**General**:

Versioning:

Java 8:

Lambda Expressions

SplitIterator

Type Annotation

Java 7:

Multiple exception handling

Strings in switch Statement

Try with Resources

Java 6:

Integrated Web Services.

JDBC 4.0 API

Java Compiler API

Java 5:

Annotations

Static Import

Generics

Enhanced for Loop

Autoboxing/Unboxing

* Java 5. Annotation is metadata about the program embedded in the program itself. It can be parsed by the annotation parsing tool or by compiler. We can also specify annotation availability to either compile time only or till runtime also. Java Built-in annotations are @Override, @Deprecated and @SuppressWarnings.
* Null is not a valid object instance, so there is no memory allocated for it. It is simply a value that indicates that the object reference is not currently referring to an object.
* Each Java application uses an independent JVM.

Each JVM is a separate process, and that means there is no sharing of stacks, heaps, etcetera. (Generally, the only things that might be shared are the read only segments that hold the code of the core JVM and native libraries ... in the same way that normal processes might share code segments.

* The JVMs are independent processes. They don't share any mutable state. Garbage collection operates on each JVM independently
* There are three types of built-in Class Loaders in Java:
  + Bootstrap Class Loader – It loads JDK internal classes, typically loads rt.jar and other core classes.
  + Extensions Class Loader – It loads classes from the JDK extensions directory, usually $JAVA\_HOME/lib/ext directory.
  + System Class Loader – It loads classes from the current classpath that can be set while invoking a program using -cp or -classpath command line options
* Major difference between Heap and Stack memory are as follows:
  + Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
  + Whenever an object is created, it’s always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
  + Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally.
* Inheritance is an *"is-a"* relationship. Composition is a *"has-a"*. You do composition by having an instance of another class C as a field of your class, instead of extending C.
* IS-A relationship is when a class uses extends and implements, but HAS-A relationship is when the class has some code of other class.

**class** Dog **extends** Animal{

**private** Cannel cannel;

}

Here the Dog “IS-A” animal and “HAS-A” Cannel;

**Variable Scoping:**

* A variable is **shadowed** if there is another variable with the same name that is closer in scope
* Note: Instance variables are not overridden, they are hidden.

**class** Program {

String name = "Instance Var.";

**void** someMethod() {

String name = "Local Var.";

System.***out***.println(name); // Local Var.

System.***out***.println(**this**.name); // Instance Var.

}

}

* A class can declare a variable with the same name as an inherited variable from its parent class, thus "**hiding**" or **shadowing** the inherited version. (This is like overriding, but for variables.

**class** Program {

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

Sub s = **new** Sub();

Base b = s;

System.***out***.println(s.name); // Output "Sub"

System.***out***.println(b.name); // Output "Base"

System.out.println((Base)this).name).

//same output as privious line.

}

}

* When shadowing does happen, you can access the super class name by either the syntax super.*name* or by casting the object to its super class, with the syntax ((*Superclass*)*object*).*name*.

**DownCasting and UpCasting:**

* the reference type (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.

i.e if there is a Animal animal = **new** Dog();

A overriding method will check the right side(so if the right is dog(), the dog version will be called) while the overloading method checks the left side( if the left is animal no matter at runtime the Dog object is passed it will always call the method that takes animal).

**public** **class** Animal {

**public** **void** eat() {

System.out.println("Generic Animal Eating Generically");

}

}

**public** **class** Horse **extends** Animal {

**public** **void** eat() {

System.***out***.println("Horse eating hay ");

}

**public** **void** eat(String s) {

System.***out***.println("Horse eating " + s);

}

}

Animal a = new Animal(); a.eat(); //animal version of eat();

Horse h = new Horse(); h.eat(); //horse version of eat(); because (hourse both side)

Animal ah = new Horse(); ah.eat(); //horse version because overriding checks right side.

Horse he = new Horse(); he.eat("Apples"); //horse version overloaded.

Animal a2 = new Animal(); a2.eat("treats"); //compiler error, no method eat in animal with string

Animal ah2 = new Horse(); ah2.eat("Carrots"); //Compiler error! Compiler still looks only at the reference, and sees that Animal doesn’t have an eat() method that takes a String. Compiler doesn’t care that the actual object might be a Horse at runtime

* The compiler and JVM know it too, so the implicit upcast is always legal for assigning an object of a subtype to a reference of one of its supertype classes (or interfaces).

Example: Dog d = new Dog(); Animal a1 = d; // upcast ok with no explicit cast

Animal a2 = (Animal) d; // upcast ok with an explicit cast

* If Dog implements Pet, and Pet defines beFriendly(), then a Dog can be implicitly cast to a Pet, but the only Dog method you can invoke then is beFriendly(), which Dog was forced to implement because Dog implements the Pet interface
* If Beagle is a Dog and Dog implements Pet then, Beagle is a pet, and if Pet has a method beFriendly() that is not implemented by Beagle, can be possible if the Dog class has already given a concrete implementation of the beFriendly method( note that Dog implements Pet and inherits and hence Beagle has to implicitly implement Pet interface).
* If we do: Dog d=**new** Animal();//Compilation error
* Dog d=(Dog)**new** Animal();  //if we do a casting.

//Compiles successfully but ClassCastException is thrown at runtime

**class** Animal { }

**class** Dog **extends** Animal {

**static** **void** method(Animal a) {

**if**(a **instanceof** Dog){

Dog d=(Dog)a;//downcasting

System.***out***.println("ok downcasting performed");

}

}

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

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

Dog.method(a);

}

Return Type:

A method can return a subclass if the return type is defined either a superclass or a interface. Example:

**public** **interface** Chewable {}

**public** **class** Gum **implements** Chewable {}

**public** **class** TestChewable { // Method with an interface return type

**public** Chewable getChewable() {

**return** **new** Gum(); // Return interface implementer }

}

}

**Reflection**:

The name reflection is used to describe code which is able to inspect other code in the same system (or itself).

For example, say you have an object of an unknown type in Java, and you would like to call a 'doSomething' method on it if one exists. Java's static typing system isn't really designed to support this unless the object conforms to a known interface, but using reflection, your code can look at the object and find out if it has a method called 'doSomething' and then call it if you want to.

So, to give you a code example of this in Java (imagine the object in question is foo) :

Method method = foo.getClass().getMethod("doSomething", null);

method.invoke(foo, null);

//The null indicates there are no parameters being passed to the foo method

One very common use case in Java is the usage with annotations. JUnit 4, for example, will use reflection to look through your classes for methods tagged with the @Test annotation, and will then call them when running the unit test.

All objects in Java have the method getClass(), which lets you determine the object's class even if you don't know it at compile time (e.g. if you declared it as an Object

Reflection is important since it lets you write programs that do not have to "know" everything at compile time, making them more dynamic, since they can be tied together at runtime. The code can be written against known interfaces, but the actual classes to be used can be instantiated using reflection from configuration files.

Example : properties file , Spring, web.xml

**Cloneable Interface:**

* + It is used to indicate that a class allows a bitwise copy of an object (that is, a clone) to be made. If you try to call clone( ) on a class that does not implement **Cloneable**, a CloneNotSupportedException is thrown. When a clone is made, the constructor for the object being cloned is not called.
  + clone repeatedly up the chain until you have cloned an object, you have a shallow copy of the object. The clone generally shares state with the object being cloned. If that state is mutable, you don't have two independent objects. If you modify one, the other changes as well. And all of a sudden, you get random behavior.
  + Don’t use clone anymore except to copy arrays. You should use clone to copy arrays, because that's generally the fastest way to do it.
* The cloneable would be in the class where we are overriding the clone method.
* Cloning is an unsecured operation. That why whenever Java run time sees cloning, it expects one marker from the developer. We are providing that marker in the form of Cloneable interface.
* While cloning, copy of the object is created by **field-by-field** assignment

@Override

public Object clone(){

try {

return super.clone(); //Cast to the object type we want to copy

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return this;

}

The default version of clone() method creates the shallow copy of an object.

**Copy Objects**:

In a shallow copy, object B points to object A's location in memory. In deep copy, all things in object A's memory location get copied to object B's memory location.

Shallow Copy example:

We have a class A. having a variable i. we have a parameterized constructor, now we have a class B which has a varable A a. it has a contructor too. B implements cloneable interface and overrides the clone method as above. Now we call the main method and clone the value of B. and get object B c= (B)b.clone();

Now if we change A’s variables like B.a.i=something; then the new Object c will also be having change as the Object A was not cloned.

Hence for deep copy we should have a default clone() in Class A and then we should have a clone method in B which clones all the classes which B needs.

@Override

    protected Object clone() throws CloneNotSupportedException

    {

        B b = (B) super.clone();

        b.a = (A) a.clone();

        return b;

    }

Here if we change the A’s object it won’t reflect in the copied Object.

**Marker interfaces**

Marker Interfaces in java are interfaces with no members declared in them. They are just an empty interfaces used to mark or identify a special operation. For example, Cloneable interface is used to mark cloning operation and Serializable interface is used to mark serialization and deserialization of an object.

**Inner interface:**

Inner interface is also called nested interface, which means declare an interface inside of another interface. Map.Entry is an nested Interface .

**public** **interface** Map {

**interface** Entry{

**int** getKey();

}

**void** clear();

}

**public** **class** MapImpl **implements** Map {

**class** ImplEntry **implements** Map.Entry{

**public** **int** getKey() {

**return** 0;

}

}

@Override

**public** **void** clear() {

*//clear*

}

}

**Native**:

The native keyword is applied to a method to indicate that the method is implemented in native code using JNI(Java Native Interface).

Native keyword only works on methods, Not classes or variables. It states that the method is implemented in platform independent code.

* Write Java code
* Compile Java code
* Create C header (.h file)
* Create C stubs file
* Write C code
* Create shared code library (or DLL)
* Run application

We want to run a method named printText();

**public** **native** **void** printText ();

For this we create a Java Class Named Happy.java and add the method inside the class with native keyword. Then we will have the static block to load the library. NOTE: the library should be smaller case of the Class name.

Then we will have the main method to instantiate a class and call the method.

**class** **Happy**

{

**public** **native** **void** printText ();

**static**

{

**System**.loadLibrary ("happy"); */\* Note lowercase of classname! \*/*

}

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

{

**Happy** happy = **new** **Happy** ();

happy.printText ();

}

}

Now we will compile the java class Happy:

% javac **Happy**.java

Then we will convert it to a C header file. This will give Happy.h

% javah **Happy**

Compiling the code to a C stub:

% javah -stubs **Happy**

Now write the actual java code:

*#include* &ltStubPreamble.h> */\* Standard native method stuff. \*/*

*#include* "Happy.h" */\* Generated earlier. \*/*

*#include* &ltstdio.h> */\* Standard C IO stuff. \*/*

**void** **Happy\_printText** (**struct** **HHappy** \***this**)

{

puts ("Happy New Year!!!");

}

Now create a shared library and this depends on the underlying OS.

After just run the class.

Use of Native Code:

* write faster code on a critical section with better CPU assembly instructions (not CPU portable)
* make direct system calls (not OS portable).

This Method calls the c code and returns the value back. Examples are System.arraycopy, isAlive(), clone() etc.

**JNI:**

Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and be called by native applications.

Drawbacks:

* An application that relies on JNI loses the platform portability Java offers
* The JNI framework does not provide any automatic garbage collection for non-JVM memory resources allocated by code executing on the native side

**Class**:

* Default access: a class with default access can be seen only by classes within the same package
* Public access: all classes in the Java Universe (JU) have access to a public class. The class can now be instantiated from all other classes, and any class is now free to subclass (extend from) it—unless, that is, the class is also marked with the nonaccess modifier final.
* Marking a class as strictfp means that any method code in the class will conform to the IEEE 754 standard rules for floating points. Without that modifier,floating points used in the methods might behave in a platform-dependent way.
* Final ClassesWhen used in a class declaration, the final keyword means the class can't be subclassed. In other words, no other class can ever extend (inherit from) a final class, and any attempts to do so will give you a compiler error. Can't subclass final classes
* Abstract ClassesAn abstract class can never be instantiated. Its sole purpose, mission in life, raison d'être, is to be extended (subclassed).

i.e. can’t do Car car = new Car();

AnotherClass.java:7: class Car is an abstract

class. It can't be instantiated.

You can, however,put nonabstract methods in an abstract class.

* A superclass can’t be instantiated because there can’t be any constructor in it.
* But we can instatntiate it using it as inner class in subclass.

abstract class My {

public void myMethod() {

System.out.print("Abstract");

}

}

class Poly extends My {

public static void main(String a[]) {

My m = new My() {};

m.myMethod();

}

}

* how you could fix a code sample that includes a method ending in a semicolon,but without an abstract modifier on the class or method. In that case, you could either mark the method and class abstract, or change the semicolon to code (like a curly brace pair).
* You can't mark a class as both abstract and final. They have nearly opposite meanings. An abstract class must be subclassed, whereas a final class must not be subclassed. If you see this combination of abstract and final modifiers, used for a class or method declaration, the code will not compile.
* the compiler looks only at the reference type, not the instance type.

**public** **class** TestAnimals {

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

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

Animal b = **new** Horse(); // Animal ref, but a Horse object

a.eat(); // Runs the Animal version of eat()

b.eat(); // Runs the Horse version of eat()

}

}

**class** Animal {

**public** **void** eat() {

System.out.println("Generic Animal Eating Generically");

}

}

**class** Horse **extends** Animal {

**public** **void** eat() {

System.***out***.println("Horse eating hay, oats, " + "and horse treats");

}

**public** **void** buck() {

}

}

But we can’t do is:

Animal c = new Horse(); c.buck(); // Can't invoke buck();

**Interface**:

* All interface methods are implicitly public and abstract. In other words, you do not need to actually type the public or abstract modifiers in the method declaration, but the method is still always public and abstract.
* Because interface methods are abstract, they cannot be marked final.
* All variables defined in an interface must be public, static, and final—in other words, interfaces can declare only constants, not instance variables.
* interfaces are implicitly abstract whether you type abstract or not.declarations are legal, and functionally identical:

public abstract interface Rollable { }

public interface Rollable { }

* An interface can *extend* one or more other interfaces.
* An interface cannot extend anything but another interface.
* An interface cannot implement another interface or class.
* You can do Interface A = new B();

This is possible as if the class B implements the Interface A it has to get all the methods from The interface A, hence A is small and B is big.

But we can’t do B b = new A(); //as Interface are abstract you can’t instantiate them.

Same of a Abstract class If C is a abstract class it will throw exception. If we try to do below:

B b = new C();

If C is just a superclass then:

B b = new C();

* If we have a method in Interface with signature void doStuff(); we can’t implement the interface and have a method as void doStuff() in the subclass, as we see all the methods in interface are abstract and PUBLIC, but as soon as we implement it in class the void doStuff() in class has a DEFAULT access, hence the implementation has narrowed the access, and it will throw error.

Attempting to assign weaker privilege; was public

* For a subclass, if a member of its superclass is declared public, the subclass inherits that member regardless of whether both classes are in the same package.
* When a member is declared private, a subclass can't inherit it.
* You can, however, declare a matching method in the subclass. But regardless of how it looks, ***it is not an overriding method!*** It is simply a method that happens to have the same name as a private method (which you're not supposed to know about) in the superclass.
* The protected and default access control levels are almost identical, but with one critical difference. A *default* member may be accessed only if the class accessing the member belongs to the same package (default members are visible to subclasses only if those subclasses are in the same package as the superclass.) whereas a *protected* member can be accessed (through inheritance) by a subclass ***even if the subclass is in a different package.***
* **The subclass can see the** protected **member only through inheritance. NO DOT OPERATOR.**
* Once the subclass-outside-the-package inherits the protected member, that member (as inherited by the subclass) becomes private to any code outside the subclass,

i.e. if the SuperClass is A in package com.sumit.A and Subclass B is in package com.sumit.B

B can Access A’s member variable protected int x=0; only using inheritance that is B extends A. not like

A a = new A();

System.out.println(a.x); //This will not compile.

In Addition if a class C under same package as B, com.sumit.B , will not be able to access “x” as in the class B the inherited variable x acts as private. So if C does

B b = new B();

System.out.println(b.x); //will throw exception.

* any local variable declared with an access modifier will not compile.
* The final keyword prevents a method from being overridden in a subclass, and is often used to enforce the API functionality of a method. For example, the Thread class has a method called isAlive() that checks whether a thread is still active. If you extend the Thread class, though, there is really no way that you can correctly implement this method yourself (it uses native code, for one thing).
* a final argument in a method must keep the same value that the parameter had when it was passed into the method( can’t assign new value).
* An abstract method is a method that's been *declared* (as abstract) but not *implemented*. *it has no method body*
* **The first concrete subclass of an abstract class must implement *all* abstract methods of the superclass.**
* he abstract modifier can never be combined with the static modifier

**Contructor**:

* Constructors can't be marked static (they are after all associated with object instantiation), they can't be marked final or abstract (because they can't be overridden).
* 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,. The rest of the bits represent the value, using two's complement notation
* double holds 64 bits and a float 32.
* local variable must be initialized before you try to use it. The compiler will reject any code that tries to use a local variable that hasn't been assigned a value, because—unlike instance variables—local variables don't get default values.
* this.size = size; // this.size means the current object's

// instance variable, size. The size

// on the right is the parameter

* If you mark an instance variable as transient, you're telling the JVM to skip (ignore) this variable when you attempt to serialize the object containing it.
* Instance variables and objects live on the heap.
* Local variables live on the stack.
* 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, the compiler will complain about a possible loss of precision, because you're trying to fit a number into a (potentially) less precise

"container."

* The way in which object references are stored is virtual-machine specific We know that a literal integer is always an int, but more importantly, the result of an expression involving anything int-sized or smaller is always an int. In other words, add two bytes together and you'll get an int—even if those two bytes are tiny. Multiply an int and a short and you'll get an int. Divide a short by a byte and you'll get…an int.

byte a = 3; // No problem, 3 fits in a byte

byte b = 8; // No problem, 8 fits in a byte

byte c = b + c; // Should be no problem, sum of the two bytes

// fits in a byte

The last line won't compile! You'll get an error something like this:

TestBytes.java:5: possible loss of precision

found : int

required: byte

byte c = a + b;

^

We tried to assign the sum of two bytes to a byte variable, the result of which

(11) was definitely small enough to fit into a byte, but the compiler didn't care. It

knew the rule about int-or-smaller expressions always resulting in an int. It would

have compiled if we'd done the *explicit* cast:

byte c = (byte) (a + b);

* +=, -=, \*=, and /= will all put in an implicit cast. Hence no need to explicitly define cast.
* byte a = 128;

Let's take a look at what happens in the preceding code. There, 128 is the bit pattern 10000000. It takes a full 8 bits to represent 128. But because the literal 128 is an int, we actually get 32 bits, with the 128 living in the right-most (lower-order) 8 bits. So a literal 128 is actually

00000000000000000000000010000000

Take our word for it; there are 32 bits there. To narrow the 32 bits representing 128, Java simply lops off the leftmost (higherorder) 24 bits. We're left with just the 10000000. But remember that a byte is signed, with the leftmost bit representing the sign (and not part of the value of the variable). So we end up with a negative number (the 1 that used to represent 128 now represents the negative sign bit). Remember, to find out the value of a negative number using two's complement notation, you flip all of the bits and then add 1. Flipping the 8 bits gives us 01111111, and adding 1 to that gives us 10000000, or back to 128! And when we apply the sign bit, we end up with –128.

* public class Foo {

public void doFooStuff() { }

}

public class Bar extends Foo {

public void doBarStuff() { }

}

class Test {

public static void main (String [] args) {

Foo reallyABar = new Bar(); // Legal because Bar is a

// subclass of Foo

Bar reallyAFoo = new Foo(); // Illegal! Foo is

}

}

* Remember, a Bar object is guaranteed to be able to do anything a Foo can do, so anyone with a Foo reference can invoke Foo methods even though the object is actually a Bar.
  + Attempting to access an instance variable from a static context (typically from main() ).

class ScopeErrors {

int x = 5;

public static void main(String[] args) {

x++; // won't compile, x is an 'instance' variable

}

}

* Attempting to access a local variable from a nested method.

class ScopeErrors {

public static void main(String [] args) {

ScopeErrors s = new ScopeErrors();

s.go();

}

void go() {

int y = 5;

go2();

y++; // once go2() completes, y is back in scope

}

void go2() {

y++; // won't compile, y is local to go()

}

}

* Attempting to use a block variable after the code block has completed.
* the integer defined as a class member and not under main() or method will automatically be assigned a default value whether we init it or not.
* *Array elements are always, always, always given default values,regardless of where the array itself is declared or instantiated. i.e an int array will have all the elements as 0 if we don’t init the elements.*

If we initialize an array, object reference elements will equal null if they are not initialized individually with values. If primitives are contained in an array, they will be given their respective default values.

* If a variable is in main() it should be initialized or else it will give compilation error, unless you never use it.
* Explicitly assigning a reference variable will compile fine. But if you compile a code that has unassigned variable it will throw error.

Date date = null; // Explicitly set the local reference

// variable to null

This will compile fine but

Date date;

If(Date ==null{ //Do something} will cause error.

* 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-of the- variable!
* And if you're passing an object reference variable, you're passing a copy of the bits representing the reference to an object. The called method then gets its own copy of the reference variable, to do with it what it likes. But because two identical reference variables refer to the exact same object, if the called method modifies the object (by invoking setter methods, for example), the caller will see that the object the caller's original variable refers to has also been changed.
* called method can't reassign the caller's original reference variable and make it refer to a different object.

void bar() {

Foo f = new Foo();

doStuff(f);

}

void doStuff(Foo g) {

g.setName("Boo");

g = new Foo();

}

* reassigning g does not reassign f! At the end of the bar() method, two Foo objects have been created, one referenced by the local variable f and one referenced by the local (argument) variable g. Because the doStuff() method has a copy of the reference variable, it has a way to get to the original Foo object, for instance to call

the setName() method. But, the doStuff() method does *not* have a way to get to the f reference variable. So doStuff() can change values within the object f refers to, but doStuff() can't change the actual contents (bit pattern) of f. In other words, doStuff() can change the state of the object that f refers to, but it can't make f refer to a different object!

* int a = 1;

ReferenceTest rt = new ReferenceTest();

System.out.println("Before modify() a = " + a);

rt.modify(a);

System.out.println("After modify() a = " + a);

}

void modify(int number) {

number = number + 1;

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

}

}

In this simple program, the variable a is passed to a method called modify(), which increments the variable by 1. The resulting output looks like this:

Before modify() a = 1

number = 2

After modify() a = 1

Notice that a did not change after it was passed to the method. Remember, it was a copy of a that was passed to the method. When a primitive variable is passed to a method, it is passed by value, which means pass-by-copy-of-the-bits-in-the-variable.

* Int i[ ] is fine unless and until you initialize i. or new int[ ]; then you need to give the size.
* We can add a subclass if the Super class is the arraytype.

class Car {}

class Subaru extends Car {}

class Ferrari extends Car {}

...

Car [] myCars = {new Subaru(), new Car(), new Ferrari()};

* A variable of type byte, short, or char can be explicitly promoted and assigned to an int, an array of any of those types could be assigned to an int array.

int[] splats;

int[] dats = new int[4];

char[] letters = new char[5];

splats = dats; // OK, dats refers to an int array

splats = letters; // NOT OK, letters refers to a char array

* Honda IS-A Car, so a Honda array can be assigned to a Car array. Beer IS-A Car is not true; Beer does not extend Car.
* ***You cannot reverse the legal assignments. A Car array cannot be assigned to a Honda array. A Car is not necessarily a Honda, so if you’ve declared a Honda array, it might blow up if you assigned a Car array to the Honda reference variable. Think about it: a Car array could hold a reference to a Ferrari, so someone who thinks they have an array of Hondas could suddenly fi nd themselves with a Ferrari. Remember that the IS-A test can be checked in code using the*** instanceof ***operator.***
* 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 *once*, when the class is first loaded. An *instance* initialization block runs once *every time a new instance is created*. The instance code suns right after call to super();
* The order in which the Instance block runs is according to top bottom approach.

class SmallInit {

static int x;

int y;

static { x = 7 ; } // static init block

{ y = 8; } // instance init block

}

class Init {

Init(int x) { System.out.println("1-arg const"); }

Init() { System.out.println("no-arg const"); }

static { System.out.println("1st static init"); }

{ System.out.println("1st instance init"); }

{ System.out.println("2nd instance init"); }

static { System.out.println("2nd static init"); }

public static void main(String [] args) {

new Init();

new Init(7);

}

}

Result:

1st static init

2nd static init

1st instance init

2nd instance init

no-arg const

1st instance init

2nd instance init

1-arg const

* Instance init blocks are often used as a place to put code that all the constructors in a class should share. That way, the code doesn't have to be duplicated across constructors. Finally, if you make a mistake in your static init block, the JVM can throw an ExceptionInInitializationError.
* *Wrapper objects are immutable*
* When == is used to compare a primitive to a wrapper, the wrapper will be unwrapped and the comparison will be primitive to primitive.

Method Overriding:

Java uses virtual method invocation to dynamically select the actual version of the method that will run.

**Method Overloading:**

* *In every case, when an exact match isn't found, the JVM uses the method with the smallest argument that is wider than the parameter.*

the compiler will choose widening over boxing,

Widening> Boxing>var-args

class Animal {static void eat() { } }

class Dog3 extends Animal {

public static void main(String[] args) {

Dog3 d = new Dog3();

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

}

void go(Animal a) { }

}

* No problem! The go() method needs an Animal, and Dog3 IS-A Animal. (Remember, the go() method thinks it's getting an Animal object, so it will only ask it to do Animal things, which of course anything that inherits from Animal can do.) So, in this case, the compiler widens the Dog3 reference to an Animal, and the invocation succeeds. The key point here is that reference widening depends on inheritance, in other words the IS-A test. Because of this, it's not legal to widen

from one wrapper class to another, because the wrapper classes are peers to one another. For instance, it's NOT valid to say that Short IS-A Integer.

***the following will NOT compile:***

class Dog4 {

public static void main(String [] args) {

Dog4 d = new Dog4();

d.test(new Integer(5)); // can't widen an Integer

// to a Long

}

void test(Long x) { }

}

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

class WidenAndBox {

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

public static void main(String [] args) {

byte b = 5;

go(b); // must widen then box - illegal

}

}

This is just too much for the compiler:

WidenAndBox.java:6: go(java.lang.Long) in WidenAndBox cannot be

applied to (byte)

if compiler tried to box first, the byte would have been converted to a Byte. Now we're back to trying to widen a Byte to a Long, and of course, the IS-A test 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.)
* member fields can not be overridden like methods. When a subclass defines a field with the same name, the subclass just declares a new field. The field in the superclass is *hidden*

Garbage Collection:

* 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.
* the garbage collector uses a mark and sweep algorithm,
* garbage collection cannot be forced. However, Java provides some methods that allow you to request that the JVM perform garbage collection. System.gc() or, Runtime.getRuntime.gc();
* Runtime.gc() is a native method whereas System.gc() is non - native method which in turn calls the Runtime.gc()
* The call System.gc() is effectively equivalent to the call:

Runtime.getRuntime().gc()

* Memory leak:
* The Runtime class is a special class that has a single object (a Singleton) 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.
* For any given object, finalize() will be called only once (at most) by the garbage collector. Also, the garbage collector is not guaranteed to run at any specific time
* The **java.lang.Object.finalize()** is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

Car car= new Car();

car.finalize();

* if overridding finalize() it is good programming practice to use a try-catch-finally statement and to always call super.finalize(). This is a safety measure to ensure you do not inadvertently miss closing a resource used by the objects calling class

protected void finalize() throws Throwable {

try {

close(); // close open files

} finally {

super.finalize();

}

}

* To make Java more memory efficient, the JVM sets aside a special area of memory called the "String constant pool." When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created. (The existing String simply has an additional reference.).
* A common use for StringBuffers and StringBuilders is file I/O when large, ever-changing streams of input are being handled by the program.
* Sun recommends that you use StringBuilder instead of StringBuffer whenever possible because StringBuilder will run faster.
* Find the right bucket (using hashCode()) 2. Search the bucket for the right element (using equals() )
* A hashCode() that returns the same value for all instances whether they’re equal or not is still a legal—even appropriate—hashCode() method! For example, public int hashCode() { return 1492; } This does not violate the contract. Two objects with an x value of 8 will have the same hashcode. But then again, so will two unequal objects, one with an x value of 12 and the other a value of -920. This hashCode() method is horribly ineffi cient, remember, because it makes all objects land in the same bucket, but even so, the object can still be found as the collection cranks through the one and only bucket—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.
* 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.
* Find the right bucket (using hashCode()) 2. Search the bucket for the right element (using equals() )
* A hashCode() that returns the same value for all instances whether they’re equal or not is still a legal—even appropriate—hashCode() method! For example, public int hashCode() { return 1492; } This does not violate the contract. Two objects with an x value of 8 will have the same hashcode. But then again, so will two unequal objects, one with an x value of 12 and the other a value of -920. This hashCode() method is horribly ineffi cient, remember, because it makes all objects land in the same bucket, but even so, the object can still be found as the collection cranks through the one and only bucket—using equals()—
* Keep variables non-transient or, if they must be marked transient, don't use them to determine hashcodes or equality.

**Collections:**

* Collections (capital C and ends with s) is the java.util.Collections class
* Collection (capital C), which is actually the java.util.Collection interface from which Set, List, and Queue extend.
* Vector is basically the same as an ArrayList, but Vector methods are synchronized for thread safety.
* A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another.
* A HashSet is an unsorted, unordered Set. It uses the hashcode of the object being inserted, so the more efficient your hashCode() implementation the better access performance you'll get.
* A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements.
* The TreeSet is one of two sorted collections (the other being TreeMap). It uses a Red-Black tree structure (but you knew that), and guarantees that the elements will be in ascending order, according to natural order
* The HashMap gives you an unsorted, unordered Map.
* HashMap allows one null key and multiple null values in a collection.
* LinkedHashMap collection maintains insertion order.
* You can probably guess by now that a TreeMap is a sorted Map. TreeMap implements NavigableMap.
* 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.
* collections can hold Objects but not primitives.

If you do Collection cs = new ArrayList<String>();

Or, Collection cs = new HashSet<String>();

But it won’t compile on Collection cs =new HashMap<String, Integer>(); //as Hashmap is not a part of Collection.

If you do Collection cs = new Set();

Or, Collection cs = new List();

Or,

Collections cs = new List();

Both wont compile as both are abstract classes.

If you want to compile Collections ls = new ArrayList();

It will give you incompatible types error. incompatible types: ArrayList cannot be converted to Collections

If you do List ls = new Collections();

It will not compile as Collections() constructor is defined as private in Collections.

If you do List ls = new Collection();

It won’t compile as Collection is abstract.

* 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().

class DVDInfo implements Comparable { // #1

// existing code

public int compareTo(DVDInfo d) {

return title.compareTo(d.getTitle()); // #2 } }

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

* 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.

import java.util.\*;

class GenreSort implements Comparator {

public int compare(DVDInfo one, DVDInfo two) {

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

}

GenreSort gs = new GenreSort();

Collections.sort(dvdlist, gs); //Where dvdList is a Array or list. Any thing

* **Binary Search:**
  + The collection/array being searched must be sorted before you can search it.
  + If the collection/array you want to search was sorted using a Comparator, it must be searched using the same Comparator, which is passed as the second argument to the binarySearch() method. Remember that Comparators cannot be used when searching arrays of primitives.
  + Iterator:
  + boolean hasNext() Returns true if there is at least one more element in the collection being traversed. Invoking hasNext() does NOT move you to the next element of the collection.
  + Object next() This method returns the next object in the collection, AND moves you forward to the element after the element just returned.
  + whenever you want a collection to be sorted, its elements must be mutually comparable.i.e->

Set s = new TreeSet();

s.add(“a”);

s.add(new Integer(“b”);

s.add(“true”);

//this will give compiler error as they are not mutually comarable. But if we use Set s = new HashSet() this would compile as HashSet doesnt care about the sorting.

**Synchronized vs Concurrent Collection:**

* synchronized collections locks the whole collection
* Concurrent collection uses lock stripping. For example, the ConcurrentHashMap divides the whole map into several segments and locks only the relevant segments, which allows multiple threads to access other segments of same ConcurrentHashMap without locking. But if the Map grows longer its costly to lock the whole MAP. Instead it doesn’t matter to Concurrenthashmap as it uses a part to lock.
* CopyOnWriteArrayList allows multiple reader threads to read without synchronization and when a write happens it copies the whole ArrayList and swap with a newer one.
* Synchronized ArrayList is a synchronized collection while CopyOnWriteArrayList is a concurrent collection.  
  CopyOnWriteArray is more scalable than Sync ArrayList.
* The Iterator returned from synchronized ArrayList is a fail fast but iterator returned by CopyOnWriteArrayList is a fail-safe iterator i.e. it will not throw ConcurrentModifcationException even when the list is modified
* NOTE: the size of ArrayList if its big then obviously cost of copying after a write operation is high enough to compensate the cost of locking but if ArrayList is really tiny then you can still use CopyOnWriteArrayList

[**Difference between <? super T> and <? extends T>**](http://stackoverflow.com/questions/4343202/difference-between-super-t-and-extends-t-in-java)

### extends

The wildcard declaration of List<? extends Number> foo3 means that any of these are legal assignments:

List<? extends Number> foo3 = new ArrayList<Number>(); // Number "extends" Number (in this context)

List<? extends Number> foo3 = new ArrayList<Integer>(); // Integer extends Number

List<? extends Number> foo3 = new ArrayList<Double>(); // Double extends Number

1. **Reading** - Given the above possible assignments, what type of object are you guarenteed to read from List foo3:
   * You can read a **Number** because any of the lists that could be assigned to foo3 contain a Number or a subclass of Number.
   * You can't read an Integer because foo3 could be pointing at a List<Double>.
   * You can't read a Double because foo3 could be pointing at a List<Integer>.
2. **Writing** - Given the above possible assignments, what type of object could you add to List foo3 that would be legal for **all** the above possible ArrayList assignments:
   * You can't add an Integer because foo3 could be pointing at a List<Double>.
   * You can't add a Double because foo3 could be pointing at a List<Integer>.
   * You can't add a Number because foo3 could be pointing at a List<Integer>.

You can't add any object to*List<? extends T>*because you can't guarantee what kind of*List*it is really pointing to, so you can't guarantee that the object is allowed in that*List*. The only "guarantee" is that you can only read from it and you'll get a*T*or subclass of*T*.

### super

Now consider List <? super T >.

The wildcard declaration of List<? super Integer> foo3 means that any of these are legal assignments:

List<? super Integer> foo3 = new ArrayList<Integer>(); // Integer is a "superclass" of Integer (in this context)

List<? super Integer> foo3 = new ArrayList<Number>(); // Number is a superclass of Integer

List<? super Integer> foo3 = new ArrayList<Object>(); // Object is a superclass of Integer

1. **Reading** - Given the above possible assignments, what type of object are you guaranteed to receive when you read from List foo3:
   * You aren't guaranteed an Integer because foo3 could be pointing at a List<Number> or List<Object>.
   * You aren't guaranteed an Number because foo3 could be pointing at a List<Object>.
   * The **only** guarantee is that you will get an instance of an **Object** or subclass of Object(but you don't know what subclass).
2. **Writing** - Given the above possible assignments, what type of object could you add to List foo3 that would be legal for **all** the above possible ArrayList assignments:
   * You can add an Integer because an Integer is allowed in any of above lists.
   * You can add an instance of a subclass of Integer because an instance of a subclass of Integer is allowed in any of the above lists.
   * You can't add a Double because foo3 could be pointing at a ArrayList<Integer>.
   * You can't add a Number because foo3 could be pointing at a ArrayList<Integer>.
   * You can't add a Object because foo3 could be pointing at a ArrayList<Integer>.

**Threads**:

* one call stack per thread.
* The JVM, which gets its turn at the CPU by whatever scheduling mechanism the underlying OS uses, operates like a mini-OS and schedules its own threads regardless of the underlying operating system.
* types of threads (user and daemon) is that the JVM exits an application only when all user threads are complete—the JVM doesn't care about letting daemon threads complete, so once all user threads are complete, the JVM will shut down, regardless of the state of any daemon threads.
* The run() method will call other methods, of course, but the thread of execution—the new call stack—always begins by invoking run().
* Implementing using thread class: We can use overloding of run method.

class MyThread extends Thread {

public void run() {

System.out.println("Important job running in MyThread");

}

}

But the overloaded method let say run(String s) will not be called implicitly and if called explicitly it wont haveits own stack. It will continue under the main() stack only.

* Using Runnable Interface:

class MyRunnable implements Runnable {

public void run() {

System.out.println("Important job running in MyRunnable");

}

}

* Instantiating a Runnable interface will be:

MyRunnable r = new MyRunnable();

Thread t = new Thread(r); //Passing r into Thread instance.

If you create a thread using the no-arg constructor, the thread will call its own run() method when it's time to start working. That's exactly what you want when you extend Thread, but when you use Runnable, you need to tell the new thread to use your run()method rather than its own.

You can pass a single Runnable instance to multiple Thread objects.

MyRunnable r = new MyRunnable();

Thread foo = new Thread(r);

Thread bar = new Thread(r);

Thread bat = new Thread(r); // Giving the same target to multiple threads means that several threads of execution will be running the very same job.

* 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. The isAlive() method is the best way to determine if a thread has been started but has not yet completed its run() method.
* After we give t.start();

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. i.e **Running** state.

* Once a thread has been started, it can never be started again. It will give compiler error. Only a new thread can be started, and then only once. A runnable thread or a dead thread cannot be restarted.
* The getld() method returns a positive, unique, long number, and that number will be that thread's only ID number for the thread's entire life.
* A thread can be only in one of five states
  + 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. Thread is NOT ALIVE.
  + 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. BUT is ALIVE.
  + Running This is the state a thread is in when the thread scheduler selects it.
  + Waiting/blocked/sleeping This is the state a thread is in when it's not eligible to run. Okay, so this is really three states combined into one, but they all have one thing in common: the thread is still alive, but is currently not eligible to run. In other words, it is not runnable,
  + 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.
  + The sleep() method is a static method of class Thread. You use it in your code to "slow a thread down" by forcing it to go into a sleep mode before coming back to runnable (where it still has to beg to be the currently running thread). When a thread sleeps, it drifts off somewhere and doesn't return to runnable until it wakes up. you wrap calls to sleep() in a try/catch, as in the preceding code. ONLY THE CURRENT RUNNING THREAD GOES TO SLEEP.
  + **NOTE**: Just because a thread’s sleep() expires, and it wakes up, does not mean it will return to running! Remember, when a thread wakes up, it simply goes back to the runnable state. So the time specifi ed in sleep() is the minimum duration in which the thread won’t run, but it is not the exact duration in which the thread won’t run.
* 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 most cases, the running thread will be of equal or greater priority than the highest priority threads in the pool.

FooRunnable r = new FooRunnable();

Thread t = new Thread(r);

t.setPriority(8);

t.start();

Thread.MIN\_PRIORITY (1) Thread.NORM\_PRIORITY (5) Thread.MAX\_PRIORITY (10) these 3 define the priority.

* What **yield**() is supposed to do is 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.

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,

* 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.
* Thread t = new Thread(); t.start(); t.join();

The preceding code takes the currently running thread (if this were in the main() method, then that would be the main thread) and joins it to the end of the thread referenced by t. This blocks the current thread from becoming runnable until after the thread referenced by t is no longer alive. In other words, the code t.join() means "Join me (the current thread) to the end of t, so that t must finish before I (the current thread) can run again."

**Synchronization**:

* You can't guarantee that a single thread will stay running throughout the entire atomic operation. But you can guarantee that even if the thread running the atomic operation moves in and out of the running state, no other running thread will be able to act on the same data.
* 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.
* 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.
* Only methods (or blocks) can be synchronized, not variables or classes.
* If a thread goes to sleep, it holds any locks it has—it doesn't release them.
* 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.
* You can synchronize a block of code rather than a method.

Eample:

synchronized(this) {

System.out.println("synchronized");

}

When you synchronize a method, the object used to invoke the method is the object whose lock must be acquired.

But when you synchronize a block of code, you specify which object's lock you want to use as the lock, ie sysnocronized(this)

* static methods can be synchronized. There is only one copy of the static data you're trying to protect, so you only need one lock per class to synchronize static methods—a lock for the whole class. There is such a lock; every class loaded in Java has a corresponding instance of java.lang.Class representing that class. It's that java.lang.Class instance whose lock is used to protect the static methods of the class (if they're synchronized).
* Example: syncronized(MyClass.class){ }
* Instead of initialising a class we can write:

Class cl = Class.forName("MyClass");

Or, (MyClass.class)

* Threads calling non-static synchronized methods in the same class will only block each other if they're invoked using the same instance. That's because they each lock on this instance, and if they're called using two different instances, they get two locks, which do not interfere with each other.
* Threads calling static synchronized methods in the same class will always block each other—they all lock on the same Class instance.
* A static synchronized method and a non-static synchronized method will not block each other, ever. The static method locks on a Class instance while the non-static method locks on the this instance—these actions do not interfere with each other at all.
* For synchronized blocks, you have to look at exactly what object has been used for locking. (What's inside the parentheses after the word synchronized?) Threads that synchronize on the same object will block each other. Threads that synchronize on different objects will not.
* 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—
* *Access to static fields should be done from static synchronized methods. Access to non-static fields should be done from non-static synchronized methods.*

**DeadLock**:

* Deadlock occurs when two threads are blocked, with each waiting for the other's lock. Neither can run until the other gives up its lock, so they'll sit there forever.

Create two Objects of class. In one case put 1st inside second using synchronized and 2nd time put second into first.

public int read() {

synchronized(resourceA) { // May deadlock here

synchronized(resourceB) {

return resourceB.value + resourceA.value;

}

public void write(int a, int b) {

synchronized(resourceB) { // May deadlock here

synchronized(resourceA) {

resourceA.value = a;

resourceB.value = b;

}

* Code like this almost never results in deadlock because the CPU has to switch from the reader thread to the writer thread at a particular point in the code, and the chances of deadlock occurring are very small. The application may work fine 99.9 percent of the time.
* *wait(), notify(), and notifyAll()* must be called from within a synchronized context! A thread can't invoke a wait or notify method on an object unless it owns that object's lock.

*These methods comes from Object Class.*

1. class ThreadA {

2. public static void main(String [] args) {

3. ThreadB b = new ThreadB();

4. b.start();

5.

6.

synchronized(b) {

7. try {

8. System.out.println("Waiting for b to complete...");

9. b.wait();

10. } catch (InterruptedException e) {}

11. System.out.println("Total is: " + b.total);

12. }

13. }

14. }

15.

16. class ThreadB extends Thread {

17. int total;

18.

19. public void run() {

20. synchronized(this) {

21. for(int i=0;i

* This program contains two objects with threads: ThreadA contains the main thread and ThreadB has a thread that calculates the sum of all numbers from 0 through 99. As soon as line 4 calls the start() method, ThreadA will continue with the next line of code in its own class, which means it could get to line 11 before ThreadB has finished the calculation. To prevent this, we use the wait() method in line 9. Notice in line 6 the code synchronizes itself with the object b—this is because in order to call wait() on the object, ThreadA must own a lock on b. For a thread to call wait() or notify(), the thread has to be the owner of the lock for that object. When the thread waits, it temporarily releases the lock for other threads to use, but it will need it again to continue execution.
* synchronized(this) { notify(); }

This code notifies a single thread currently waiting on the this object. The lock can be acquired much earlier in the code, such as in the calling method. Note that if the thread calling wait() does not own the lock, it will throw an Exception.

* When the wait() method is invoked on an object, the thread executing that code gives up its lock on the object immediately. However, when notify() is called, that doesn’t mean the thread gives up its lock at that moment. If the thread is still completing synchronized code, the lock is not released until the thread moves out of synchronized code. So just because notify() is called doesn’t mean the lock becomes available at that moment.
* an object can have many threads waiting on it, and using notify() will affect only one of them. Which one, exactly, is not specified and depends on the JVM implementation, so you should never rely on a particular thread being notified in preference to another. Thats why always use NOTIFYALL();
* **The methods wait() , notify(), and notifyAll() are methods of only java.lang.Object, not of java.lang.Thread or java.lang.Runnable. Be sure you know which methods are defi ned in Thread, which in Object, and which in Runnable (just run(), so that’s an easy one). Of the key methods in Thread, be sure you know which are static— sleep() and yield(), and which are not static—join() and start().**
* If you declare the method as synchonized (as you're doing by typing public synchronized void addA()) you synchronize on the **whole** object, so two thread accessing a different variable from this same object would block each other anyway.

If you want to synchronize only on one variable at a time, so two threads won't block each other while accessing different variables, you have synchronize on them separately in synchronized () blocks. If a and b were object references you would use:

public void addA() {

synchronized( a ) {

a++;

}

}

public void addB() {

synchronized( b ) {

b++;

}

}

But since they're primitives you can't do this.

I would suggest you to use AtomicInteger instead:

import java.util.concurrent.atomic.AtomicInteger;

class X {

AtomicInteger a;

AtomicInteger b;

public void addA(){

a.incrementAndGet();

}

public void addB(){

b.incrementAndGet();

}

}

* Synchronizing on the method is functionally equivalent to having a synchronized (this) block around the body of the method. The object "this" doesn't become locked, rather the object "this" is used as the mutex and the body is prevented from executing concurrently with other code sections also synchronized on "this." It has no effect on other fields/methods of "this" that aren't synchronized.

**MUTEX & SEMAPHRORE:**

A Mutex is a mutually exclusive flag

* Is the number of free identical toilet keys. Example, say we have four toilets with identical locks and keys. The semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free), then the count value is decremented as people are coming in. If all toilets are full, ie. there are no free keys left, the semaphore count is 0. Now, when eq. one person leaves the toilet, semaphore is increased to 1 (one free key), and given to the next person in the queue.
* When I am having a big heated discussion at work, I use a rubber chicken which I keep in my desk for just such occasions. The person holding the chicken is the only person who is allowed to talk. If you don't hold the chicken you cannot speak. You can only indicate that you want the chicken and wait until you get it before you speak. Once you have finished speaking, you can hand the chicken back to the moderator who will hand it to the next person to speak. This ensures that people do not speak over each other, and also have their own space to talk.

Replace Chicken with Mutex and person with thread and you basically have the concept of a mutex.

Officially: "A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore)." Ref: Symbian Developer Library

* Mutex is basically mutual exclusion. Only one thread can acquire the resource at once. When one thread acquires the resource, no other thread is allowed to acquire the resource until the thread owning the resource releases. All threads waiting for acquiring resource would be blocked.

Semaphore is used to control the number of threads executing. There will be fixed set of resources. The resource count will gets decremented every time when a thread owns the same. When the semaphore count reaches 0 then no other threads are allowed to acquire the resource. The threads get blocked till other threads owning resource releases.

In short, the main difference is *how many threads are allowed to acquire the resource at once ?*

* Mutex --its ONE.
* Semaphore -- its DEFINED\_COUNT,
* Semaphore can be counted, while mutex can only count to 1.

Suppose you have a thread running which accepts client connections. This thread can handle 10 clients simultaneously. Then each new client sets the semaphore until it reaches 10. When the Semaphore has 10 flags, then your thread won't accept new connections

Mutex are usually used for guarding stuff. Suppose your 10 clients can access multiple parts of the system. Then you can protect a part of the system with a mutex so when 1 client is connected to that sub-system, no one else should have access. You can use a Semaphore for this purpose too. A mutex is a ["Mutual Exclusion Semaphore"](http://www.javaworld.com/javaworld/jw-10-1998/jw-10-toolbox.html).

Example for Mutex: lock/Unlock

Semaphore: Increament/Decrement.

**Spinlock:** Use a spinlock when you really want to use a mutex but your thread is not allowed to sleep. e.g.: An interrupt handler within OS kernel must never sleep. If it does the system will freeze / crash. If you need to insert a node to globally shared linked list from the interrupt handler, acquire a spinlock - insert node - release spinlock.

* Semaphores have no notion of ownership, this means that any thread can release a semaphore (this can lead to many problems in itself but can help with "death detection"). Whereas a mutex does have the concept of ownership (i.e. you can only release a mutex you have acquired).  
  Ownership is incredibly important for safe programming of concurrent systems. I would always recommend using mutex in preference to a semaphore (but there are performance implications).

Mutexes also may support priority inheritance (which can help with the priority inversion problem) and recursion (eliminating one type of deadlock).

It should also be pointed out that there are "binary" semaphores and "counting/general" semaphores. Java's semaphore is a counting semaphore and thus allows it to be initialized with a value greater than one (whereas, as pointed out, a mutex can only a conceptual count of one). The usefulness of this has been pointed out in other posts.

So to summarize, unless you have multiple resources to manage, I would always recommend the mutex over the semaphore

**http://www.oracle.com/technetwork/articles/javase/index-140767.html**

public class Mutex {

public void acquire() throws InterruptedException { }

public void release() { }

public boolean attempt(long msec) throws InterruptedException { }

}

//The above class can be used as:

try {

mutex.acquire();

try {

// do something

} finally {

mutex.release();

}

} catch(InterruptedException ie) {

// ...

}

**Semphore code:** **http://stackoverflow.com/questions/34519/what-is-a-semaphore?rq=1**

**Exception Handling:*It is illegal to use a*** try ***clause without either a*** catch ***clause or a*** finally ***clause. A*** try ***clause by itself will result in a compiler error. Any*** catch ***clauses must immediately follow the*** try ***block. Any*** finally ***clause must immediately follow the last*** catch ***clause (or it must immediately follow the*** try ***block if there is no*** catch***). It is legal to omit either the*** catch ***clause or the*** finally ***clause, but not both.***

* the call stack is the chain of methods that your program executes to get to the current method. If your program starts in method main() and main() calls method a(), which calls method b(), which in turn calls method c(), the call stack consists of the following:

c

b

a

main

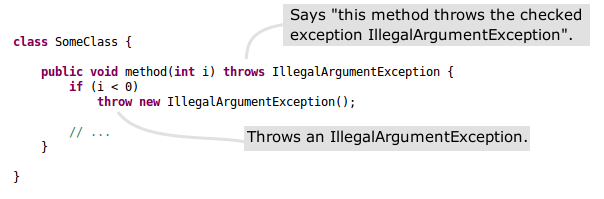
* An exception is first thrown from the top of the stack (in other words, c ), and if it isn't caught by the same person or method(c) who threw it (method c), it drops down the call stack to the previous method b, which is the person standing on the deck one floor down. If not caught there, by the person one floor down, the exception/ball again drops down to the previous method (person on the next floor down), and so on until it is caught or until it reaches the very bottom of the call stack. This is called exception propagation.
* *You can throw the exception out of* main() *as well.*
* throws: Used when writing methods, to declare that the method in question throws the specified (checked) exception.

As opposed to checked exceptions, runtime exceptions (NullPointerExceptions etc) may be thrown without having the method declare throws NullPointerException.

* throw: Instruction to actually throw the exception. (Or more specifically, the *Throwable*).

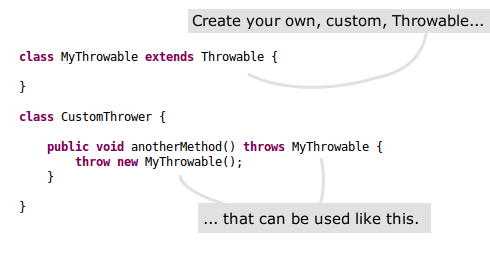
The throw keyword is followed by a reference to a Throwable (usually an exception).

**Example:**



* Throwable: A class which you must extend in order to create your own, custom, throwable.

**Example:**

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

* All exception classes are subtypes of class Exception. This class derives from the class Throwable (which derives from the class Object).
* When an exception is thrown, Java will try to find (by looking at the available catch clauses from the top down) a catch clause for the exception type. If it doesn't find one, it will search for a handler for a supertype of the exception. If it does not find a catch clause that matches a supertype for the exception, then the exception is propagated down the call stack. This process is called exception matching.
* The subclass exception should come first then the superclass. Example:
* ArrayIndexOutOfBound and StringIndexoutOfBound are subclass of IndexOutOfBound. So we should have our catch statement as:

Catch(ArrayIndexOutOfBoundException ae){}

Catch(IndexOutOfBoundException ie){} //If we reverse the order the class wont compile.

exception java.io. ArrayIndexOutOfBoundException has already been caught

* If one Exception class is not a subtype or supertype of the other, then the order in which

the catch clauses are placed doesn't matter. Same as above it doesn’t matter if we write arrayIndex exception or Stringindex first or vice versa.

void doStuff() {

doMore();

}

void doMore() {

throw new IOException();

}

* First, the doMore() method throws a checked exception, but does not declare it! But suppose we fi x the doMore() method as follows:
* void doMore() throws IOException { … }
* The doStuff() method is still in trouble because it, too, must declare the
* IOException, unless it handles it by providing a try/catch, with a catch clause that can take an IOException.
* When an object of a subtype of Exception is thrown, it must be handled or declared. These objects are called checked exceptions, and include all exceptions except those that are subtypes of RuntimeException,

**Inner Class:**

class MyOuter {

class MyInner { }

}

Piece of cake. And if you compile it,

%javac MyOuter.java

you'll end up with *two* class files:

MyOuter.class

MyOuter$MyInner.class

* a regular inner class can't have static declarations of any kind. The only way you can access the inner class is through a live instance of the outer class! Hence no main method. There is no semicolon at the end of regular inner class.
* Instantiating a regular Inner class outside the regular outer class.

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

Instantiating a Method local class:

class MyOuter2 {

private String x = "Outer2";

void doStuff() {

class MyInner {

public void seeOuter() {

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

} // close inner class method

} // close inner class definition

MyInner mi = new MyInner(); // This line must come

// after the class

mi.seeOuter();

} // close outer class method doStuff()

}

* 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.
* the inner class object cannot use the local variables of the method the inner class is in. Explained:

The local variables of the method live on the stack, and exist only for the lifetime of the method. You already know that the scope of a local variable is limited to the method the variable is declared in. When the method ends, the stack

frame is blown away and the variable is history. But even after the method completes, the inner class object created within it might still be alive on the heap if, for example, a reference to it was passed into some other code and then stored in an instance variable. Because the local variables aren't guaranteed to be alive as long as the method-local inner class object, the inner class object can't use them. Unless the local variables are marked final!

* the whole point of making an anonymous inner class—to override one or more methods of the superclass! (Or to implement methods of an interface

class Popcorn {

public void pop() {

System.out.println("popcorn");

}

}

class Food {

**Popcorn p = new Popcorn() {**

public void pop() {

System.out.println("anonymous popcorn");

}

**};**

}

* The Popcorn reference variable refers not to an instance of Popcorn, but to an instance of an anonymous (unnamed) subclass of Popcorn.

In the previous examples,

we defined a new anonymous subclass of type Popcorn as follows:

Popcorn p = new Popcorn() {

But if Popcorn were an interface type instead of a class type, then the new anonymous class would be an implementer of the interface rather than a subclass of

the class.

interface Cookable {

public void cook();

}

class Food {

Cookable c = new Cookable() {

public void cook() {

System.out.println("anonymous cookable implementer");

}

};

}

anonymous interface implementers—*they can implement only one interface*.

Source code of Equals Method in String class //NOTE: it is overridden.

public boolean More ...equals(Object anObject) {

if (this == anObject) {

return true;

}

if (anObject instanceof String) {

String anotherString = (String)anObject;

int n = count;

if (n == anotherString.count) {

char v1[] = value;

char v2[] = anotherString.value;

int i = offset;

int j = anotherString.offset;

while (n-- != 0) {

if (v1[i++] != v2[j++])

return false;

}

return true;

}

}

return false;

}

instanceOf checks for null as well, so if anObject is null the instanceOf will return false;

in regular scenario if we don’t override the equals method, what will happen is the .equals method called will be from the Object class, and it has only the below statement:

public boolean equals(Object obj) {

return (this == obj);

}

Hence same as the string class we have to override the equals method and give our own functionality to the equals method.

Example:

Class Car{

Private String model;

Private String year;

//getters and setters.

}

Class Driver{

Main(){

Car car1 = new Car(“Porsche”, “1992”);

Car car2 = new Car(“Porsche”, “1992”);

//if we do car1.equals(car2) internally the object class’s equals method is called and the equal method has only (this==obj) which is false in this case. If it would be car1.equals(car1), it will assert true. Because the this object is same car1.

The equals object should satisfy the below rules:

Reflexive:

A==A

Symmetry

A==B

Transitivity

And comparing to null gives always false and not nullpointerexception.

1.If equals returns true hashCode should return true.

2. if hashCode is invoked on the same object it should return always the same value. Unless if object is change.

3. Sim to return different hashCode if equals return false. //not necessary but should improve performance.

Override only equals

If only equals is overriden, then when you call myMap.put(first,someValue) first will hash to some bucket and when you call myMap.put(second,someOtherValue) it will hash to some other bucket (as they have a different hashCode). So, although they are equal, as they don't hash to the same bucket, the map can't realize it and both of them stay in the map.

Although it is not necessary to override equals() if we override hashCode(), let's see what would happen in this particular case where we know that two objects of MyClass are equal if their importantField is equal but we do not override equals().

Override only hashCode

Imagine you have this

MyClass first = new MyClass("a","first");

MyClass second = new MyClass("a","second");

If you only override hashCode then when you call myMap.put(first,someValue) it takes first, calculates its hashCode and stores it in a given bucket. Then when you call myMap.put(second,someOtherValue) it should replace first with second as per the [Map Documentation](http://java.sun.com/j2se/1.4.2/docs/api/java/util/Map.html#put%28java.lang.Object,%20java.lang.Object%29) because they are equal (according to the business requirement).

But the problem is that equals was not redefined, so when the map hashes second and iterates through the bucket looking if there is an object k such that second.equals(k) is true it won't find any as second.equals(first) will be false

protected void finalize()

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

The general contract of finalize is that it is invoked if and when the JavaTM virtual machine has determined that there is no longer any means by which this object can be accessed by any thread that has not yet died, except as a result of an action taken by the finalization of some other object or class which is ready to be finalized. The finalize method may take any action, including making this object available again to other threads; the usual purpose of finalize, however, is to perform cleanup actions before the object is irrevocably discarded. For example, the finalize method for an object that represents an input/output connection might perform explicit I/O transactions to break the connection before the object is permanently discarded.

The finalize method of class Object performs no special action; it simply returns normally. Subclasses of Object may override this definition.

The Java programming language does not guarantee which thread will invoke the finalize method for any given object. It is guaranteed, however, that the thread that invokes finalize will not be holding any user-visible synchronization locks when finalize is invoked. If an uncaught exception is thrown by the finalize method, the exception is ignored and finalization of that object terminates.

After the finalize method has been invoked for an object, no further action is taken until the Java virtual machine has again determined that there is no longer any means by which this object can be accessed by any thread that has not yet died, including possible actions by other objects or classes which are ready to be finalized, at which point the object may be discarded.

The finalize method is never invoked more than once by a Java virtual machine for any given object.

Any exception thrown by the finalize method causes the finalization of this object to be halted, but is otherwise ignored.

**Strings:**

String a = "abc"; // 1 Object : "abc" added to pool

String b = "abc"; // 0 Object : because it is already in the pool

String c = new String("abc"); // 1 Object

String d = new String("def"); // 1 Object + "def" is added to the Pool

String e = d.intern(); // (e==d) is "false" because e refers to the String in pool

String f = e.intern(); // (f==e) is "true"

Total Objects: 4 ("abc", c, d, "def").