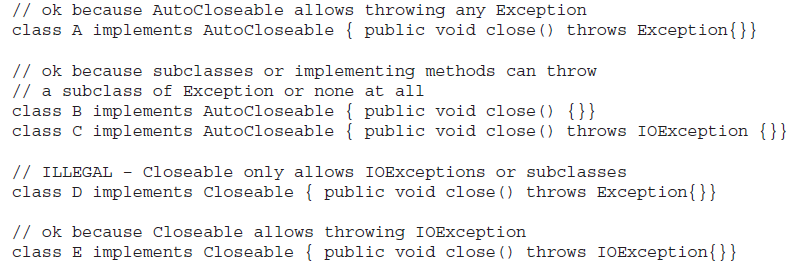
|  |  |
| --- | --- |
|  |  |

### What is difference between Closeable and Autocloseable interface ?

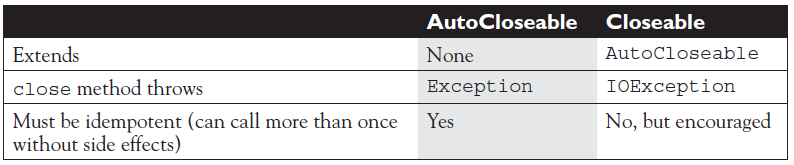
* Both Closeable and Autocloseable interfaces are used to close resources.
* Closeable was introduced with JDK5, while Autocloseable is super interface of Closeable and was introduced with JDK7.

|  |  |
| --- | --- |
|  |  |

Closeable has some limitation as it can only throw IOException, and it couldn’t be changed without breaking legacy code, hence Autocloseable was introduced as super interface that can throw Exception.



* AutoCloseable was specifically designed to work with try-with-resources statements. As Closeable extends AutoCloseable, you can use try-with-resources to close any resources that implement either Closeable or AutoCloseable.
* What happens if we call close() multiple times ? For classes that implement AutoCloseable, the implementation is required to be **idempotent**. Which means you can call close() all day and nothing will happen the second time and beyond. It will not attempt to close the resource again and it will not blow up. For classes that implement Closeable, there is no such guarantee.



### What is the use of Assertion in java ? Explain how to use it ?

* Assertions give you a way to **test** your **assumptions** during development and debugging.
* Assestions come in 2 flavors: really simple and simple

|  |  |
| --- | --- |
|  | The difference between the two is that the simple version adds a **second** expression, separated from the first (boolean expression) by a **colon**, this expression's string value is added to the stack trace. Both versions throw an immediate AssertionError, but the simple version gives you a little more debugging help while the really simple version simply tells you only that your assumption was false. |

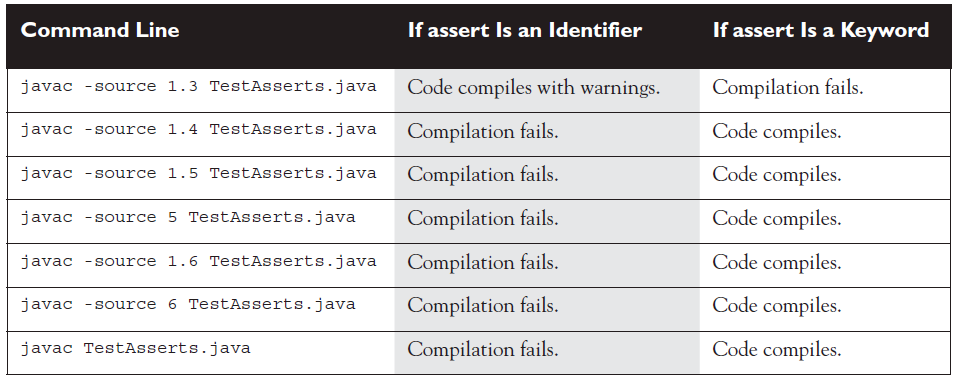
* Assertions are typically **enabled** during testing but **disabled** during deployment.
* You can use assert as an identifier (in older versions like 1.3) or as keyword (as of version 1.4), but not both together.

To compile older code that uses assert as an identifier,



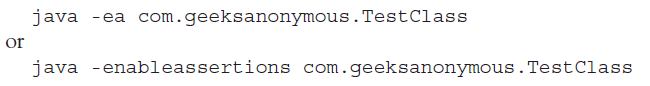
To compile code with version 1.4 or higher where assert is keyword,



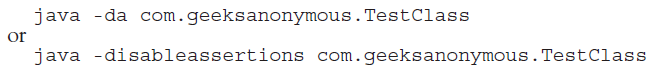


* Assertions are **disabled** at runtime by default.

To enable them, use a command-line flag **-ea** or **-enableassertions** flag.



Selectively disable assertions by using the **-da** or **-disableassertions** flag.



* There are various ways to use command line switches to selectively enable or disable assertions,
  + *With* ***no*** *arguments*

Enables or Disables assertions in all classes except for system classes, like example above.

* + *With a* ***package*** *name*

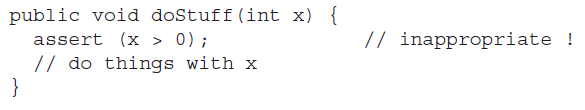
Enables or Disables assertions in the package specified and any packages below this package in the same directory hierarchy.

* + *With a* ***class*** *name*

Enables or Disables assertions in the class specified.

|  |  |
| --- | --- |
|  |  |

* Appropriate use of assertions : not all legal use of assertions are considered appropriate. Below are few of appropriate recommended usages.
  + **Do not** **use** assertions to validate arguments to public methods.

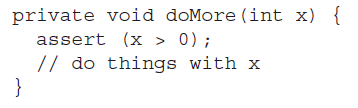


A public method might be called from code that you don’t control, and since assertions aren’t guaranteed to run, any argument validations/constraints/enforcements must be enforced by method itself, and you don’t want publicly accessible methods to work conditionally.

* + **Do not use** assertions to validate command line arguments.

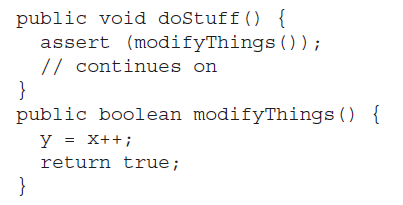
This is a special case of above rule, if your program requires command line arguments, you must use the exception mechanism to enforce them.

* + **Do use** assertions to validate arguments to private method.

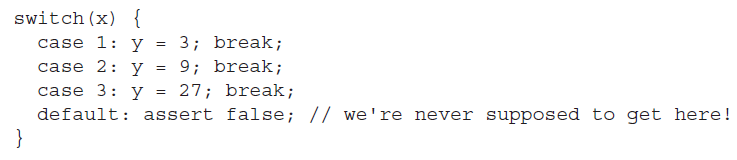


Since you have complete control of private method’s caller, and you assume the logic in code calling the private method correct, you can test that assumption with assertion.

* + **Do not** use assert expressions that cause side effects. Assertions aren't guaranteed to always run, and you don't want behavior that changes depending on whether assertions are enabled.



* + **Do use** assertions—even in public methods—to validate that a particular code block will never be reached. You can use assert false; for code that should never be reached, so that an assertion error is thrown immediately if the assert statement is executed.



# Strings, Data formatting, Resource bundles, DateTime, Locales

### How differently does java handle String object ?

* String objects are **immutable**, and String reference variables are not.
* Each character in a string is a **16 bit** unicode character.
* If you create a new String without assigning it, it will be lost to your program.
* If you redirect a String reference to a new String, the old String can be lost.
* String methods use **zero-based** indexes, except for the second argument of substring().
* The String class is **final**—its methods can't be overridden.
* When the JVM finds a String literal, it is added to the String **literal pool**. To make Java more memory efficient, the JVM sets aside a special area of memory called the **"String constant pool/literal 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.
* Strings have a method: **length();** arrays have an **attribute** named length.
* Remember that chained methods are evaluated from left to right.
* String methods to remember: charAt(), concat(), equalsIgnoreCase(), length(), replace(), substring(), toLowerCase(), toString(), toUpperCase(), and trim().
* StringBuffer methods to remember: append(), delete(), insert(),reverse(), and toString().

### What are StringBuffer and StringBuilder classes?

Since String object is immutable, if we have lot of string manipulations it will end up with lot of abandoned String objects in the String Pool. To overcome this StringBuilder and StringBuffer classes are used, as they can be modified over and over again without leaving behind lot of discarded string objects.

* A common use for StringBuffers and StringBuilders is file I/O when large, ever-changing streams of input are being handled by the program.
* The StringBuffer's API is the same as the StringBuilder's API, **except that StringBuffer's methods are synchronized for thread safety while StringBuilders are not**. Hence StringBuilder methods should run faster than StringBuffer methods.
* All of the following bullets apply to both StringBuffer and StringBuilder:
* They are **mutable**—they can change without creating a new object.
* StringBuffer methods act on the invoking object, and objects can change without an explicit assignment in the statement.
* StringBuffer equals() is not overridden; it doesn't compare values.

### What is i18n?

**i18n** is shorthand for internationalization. Java API provides extensive set of classes to help work with dates, numbers, currencies and locales to cater to customizing results, that are language and regional specific.

### Handling Dates, Numbers & Currencies.

#### Prior to Java8

The four important Date related classes are

1. **java.util.Date**

most of this classes methods are deprecated, but this class can be used to bridge between calendar and DateFormat class.

1. **java.util.Calendar**

this class provides huge variety of methods to helps you convert and manipulate date and time.

1. **java.text.NumberFormat**

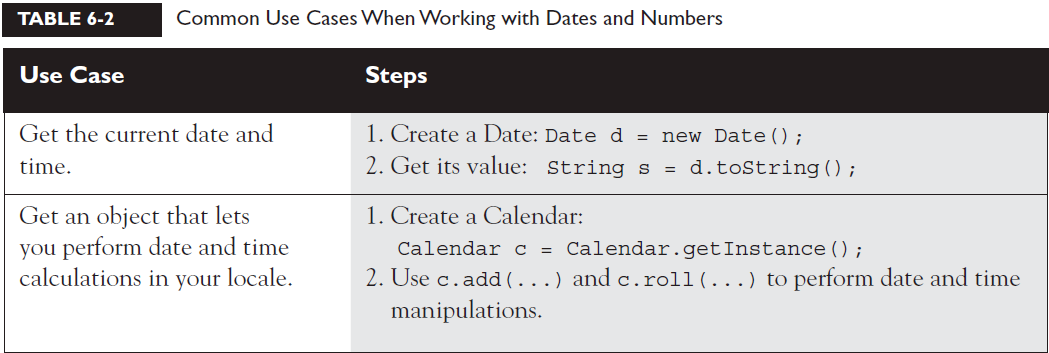
this class is used to format numbers and currencies for locales around the world.

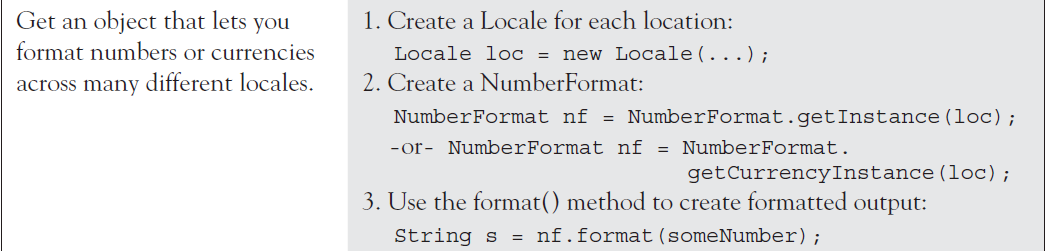
1. **java.text.DateFormat**

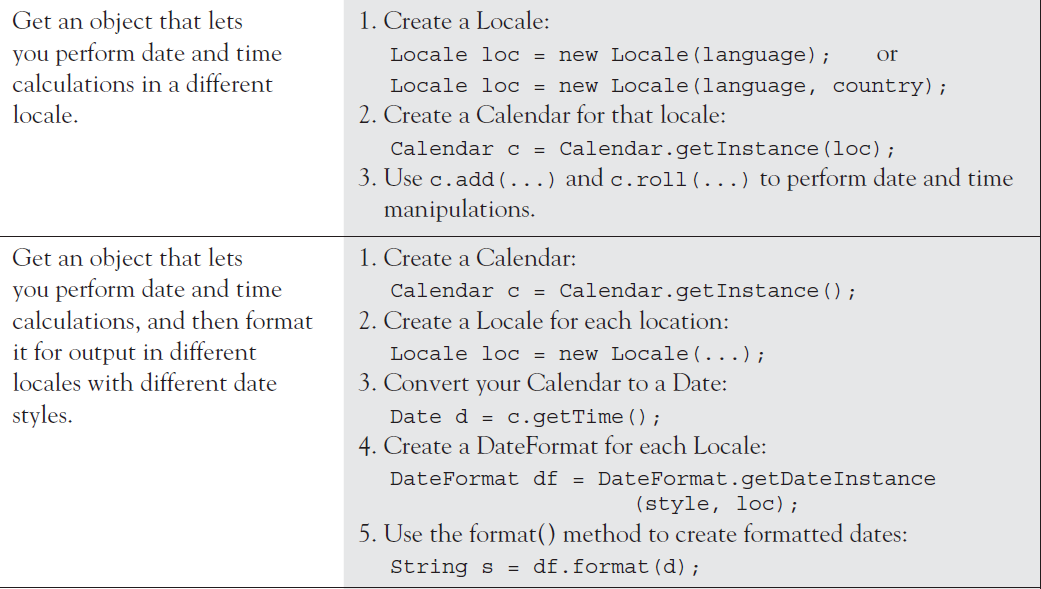
this class is used to format dates for numerous locales around the world.

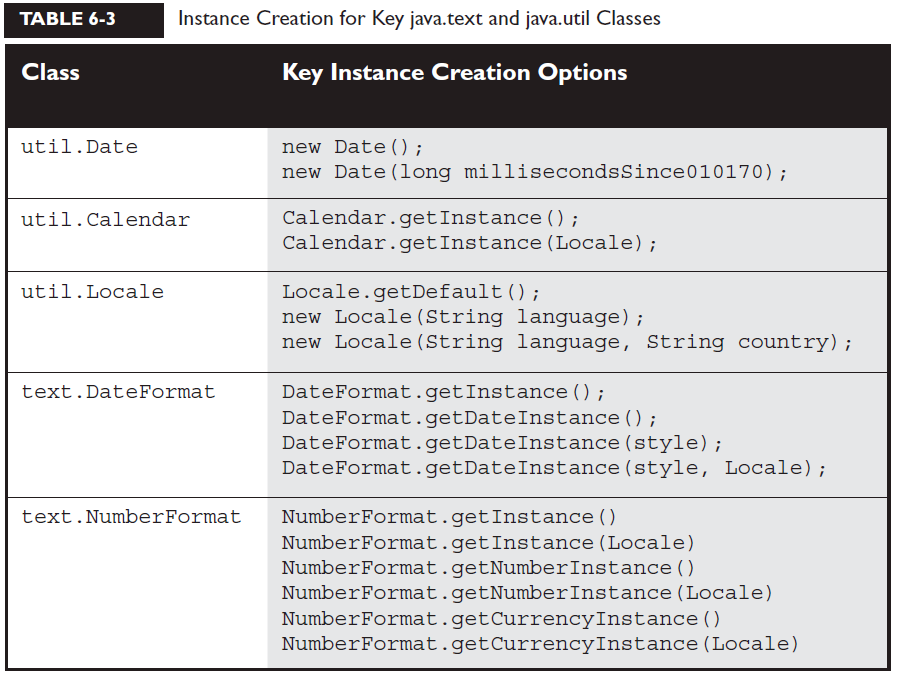
1. **java.util.Locale**

This class is used in conjunction with DateFormat and NumberFormat to format dates, numbers and currency for specific locales.







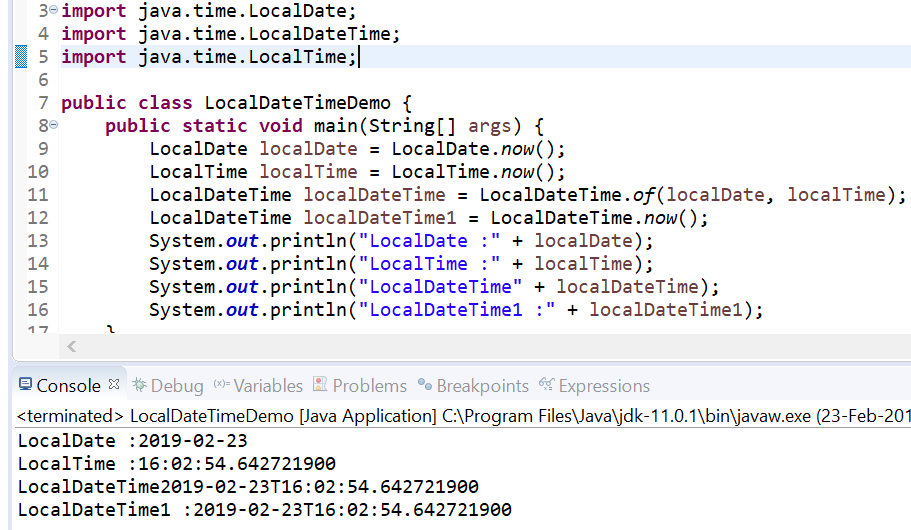


* The classes you need to understand are java.util.Date, java.util.Calendar, java.text.DateFormat, java.text.NumberFormat, and java.util.Locale.
* Most of the Date class's methods have been deprecated.
* A Date is stored as a long, the number of milliseconds since January 1, 1970.
* Date objects are go-betweens the Calendar and Locale classes.
* The Calendar provides a powerful set of methods to manipulate dates, performing tasks such as getting days of the week, or adding some number of months or years (or other increments) to a date.
* Create Calendar instances using static factory methods (getInstance()).
* The Calendar methods you should understand are add(), which allows you to add or subtract various pieces (minutes, days, years, and so on) of dates, and roll(), which works like add() but doesn't increment a date's bigger pieces. (For example: adding 10 months to an October date changes the month to August, but doesn't increment the Calendar's year value.)
* DateFormat instances are created using static factory methods (getInstance() and getDateInstance()).
* There are several format "styles" available in the DateFormat class.
* DateFormat styles can be applied against various Locales to create a wide array of outputs for any given date.
* The DateFormat.format() method is used to create Strings containing properly formatted dates.
* The Locale class is used in conjunction with DateFormat and NumberFormat.
* Both DateFormat and NumberFormat objects can be constructed with a specific, immutable Locale.
* For the exam you should understand creating Locales using language, or a combination of language and country.

#### Since Java8

Since java8 lot of new classes in java.time package are introduced to handle dates and times from around the world. The classes in java.time are organized as below,

**Local dates and times** : these dates and times are local to your timezone and hence don’t have timezone information. Classes such as **LocalDate**, **LocalTime** and **LocalDateTime** represent above.



**Zoned dates and times :** these dates and times include timezone information. Classes such as **ZonedDateTime**, **OffsetDateTime** represent above.

**Formatter for dates and times :** with java.time.format.**DateTimeFormatter** you can parse and print dates and times with patterns and variety of styles.

**Adjustments to dates and times** : with java.time.temporal.**TemporalAdjusters** and java.time.temporal.**ChronoUnit,** we can adjust and manipulate dates and times by handy increments.

**Periods, Durations and Instants** : Periods and Durations represent an amount of time, periods for days or longer, and durations for shorter periods like minutes and seconds. Instants represent specific instant in time.

## File Navigation and I/O

### I/O Classes

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

**File :** class File is an “*abstract representation of file and directory pathnames”*. The File class isn't used to actually read or write data; it's used to work at a higher level, making new empty files, searching for files, deleting files, making directories, and working with paths.

**FileReader :** this class is used to read character files. Its Its read() methods are fairly low-level, allowing you to read single characters, the whole stream of characters, or a fixed number of characters. FileReaders are usually *wrapped* by higher-level objects such as BufferedReaders, which improve performance and provide more convenient ways to work with the data.

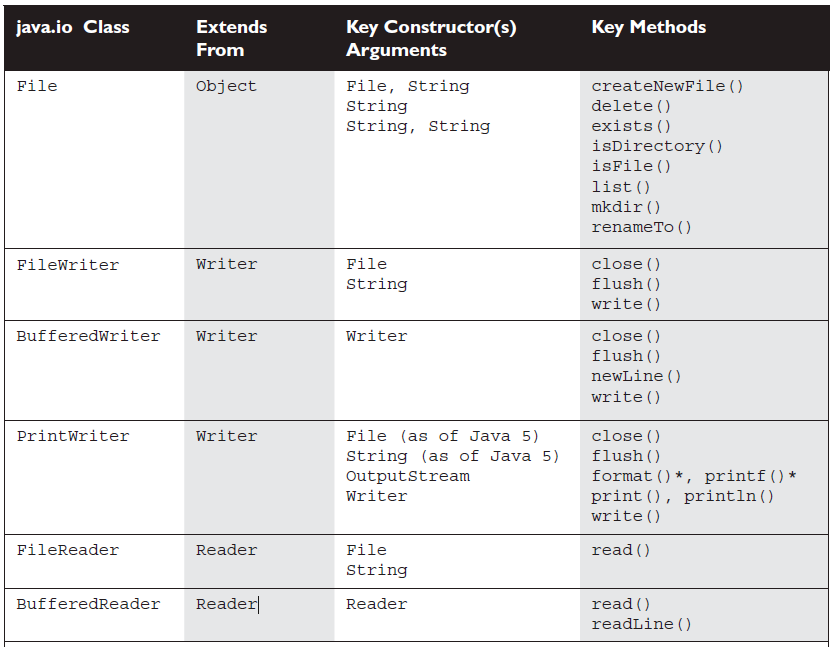
**BufferedReader :** this class is used to make lower-level reader classes like FileReader more efficient and easier to use. BufferedReaders read relatively large chunk of data from a file at once and keep this data in a buffer, which minimizes the number of times the time-intensive file reader operations are performed.

**FileWriter :** This class is used to write to character files. Its write() methods allow you to write character(s) or Strings to a file. FileWriters are usually wrapped by higher-level Writer objects such as BufferedWriters or PrintWriters, which provide better performance and higher-level, more flexible methods to write data.

**BufferedWriter :** This class is used to make lower-level classes like FileWriters more efficient and easier to use. Compared to FileWriters, BufferedWriters write relatively large chunks of data to a file at once.

**PrintWriter**

**Console :** this class provides methods to read input from the console and write formatted output to the console.



* The classes you need to understand in java.io are File, FileReader, BufferedReader, FileWriter, BufferedWriter, PrintWriter, and Console.
* A new File object doesn't mean there's a new file on your hard drive.
* File objects can represent either a file or a directory.
* The File class lets you manage (add, rename, and delete) files and directories.
* The methods createNewFile() and mkdir() add entries to your file system.
* FileWriter and FileReader are low-level I/O classes. You can use them to write and read files, but they should usually be wrapped.
* Classes in java.io are designed to be "chained" or "wrapped." (This is a common use of the decorator design pattern.)
* It's very common to "wrap" a BufferedReader around a FileReader or a BufferedWriter around a FileWriter, to get access to higher-level (more convenient) methods.
* PrintWriters can be used to wrap other Writers, but as of Java 5 they can be built directly from Files or Strings.
* Java 5 PrintWriters have new append(), format(), and printf() methods.
* Console objects can read non-echoed input and are instantiated using System.console().

## Serialization

Serialization lets you save the object and its instance variables, unless the instance variables are explicitly mentioned as **transient**. Classes that need to be serialized must extend **Serializable** interface(marker interface) from java.io package.

Serialization takes care of saving the entire objects “object graph” automatically. Each object internally can refer to futher objects and inturn they can refer to more.. this is called as object graph.

The Java IO API’s that help you to serialize an object are *DataInputStream, DataOutputStream, FileInputStream, FileOutputStream, ObjectInputStream, ObjectOutputStream, and Serializable.*

The magic of basic serialization happens with just two methods: one to serialize objects and write them to a stream, and a second to read the stream and deserialize objects.

* ObjectOutputStream.writeObject() // serialize and write
* ObjectInputStream.readObject() // read and deserialize

The java.io.ObjectOutputStream and java.io.ObjectInputStream classes are considered to be *higher*-level classes in the java.io package, and that means that we'll wrap them around *lower*-level classes, such as java.io.FileOutputStream and java.io.FileInputStream.

If a superclass is serializable, then all subclasses of that class automatically implements serializable implicitly. So if any of a superclass in the inheritance tree implements Serializable, then by default the subclass is serializable.

* The classes you need to understand are all in the java.io package; they include: ObjectOutputStream and ObjectInputStream primarily, and FileOutputStream and FileInputStream because you will use them to create the low-level streams that the ObjectXxxStream classes will use.
* A class must implement Serializable before its objects can be serialized.
* The ObjectOutputStream.writeObject() method serializes objects, and the ObjectInputStream.readObject() method deserializes objects.
* If you mark an instance variable transient, it will not be serialized even thought the rest of the object's state will be.
* You can supplement a class's automatic serialization process by implementing the writeObject() and readObject() methods. If you do this, embedding alls to defaultWriteObject() and defaultReadObject(), respectively, will handle the part of serialization that happens normally.
* If a superclass implements Serializable, then its subclasses do automatically.
* If a superclass doesn't implement Serializable, then when a subclass object is deserialized, the superclass constructor will be invoked, along with its superconstructor(s).

## Parsing , Tokenizing and Formatting

* regex is short for regular expressions, which are the patterns used to search for data within large data sources.
* regex is a sub-language that exists in Java and other languages (such as Perl).
* regex lets you to create search patterns using literal characters or metacharacters. Metacharacters allow you to search for slightly more abstract data like "digits" or "whitespace".

Study the \d, \s, \w, and . metacharacters

* regex provides for quantifiers which allow you to specify concepts like: "look for one or more digits in a row."
* Study the ?, \*, and + greedy quantifiers.
* Remember that metacharacters and Strings don't mix well unless you remember to "escape" them properly. For instance String s = "\\d";
* The Pattern and Matcher classes have Java's most powerful regex capabilities.
* You should understand the Pattern compile() method and the Matcher matches(), pattern(), find(), start(), and group() methods.
* You WON'T need to understand Matcher's replacement-oriented methods.
* You can use java.util.Scanner to do simple regex searches, but it is primarily intended for tokenizing.
* Tokenizing is the process of splitting delimited data into small pieces.
* In tokenizing, the data you want is called tokens, and the strings that separate the tokens are called delimiters.
* Tokenizing can be done with the Scanner class, or with String.split().
* Delimiters are single characters like commas, or complex regex expressions.
* The Scanner class allows you to tokenize Strings or streams or files.
* The String.split() method tokenizes the entire source data all at once, so large amounts of data can be quite slow to process.
* New to Java 5 are two methods used to format data for output. These methods are format() and printf(). These methods are found in the PrintStream class, an instance of which is the out in System.out.
* The format() and printf() methods have identical functionality.
* Formatting data with printf() (or format()) is accomplished using formatting strings that are associated with primitive or string arguments.
* The format() method allows you to mix literals in with your format strings.
* The format string values you should know are

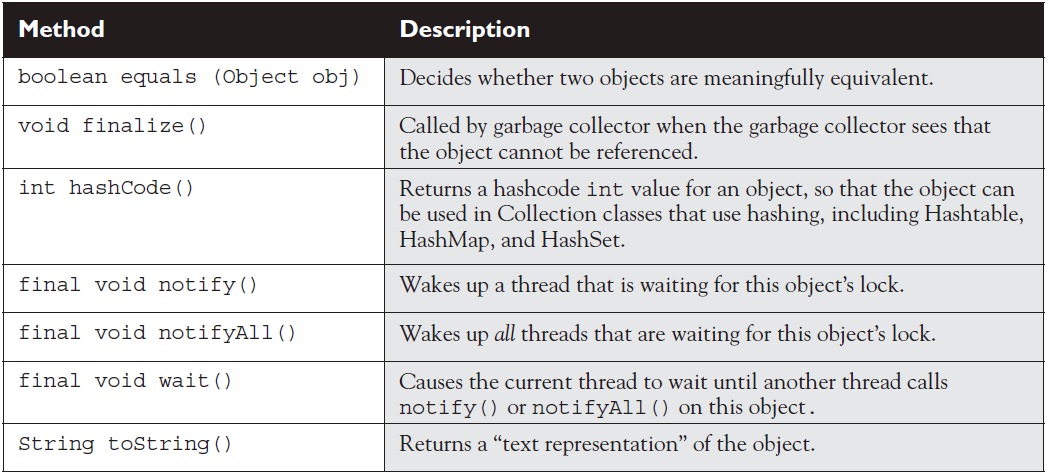
❑ Flags: -, +, 0, "," , and (

❑ Conversions: b, c, d, f, and s

* If your conversion character doesn't match your argument type, an exception will be thrown.

# Chapter 7 : Generics and Collections

Everything in java apart from primitives are Objects. Every class extends from java.lang.Object class. Some of the important methods of Object class are..



## Overriding Object methods

1. **toString()**

when you want a mere mortal to be able to read something meaningful about the objects of your class. Code can call toString() on your object when it wants to read useful details about your object. When you pass an object reference to the System.out.println() method, for example, the object's toString() method is called.

If toString() method of a object is not overridden, and you print it using System.out.print(), it gives you the **class Name** follwed by **@** **symbol**, followed by the **unsigned Hexadecimal** representation of the objects hashcode.

the most common implementations of toString() is to simply spit out the object's state (in other words, the current values of the important instance variables).

1. **equals()**

the String class and the wrapper classes have overridden the equals() method (inherited from class Object), so that you could compare two different objects (of the same type) to see if their contents are meaningfully equivalent.

If you need to know if 2 references are equal or 2 primitives are equal use “==” operator. But to check if two objects themselves are equal , you need to use equals method. For each class you write you must decide if it makes sense to consider two different instances equal. For some classes you might decide that 2 objects can never be equal(example 2 different car objects irrespective of make,model,year , configuration being same cannot be equal).

BUT if you donot override a class’s equals() method, you won’t be able to use these objects as a key in the hashtable and you probably won’t get accurate Sets, such that there are no conceptual duplicates.

The equals() method in class Object uses only the == operator for comparisons, so unless you override equals(), two objects are considered equal only if the two references refer to the same object.

**Equals() contract**

A Java contract is a set of rules that should be followed, or rather must be followed if you want to provide a "correct" implementation as others will expect it to be. Or to put it another way, if you don't follow the contract, your code may still compile and run, but your code (or someone else's) may break at runtime in some unexpected way.

* It is **reflexive**. For any reference value x, x.equals(x) should return true.
* It is **symmetric**. For any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
* It is **transitive**. For any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true.
* It is **consistent**. For any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
* For any non-null reference value x, x.equals(null) should return false.

equals() and hashCode() are bound together by a joint contract that specifies if two objects are considered equal using the equals() method, then they must have **identical hashcode** values.

1. **hashCode()**

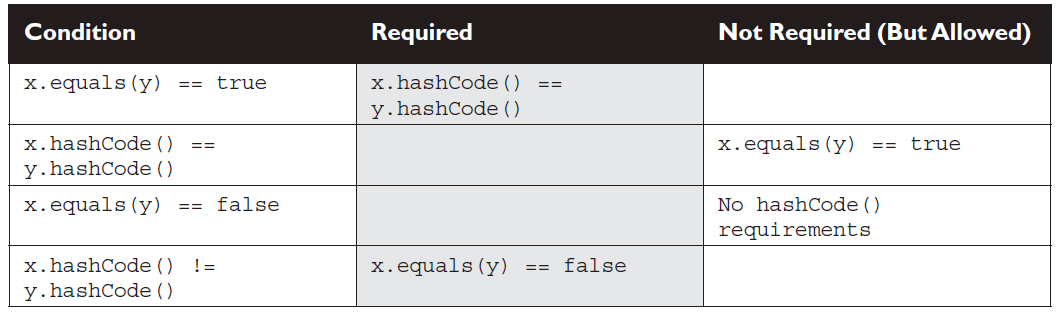
hashcodes are typically used to increase the performance of large collections of data. It is like a **Object ID** Number. A hascode generated via applying a hashCode generation algorithm on a give Object, gives a integer value that helps you to retreive the object later during search from a collection. In real-life hashing, it’s not uncommon to have more than one entry in a bucket(or objects having same hashcode). Hashing retrieval is a two-step process.

1. Find the right bucket (using hashCode()) in which objects can be found.

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

Hashcode contract :

* Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode() method must ***consistently return the same integer***, provided no information used in equals() comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
* If two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must ***produce the same integer result***.
* It is NOT required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode() method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.



As **transient** variables can change their value upon deserialization (reverting back to default value) thehy can really mess with your equals() and hashCode() implementations. Keep variables non-transient or, if they must be marked transient, don't use them to determine hashcodes or equality.

## Collections

The collections framework in java, gives us the data sturcture, classes & interfaces like lists, sets , maps, and queues, to satisfy most of the coding needs.

Few basic operations that can be done using collections are :

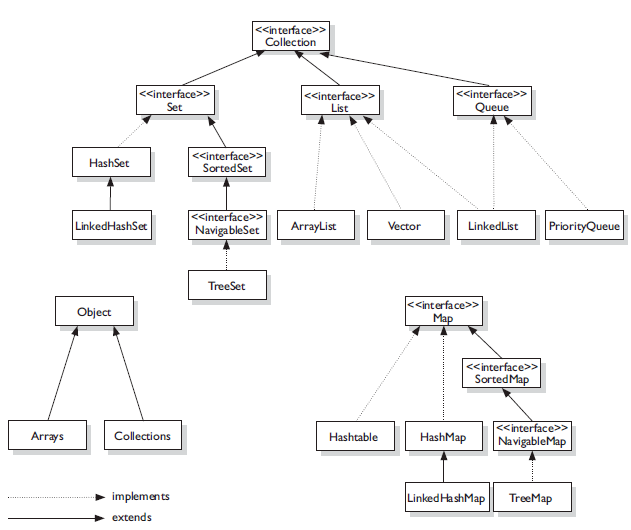
1. Add objects to the collection
2. Remove objects from the collection
3. Find out whether an object(or group of objects) is in a collection
4. Retreive an object from the collection(without removing)
5. Iterate through the collection, looking at each element (object) one after the other.

Some of the core interfaces of collections framework are

|  |  |  |
| --- | --- | --- |
| Collection | Set | SortedSet |
| List | Map | SortedMap |
| Queue | NavigableSet | NavigableMap |

Some of the important core concrete implementation classes are

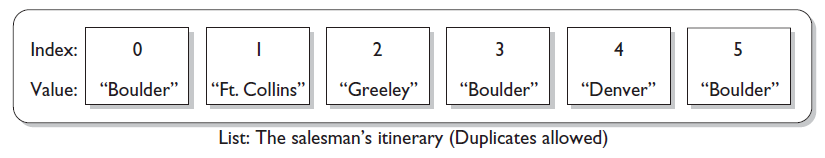
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Maps** | **Sets** | **Lists** | **Queues** | **Utilities** |
| HashMap | HashSet | ArrayList | PriorityQueue | Collections |
| HashTable | LinkedHashSet | Vector |  | Arrays |
| TreeMap | TreeSet | LinkedList |  |  |
| LinkedHashMap |  |  |  |  |

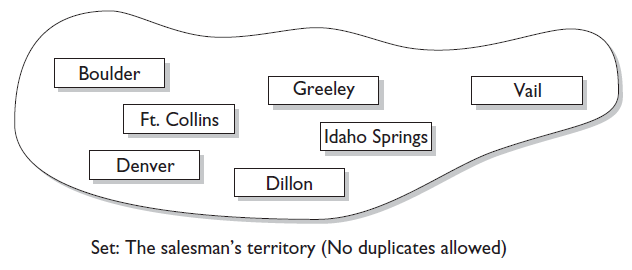


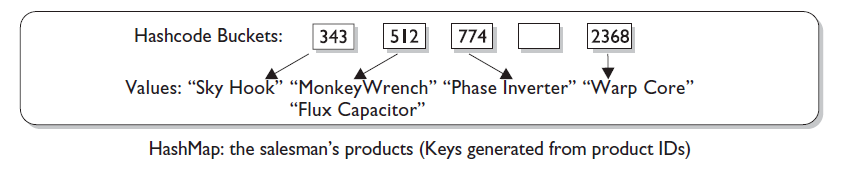
### Collections class

Collections class has many static utility methods. Collections come in basic 4 flavors.

1. Lists – Lists of things (classes that implement List)
2. Sets – Unique things (classes that implement Set)
3. Maps – Things with a Unique ID(classes that implement Map)
4. Queues – Things arranged by the order in which they are to be processed.







Each of these flavors have sub-flavors

1. Sorted 2) Unsorted 3) Ordered 4) Unordered

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 because sorting is a specific type of ordering.

Iterating through a collection usually means walking through the elements one after another starting from the first element. Sometimes, though, even the concept of first is a little strange—in a Hashtable there really isn't a notion of first, second, third, and so on. In a Hashtable, the elements are placed in a (seemingly) chaotic order based on the hashcode of the key.

When a collection is **ordered** it means that you can iterate through the collection in a specific order. A **sorted** collection means that the order in the collection is determined according to some rule or rules known as the **sort order**. A sort order has nothing to do with when an object was added to the collection, or when was the last time it was accessed, or what "position" it was added at. Sorting is done based on properties of the objects themselves. You put objects into the collection, and the collection will figure out what order to put them in, based on the sort order. A collection that keeps an order (such as any List, which uses insertion order) is not really considered sorted unless it sorts using some kind of sort order. Most commonly, the sort order used is something called the natural order.

For a collection of strings the natural order is alphabetic, for Integer objects natural order is numeric value. The natural order of other custom Objects is set by implementing the **Comparable** interface that defines how instance of a class be compared to another one. 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

A List cares about index. The one thing that List has that non-lists don't have is a set of methods related to the index. Those key methods include things like **get**(int index), **indexOf**(Object o), **add**(int index, Object obj), and so on.

1. **ArrayList** : this is like a growable array. It gives fast iteration and fast random access. It is a ordered collection(by index) but not sorted. ArrayList implements RandomAccess interface(a marker interface meaning it has no methods) that says “this list supports fast random access”.

arrayList is like an array++. Advantages of ArrayList over array are,

* + It can grow dynamically.
  + It provides more powerful insertion and search mechanisms than arrays.

1. **Vector** : A Vector is basically the same as an ArrayList, but vector methods are synchronized for thread safety.
2. **LinkedList** : A linked list is an ordered by index position, like ArrayList except that the elements are doubly linked to one another. This linkage gives you new methods (beyond what you get from the List interface) for adding and removing from the beginning or end, which makes it an easy choice for implementing a stack or queue. Keep in mind that a LinkedList may iterate more slowly than an ArrayList, but it's a good choice when you need fast insertion and deletion. LinkedList class also implements java.util.Queue interface, hence it supports queue methods like peek(), poll() and offer().

### Set Interface

A set cares about uniqueness – it doesn’t allow duplicates. The equals() method is used to determine whether two objects are identical and duplicates are not inserted if one already exist.

1. **HashSet :**  A hashset is a unsorted, unordered Set. It uses the HashCode of the object being inserted, so the more efficient your hashCode() impmementation the better access performance you’ll get. Use this class when you want a collection with no duplicates and you don’t care about order when you iterate through it.
2. **LinkedHashSet :** A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked list across all elements. Use this class instead of HashSet when you care about the iteration order. When iterated through the HashSet the order is upredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.
3. **TreeSet :** the TreeSet is a sorted collection that guarantees that the elements will be in ascending order, according to natural order. You can also construct a TreeSet with a constructor that lets you give the collection where your own rules for what the order should be(rather than relying on the ordering defined by elements class)by using a Comparator or Comparable. TreeSet implements NavigableSet.

When using **HashSet** or **LinkedHashSet**, the objects you add to them must override hashCode(). If they don’t override hashCode(), the default Object. hashCode() method will allow multiple objects that you might consider "meaningfully equal" to be added to your "no duplicates allowed" set.

### Map Interface

A Map cares about unique identifiers. You map a unique key (the ID) to a specific value, where both the key and the value are, of course, objects. 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.

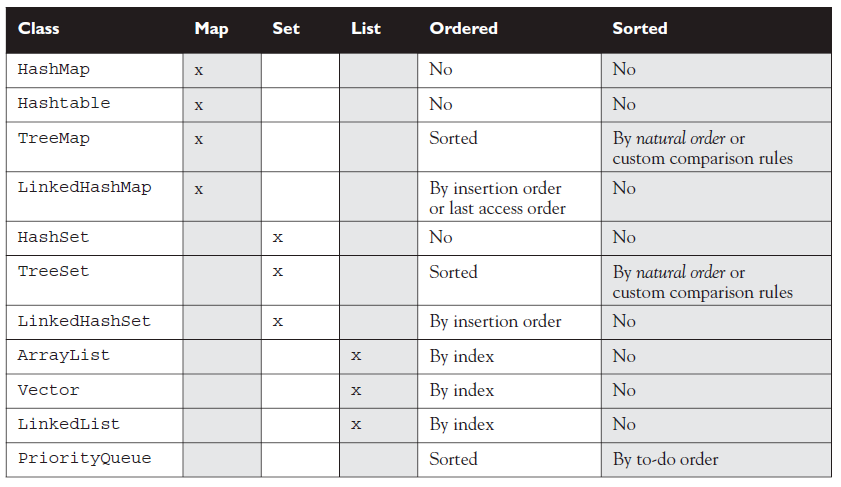
1. **HashMap :** The HashMap gives you an unsorted, unordered Map. When you need a Map and you don't care about the order (when you iterate through it), then HashMap is the way to go; the other maps add a little more overhead. Where the keys land in the Map is based on the key's hashcode, so, like HashSet, the more efficient your hashCode() implementation, the better access performance you'll get. HashMap allows one null key and multiple null values in a collection.
2. **Hashtable :**  Hashtable is the synchronized counterpart to HashMap, the key functions of Hashtable are synchronized for thread safety. A Hashtable doesn’t allow null values.
3. **LinkedHashMap**
4. **TreeMap**

### Queue Interface

A Queue is designed to hold a list of "to-dos," or things to be processed in some way. Although other orders are possible, queues are typically thought of as FIFO (first-in, first-out). Queues support all of the standard Collection methods and they also add methods to add and subtract elements and review queue elements.

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.



## Using Collections Framework

### Sorting Collections

java.util.Collections provide sort() method that can be used to sort collections. the sort() method takes a List argument, and that the objects in the List must implement an interface called Comparable. String and wrapper classes implement 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(). Here's an invocation of 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)

Another way to sort collections is to implement **Comparator** interface. The overloaded sort() method takes both a list and 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()**.

**sorting with Arrays Class**

The Arrays.sort() method is overridden in the same way the Collections.sort() method is.

■ Arrays.sort(arrayToSort)

■ Arrays.sort(arrayToSort, Comparator)

In addition, the Arrays.sort() method is overloaded about a million times to provide a couple of sort methods for every type of primitive. The Arrays.sort() methods that sort primitives always sort based on natural order.

# Chapter 8 : Inner Classes

Inner classes let you define one class **within** another. They provide a type of scoping for your classes since you can make one class a member of another class. Just as classes have member variables and methods, a class can also have member **classes**.

There are 4 flavors of inner classes

1. Normal/regular inner classes
2. Static inner classes
3. Method-Local inner classes
4. Anonymous inner classes

**Instantiating an Inner Class**

To create an instance of an inner class, *you must have an instance of the outer class* to tie to the inner class. There are no exceptions to this rule: an inner class instance can never stand alone without a direct relationship to an instance of the outer class.

the rules for an inner class referencing itself or the outer instance are as follows:

- To reference the inner class instance itself, from within the inner class code, use this.

- To reference the "outer this" (the outer class instance) from within the inner class code, use NameOfOuterClass.this (example, MyOuter.this).

**Member Modifiers Applied to Inner Classes**

A regular inner class is a member of the outer class just as instance variables and methods are, so the following modifiers can be applied to an inner class:

* final
* abstract
* public
* private
* protected
* static—*but* static *turns it into a* static *nested class not an inner class*
* strictfp

**Method-Local Inner Class**

*A method-local inner class can be instantiated only within the method where the inner class is defined*. In other words, no other code running in any other method—inside or outside the outer class—can ever instantiate the method-local inner class. Like regular inner class objects, the method-local inner class object shares a special relationship with the enclosing (outer) class object, and can access its private (or any other) members. However, *the inner class object cannot use the local variables of the method the inner class is in*.

Modifiers within a method: the same rules apply to method-local inner classes as to local variable declarations. You can't, for example, mark a method-local inner class public, private, protected, static, transient, and the like. The only modifiers you *can* apply to a method-local inner class are abstract and final, but as always, never both at the same time.

a local class declared in a **static** method has access to only static members of the enclosing class, since there is no associated instance of the enclosing class. If you're in a static method there is no this, so an inner class in a static method is subject to the same restrictions as the static method. In other words, no access to instance variables.

## Inner Classes

* A "regular" inner class is declared *inside* the curly braces of another class, but *outside* any method or other code block.
* An inner class is a full-fledged member of the enclosing (outer) class, so it can be marked with an access modifier as well as the abstract or final modifiers. (Never both abstract and final together— remember that abstract *must* be subclassed, whereas final *cannot* be subclassed).
* An inner class instance shares a special relationship with an instance of the enclosing class. This relationship gives the inner class access to *all* of the outer class's members, including those marked private.
* To instantiate an inner class, you must have a reference to an instance of the outer class.
* From code within the enclosing class, you can instantiate the inner class using only the name of the inner class, as follows: MyInner mi = new MyInner();
* From code outside the enclosing class's instance methods, you can instantiate the inner class only by using both the inner and outer class names, and a reference to the outer class as follows:

MyOuter mo = new MyOuter();

MyOuter.MyInner inner = mo.new MyInner();

* From code within the inner class, the keyword this holds a reference to the inner class instance. To reference the *outer* this (in other words, the instance of the outer class that this inner instance is tied to) precede the keyword this with the outer class name as follows: MyOuter.this;

## Method local inner classes

* A method-local inner class is defined within a method of the enclosing class.
* For the inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but after the class definition code.
* A method-local inner class cannot use variables declared within the method (including parameters) unless those variables are marked final.
* The only modifiers you can apply to a method-local inner class are abstract and final. (Never both at the same time, though.)

## Anonymous Inner Classes

* Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named interface.
* An anonymous inner class is always created as part of a statement; don't forget to close the statement after the class definition with a curly brace. This is a rare case in Java, a curly brace followed by a semicolon.
* Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.
* An anonymous inner class can extend one subclass or implement one interface. Unlike non-anonymous classes (inner or otherwise), an anonymous inner class cannot do both. In other words, it cannot both extend a class and implement an interface, nor can it implement more than one interface.
* An argument-defined inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: });

## Static nested classes

* Static nested classes are inner classes marked with the static modifier.
* A static nested class is not an inner class, it's a top-level nested class.
* Because the nested class is static, it does not share any special relationship with an instance of the outer class. In fact, you don't need an instance of the outer class to instantiate a static nested class.
* Instantiating a static nested class requires using both the outer and nested class names as follows:

BigOuter.Nested n = new BigOuter.Nested();

* A static nested class cannot access non-static members of the outer class, since it does not have an implicit reference to any outer instance (in other words, the nested class instance does not get an outer this reference).

# Chapter 9 – Threads

## Thread Basics

A thread is a sequence of code under execution having it’s own method stack and running concurrently with other threads. In Java “Thread ” means 2 different things

1. an Instance of class java.lang.Thread
2. a thread of execution

An instance of Thread is just an object, that has variables and methods and lives and dies on the heap. But a thread of exection is an individual process that has its own call stack. There is one thread per call stack, or one call stack per thread.

The main() method that starts the whole ball rolling, runs in one thread, called **main** thread.

You can define and instantiate a thread in one of the 2 ways:

1. extend java.lang.Thread class (and overriding run() method)
2. implement the Runnable Interface.(and implementing run() method)

Extending the Thread class is the easiest, but it's usually not a good OO practice. Why? Because subclassing should be **reserved for specialized versions** of more general superclasses. So the only time it really makes sense (from an OO perspective) to extend Thread is when you have a more specialized version of a Thread class. In other words, because *you have more specialized thread-specific behavior*. Chances are, though, that the thread work you want is really just a job to be done *by* a thread. In that case, you should design a class that implements the Runnable interface, which also leaves your class **free to extend** some *other* class.

## Thread Constructors and Thread Execution

There are 4 main type of Thread constructors..

1. Thread()
2. Thread(String threadName)
3. Thread(Runnable target)
4. Thread(Runnable target,String threadName)

Where target is the instance with the job to do.

When a thread has been instantiated but not started (in other words, the **start()** method has not been invoked on the Thread instance), the thread is said to be in the **new** state. At this stage, the thread is not yet considered to be **alive**. Once the start() method is called, the thread is considered to be alive (even though the run() method may not have actually started executing yet). A thread is considered **dead** (no longer alive) after the run() method **completes**.

You've created a Thread object and it knows its target (either the passed-in Runnable or itself if you extended class Thread). Now it's time to get the whole thread thing happening—to launch a new call stack. It's achieved by calling:

**t.start()**

Prior to calling start() on a Thread instance, the thread is said to be in the new state as we said. The new state means you have a Thread object but you don't yet have a true thread. So what happens after call to start()?

- A new thread of execution starts (with a new call stack).

- The thread moves from the new state to the runnable state.

- When the thread gets a chance to execute, its target run() method will run.

You start a Thread, not a Runnable. You call start() on a Thread instance, not on a Runnable instance.

*A thread is done being a thread when its target* ***run()*** *method completes.*

When a thread completes its run() method, the thread ceases to be a thread of execution. The stack for that thread dissolves, and the thread is considered dead. Not dead and gone, however, just dead. It's still a Thread *object*, just not a ***thread of execution***. So if you've got a reference to a Thread instance, then even when that Thread instance is no longer a thread of execution, you can still call methods on the Thread instance, just like any other Java object. What you can't do, though, is call start() again.

*Once a thread has been started, it can never be started again. If start() method is called second time it will cause an exception ( IllegalThreadStateException).*

Reference to the current running thread is obtained by calling the static method

**Thread.getCurrentThread()**

Some of the methods that can help us influence thread scheduling are as follows:

public static void **sleep**(long millis) throws InterruptedException

public static void **yield**()

public final void **join**() throws InterruptedException

public final void **setPriority**(int newPriority)

Every class also inherits following methods from Object class,

public final void **wait**() throws InterruptedException

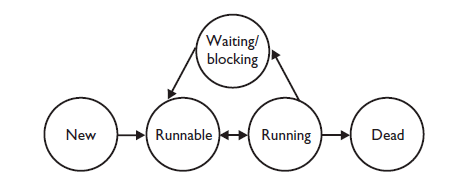
public final void **notify**()

public final void **notifyAll**()

## Thread States and Transitions

A thread can be in one of the below five states

1. **New** : This is the state the thread is in after the Thread instance has been created, but the start() method has not been invoked on the thread. It is a live Thread object, but not yet a thread of execution. At this point, the thread is considered ***not*** *alive*.
2. **Runnable :** This is the state a thread is in when it's eligible to run, but the scheduler has not selected it to be the running thread. A thread first enters the runnable state when the start() method is invoked, but a thread can also return to the runnable state after either running or coming back from a blocked, waiting, or sleeping state. When the thread is in the runnable state, it is considered **alive**.
3. **Running :** This is the state a thread is in when the thread scheduler selects it (from the runnable pool) to be the currently executing process. A thread can transition out of a running state for several reasons, including because "the thread scheduler felt like it." there are several ways to get to the runnable state, but only one way to get to the running state: the scheduler chooses a thread from the runnable pool.



1. **Waiting/blocked/sleeping :** This is the state a thread is in when it's not eligible to run. the thread is **still** alive, but is currently not eligible to run. In other words, it is not *runnable*, but it might *return* to a runnable state later if a particular event occurs. A thread may be *blocked* waiting for a resource (like I/O or an object's lock), in which case the event that sends it back to runnable is the availability of the resource—for example, if data comes in through the input stream the thread code is reading from, or if the object's lock suddenly becomes available. A thread may be *sleeping* because the thread's run code *tells* it to sleep for some period of time, in which case the event that sends it back to runnable is that it wakes up because its sleep time has expired. Or the thread may be *waiting*, because the thread's run code *causes* it to wait, in which case the event that sends it back to runnable is that another thread sends a notification that it may no longer be necessary for the thread to wait.
2. **Dead :** A thread is considered dead when its run() method completes. It may still be a viable Thread object, but it is no longer a separate thread of execution. Once a thread is dead, it can never be brought back to life! If you invoke start() on a dead Thread instance, you'll get a runtime (not compiler) exception.

## Thread Priorities and Yield()

Threads always run with some priority, usually represented as a **number between 1 and 10**.The scheduler in most JVMs uses preemptive, priority-based scheduling (which implies some sort of time slicing). *This does not mean that all JVMs use time slicing.* The JVM specification does not require a VM to implement a time-slicing scheduler, where each thread is allocated a fair amount of time and then sent back to runnable to give another thread a chance. Although many JVMs do use time slicing, some may use a scheduler that lets one thread stay running until the thread completes its run() method.

In most JVMs, however, the scheduler does use thread priorities in one important way: If a thread enters the runnable state, and it has a higher priority than any of the threads in the pool and a higher priority than the currently running thread, the lower-priority running thread usually will be bumped back to runnable and the highest-priority thread will be chosen to run. In other words, *at any given time the currently running thread usually will not have a priority that is lower than any of the threads in the pool.* In most cases, the running thread will be **of equal or greater priority** than the highest priority threads in the pool. This is as close to a guarantee about scheduling as you'll get from the JVM specification, so you must never rely on thread priorities to guarantee the correct behavior of your program.

**Setting a Thread's Priority** A thread gets a default priority that is *the priority of the thread of execution that creates it.* For example, in the code

public class TestThreads

{

public static void main (String [] args)

{

MyThread t = new MyThread();

}

}

the thread referenced by t will have the same priority as the *main* thread, since the main thread is executing the code that creates the MyThread instance. You can also set a thread's priority directly by calling the **setPriority()** method on a Thread instance.

the **static** **Thread.yield()** make the currently running thread head back to runnable to allow other threads of the same priority to get their turn. So the intention is to use yield() to promote graceful turn-taking among equal-priority threads. In reality, though, the yield() method isn't **guaranteed** to do what it claims, and even if yield() does cause a thread to step out of running and back to runnable, *there's no guarantee the yielding thread won't just be chosen again over all the others!* So while yield() might—and often does—make a running thread give up its slot to another runnable thread of the same priority, there's no guarantee. A yield() won't ever cause a thread to go to the waiting/sleeping/ blocking state. At most, a yield() will cause a thread to go from running to runnable, but again, it might have no effect at all.

The **non-static join()** method of class Thread lets one thread "join onto the end" of another thread. If you have a thread B that can't do its work until another thread A has completed its work, then you want thread B to "join" thread A. This means that thread B will not become runnable until A has finished (and entered the dead state).

## Synchronization and Locks

Every object in Java has a built-in lock that only comes into play when the object has synchronized method code. When we enter a synchronized non-static method, we automatically acquire the lock associated with the current instance of the class whose code we're executing (the this instance). Acquiring a lock for an object is also known as getting the lock, or locking the object, locking *on* the object, or synchronizing on the object. The term *monitor is also used* to refer to the object whose lock we're acquiring. Technically the lock and the monitor are two different things, but most people talk about the two interchangeably. Since there is only one lock per object, if one thread has picked up the lock, no other thread can pick up the lock until the first thread releases (or returns) the lock. This means no other thread can enter the synchronized code (which means it can't **enter any synchronized** method of that object) until the lock has been released. Typically, releasing a lock means the thread holding the lock (in other words, the thread currently in the synchronized method) exits the synchronized method. At that point, the lock is free until some other thread enters a synchronized method on that object.

Key points about synchronization are :

1. Only methods(or blocks) can be synchronized, not variables or classes.
2. Each object has just one lock.
3. Not all methods in a class need to be synchronized, a class can have both synchronized and non-sunchronized methods.
4. If two threads are about to execute a synchronized method on the same instance of the class, only one thread at a time will be able to execute the method. Meaning once a thread acquires the lock on an object, no other thread can enter any of the synchronized methods in that class(for that object).
5. If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods!
6. If a thread goes to sleep, it holds any locks it has – it doesn’t release them.
7. A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again. Also, if a thread acquires a lock and then attempts to call a synchronized method on that same object, no problem. The JVM knows that this thread already has the lock for this object, so the thread is free to call other synchronized methods on the same object, using the lock the thread already has.
8. You can synchronize a block of code rather than a method.
9. Because synchronization does hurt concurrency, you don't want to synchronize any more code than is necessary to protect your data. So if the scope of a method is more than needed, you can reduce the scope of the synchronized part to something less than a full method—to just a block. We call this, a **synchronized block**.
10. If a thread tries to enter a synchronized method and the lock is already taken, the thread is said to be **blocked** on the object's lock. Essentially, the thread goes into a kind of pool for that particular object and has to sit there until the lock is released and the thread can again become runnable/running. Just because a lock is released doesn't mean any particular thread will get it. There might be three threads waiting for a single lock, for example, and there's no guarantee that the thread that has waited the longest will get the lock first.
11. There can be situations where mere synchronizing methods and blocks of code is not sufficient to achieve synchronization(or avoid race condition).

**For ex** ; what if you have a non-static method that accesses a static field? Or a static method that accesses a non-static field (using an instance)? In these cases things start to get messy quickly, and there's a very good chance that things will not work the way you want. If you've got a static method accessing a non-static field, and you synchronize the method, you acquire a lock on the Class object. But what if there's another method that also accesses the non-static field, this time using a non-static method? It probably synchronizes on the current instance (this) instead. Remember that a static synchronized method and a non-static synchronized method will not block each other—they can run at the same time. Similarly, if you access a static field using a non-static method, two threads might invoke that method using two different this instances. Which means they won't block each other, because they use different locks. Which means two threads are simultaneously accessing the same static field—exactly the sort of thing we're trying to prevent.

**To keep things simple:** in order to make a class thread-safe, methods that access changeable fields need to be synchronized. *Access to static fields should be done from* static synchronized *methods. Access to non-*static *fields should be done from non-*static synchronized *methods.*

## Thread Summary

* Threads can be created by extending Thread and overriding the public void run() method.
* Thread objects can also be created by calling the Thread constructor that takes a Runnable argument. The Runnable object is said to be the *target* of the thread.
* You can call start() on a Thread object only once. If start() is called more than once on a Thread object, it will throw a RuntimeException.
* It is legal to create many Thread objects using the same Runnable object as the target.
* When a Thread object is created, it does not become a *thread of execution* until its start() method is invoked. When a Thread object exists but hasn't been started, it is in the *new* state and is not considered *alive*.
* Once a new thread is started, it will always enter the runnable state.
* The thread scheduler can move a thread back and forth between the runnable state and the running state.
* For a typical single-processor machine, only one thread can be running at a time, although many threads may be in the runnable state.
* There is no guarantee that the order in which threads were started determines the order in which they'll run.
* There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns regardless of the underlying JVM, you can use the sleep() method. This prevents one thread from hogging the running process while another thread starves. (In most cases, though, yield() works well enough to encourage your threads to play together nicely.)
* A running thread may enter a blocked/waiting state by a wait(), sleep(), or join() call.
* A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.
* When the sleep or wait is over, or an object's lock becomes available, the thread can only reenter the runnable state. It will go directly from waiting to running (well, for all practical purposes anyway).
* A dead thread cannot be started again.
* Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep. A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the sleep() method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
* The sleep() method is a static method that sleeps the currently executing thread's state. One thread cannot tell another thread to sleep.
* The setPriority() method is used on Thread objects to give threads a priority of between 1 (low) and 10 (high), although priorities are not guaranteed, and not all JVMs recognize 10 distinct priority levels—some levels may be treated as effectively equal.
* If not explicitly set, a thread's priority will have the same priority as the priority of the thread that created it.
* The yield() method may cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a different thread selected to run. A thread might yield and then immediately reenter the running state.
* The closest thing to a guarantee is that at any given time, when a thread is running it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will usually preempt the running low-priority thread and put the high-priority thread in.
* When one thread calls the join() method of another thread, the currently running thread will wait until the thread it joins with has completed. Think of the join() method as saying, "Hey thread, I want to join on to the end of you. Let me know when you're done, so I can enter the runnable state."
* synchronized methods prevent more than one thread from accessing an object's critical method code simultaneously. You can use the synchronized keyword as a method modifier, or to start a synchronized block of code.
* To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to synchronize on.
* While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's unsynchronized code.
* When a thread goes to sleep, its locks will be unavailable to other threads.
* static methods can be synchronized, using the lock from the java.lang.Class instance representing that class.
* The wait() method lets a thread say, "there's nothing for me to do now, so put me in your waiting pool and notify me when something happens that I care about." Basically, a wait() call means "wait me in your pool," or "add me to your waiting list."
* The notify() method is used to send a signal to one and only one of the threads that are waiting in that same object's waiting pool.
* The notify() method can NOT specify which waiting thread to notify.
* The method notifyAll() works in the same way as notify(), only it sends the signal to all of the threads waiting on the object.
* All three methods—wait(), notify(), and notifyAll()—must be called from within a synchronized context! A thread invokes wait() or notify() on a particular object, and the thread must currently hold the lock on that object.
* Deadlocking is when thread execution grinds to a halt because the code is waiting for locks to be removed from objects. Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words, both threads are waiting for each other's locks to be released; therefore, the locks will never be released! Deadlocking is bad. Don't do it.

## Stack , Heap and scope

* Local variables (method variables) live on the stack.
* Objects and their instance variables live on the heap.
* Scope refers to the lifetime of a variable.
* There are four basic scopes:

❑ Static variables live basically as long as their class lives.

❑ Instance variables live as long as their object lives.

❑ Local variables live as long as their method is on the stack; however, if their method invokes

another method, they are temporarily unavailable.

❑ Block variables (e.g., in a for or an if) live until the block completes.

## Literals ,Primitive Casting and Assignments

* Integer literals can be decimal, octal (e.g. 013), or hexadecimal (e.g. 0x3d).
* Literals for longs end in L or l.
* Float literals end in F or f, double literals end in a digit or D or d.
* The boolean literals are true and false.
* Literals for chars are a single character inside single quotes: 'd'.
* Literal integers are implicitly ints.
* Integer expressions always result in an int-sized result, never smaller.
* Floating-point numbers are implicitly doubles (64 bits).
* Narrowing a primitive truncates the *high order* bits.
* Compound assignments (e.g. +=), perform an automatic cast.
* A reference variable holds the bits that are used to refer to an object.
* Reference variables can refer to subclasses of the declared type but not to superclasses.

## Using a variable, and with methods

* When an array of objects is instantiated, objects within the array are not instantiated automatically, but all the references get the default value of null.
* When an array of primitives is instantiated, elements get default values.
* Instance variables are always initialized with a default value.
* Local/automatic/method variables are never given a default value. If you
* attempt to use one before initializing it, you'll get a compiler error.
* Methods can take primitives and/or object references as arguments.
* Method arguments are always copies.
* Method arguments are never actual objects (they can be references to objects).
* A primitive argument is an unattached copy of the original primitive.
* A reference argument is another copy of a reference to the original object.
* Shadowing occurs when two variables with different scopes share the same name. This leads to hard-to-find bugs.

## Array Declaration, construction and initialization

* Arrays can hold primitives or objects, but the array itself is always an object.
* When you declare an array, the brackets can be left or right of the name.
* It is never legal to include the size of an array in the declaration.
* You must include the size of an array when you construct it (using new) unless you are creating an anonymous array.
* Elements in an array of objects are not automatically created, although primitive array elements are given default values.
* You'll get a NullPointerException if you try to use an array element in an object array, if that element does not refer to a real object.
* Arrays are indexed beginning with zero.
* An ArrayIndexOutOfBoundsException occurs if you use a bad index value.
* Arrays have a length variable whose value is the number of array elements.
* The last index you can access is always one less than the length of the array.
* Multidimensional arrays are just arrays of arrays.
* The dimensions in a multidimensional array can have different lengths.
* An array of primitives can accept any value that can be promoted implicitly to the array's declared type;. e.g., a byte variable can go in an int array.
* An array of objects can hold any object that passes the IS-A (or instanceof) test for the declared type of the array. For example, if Horse extends Animal, then a Horse object can go into an Animal array.
* If you assign an array to a previously declared array reference, the array you're assigning must be the same dimension as the reference you're assigning it to.
* You can assign an array of one type to a previously declared array reference of one of its supertypes. For example, a Honda array can be assigned to an array declared as type Car (assuming Honda extends Car).

## Initialization blocks

* Static initialization blocks run once, when the class is first loaded.
* Instance initialization blocks run every time a new instance is created. They run after all super-constructors and before the constructor's code has run.
* If multiple init blocks exist in a class, they follow the rules stated above, AND they run in the order in which they appear in the source file.

## Wrappers,Boxing and Overloading

* The wrapper classes correlate to the primitive types.
* Wrappers have two main functions:

❑ To wrap primitives so that they can be handled like objects

❑ To provide utility methods for primitives (usually conversions)

* The three most important method families are

❑ xxxValue() Takes no arguments, returns a primitive

❑ parseXxx() Takes a String, returns a primitive, throws NFE(numberFormatException)

❑ valueOf() Takes a String, returns a wrapped object, throws NFE

* Wrapper constructors can take a String or a primitive, except for Character, which can only take a char. Radix refers to bases (typically) other than 10; octal is radix = 8, hex = 16.
* As of Java 5, boxing allows you to convert primitives to wrappers or to convert wrappers to primitives automatically.
* Using == with wrappers created through boxing is tricky; those with the same small values (typically lower than 127), will be ==, larger values will not be ==.
* Primitive widening uses the "smallest" method argument possible.
* Used individually, boxing and var-args are compatible with overloading.
* You CANNOT widen from one wrapper type to another. (IS-A fails.)
* You CANNOT widen and then box. (An int can't become a Long.)
* You can box and then widen. (An int can become an Object, via an Integer.)
* You can combine var-args with either widening or boxing.

## Garbage Collection

* In Java, garbage collection (GC) provides automated memory management.
* The purpose of GC is to delete objects that can't be reached.
* Only the JVM decides when to run the GC, you can only suggest it.
* You can't know the GC algorithm for sure.
* Objects must be considered eligible before they can be garbage collected.
* An object is eligible when no live thread can reach it.
* To reach an object, you must have a live, reachable reference to that object.
* Java applications can run out of memory.
* Islands of objects can be GCed, even though they refer to each other.
* Request garbage collection with System.gc(); (only before the SCJP 6).
* Class Object has a finalize() method.
* The finalize() method is guaranteed to run once and only once before the garbage collector deletes an object.
* The garbage collector makes no guarantees, finalize() may never run.
* You can uneligibilize an object for GC from within finalize().