

Object Oriented Design Principles - Software Design Principles

Single Responsibility Principle

Principle: There should be only one reason to change the class. If we have 2 reasons to change for a class, we have to split the functionality in two classes. It speaks about more cohesive classes, it means each class should have a focussed set of responsibility.

Example:

Suppose there is a requirement to download the file- may be in csv/json/xml format, parse the file and then update the contents into a database or file system. One approach would be to-

```
public class Task {
    public void downloadFile(location) {
        //Download the file
    }
    public void parseTheFile(file) {
        //Parse the contents of the file- XML/JSON/CSV
    }
    public void persistTheData(data) {
        //Persist the data to Database or file system.
    }
}
```

It looks good, all in one place easy to understand. But what about the number of times this class has to be updated? What about the reusability of parser code? or download code? Its not good design in terms of reusability of different parts of the code, in terms of cohesiveness.

One way to decompose the Task class is to create different classes for downloading the file- Downloader, for parsing the file- Parser and for persisting to the database or file system.

Open Close Principle

Principle : In object-oriented programming, the open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"; that is, such an entity can allow its behaviour to be modified without altering its source code.

Example:

Suppose you are writing a module to approve personal loans and before doing that you want to validate the personal information, code wise we can depict the situation as:

```
public class LoanApprover
{
    public void approveLoan(PersonalValidator validator)
    {
        if ( validator.isValid())
        {
            //Process the loan.
        }
    }
}
public class PersonalLoanValidator
{
    public boolean isValid()
    {
        //Validation logic
    }
}
```

As you all know the requirements are never the same and now its required to approve vehicle loans, consumer goods loans and what not. So one approach to solve this requirement is to:

```
public class LoanApprovalHandler
{
    public void approvePersonalLoan (PersonalLoanValidator validator)
    {
        if ( validator.isValid())
        {
            //Process the loan.
        }
    }
    public void approveVehicleLoan (VehicleLoanValidator validator )
    {
        if ( validator.isValid())
        {
            //Process the loan.
        }
    }
    // Method for approving other loans.
}
public class PersonalLoanValidator
{
    public boolean isValid()
    {
        //Validation logic
    }
}
public class VehicleLoanValidator
{
    public boolean isValid()
    {
        //Validation logic
    }
}
```

We have edited the existing class to accomodate the new requirements- in the process we ended up changing the name of the existing method and also adding new methods for different types of loan approval. This clearly violates the OCP. Lets try to implement the requirement in a different way

```

/**
 * Abstract Validator class
 * Extended to add different
 * validators for different loan type
 */
public abstract class Validator
{
    public boolean isValid();
}
/**
 * Personal loan validator
 */
public class PersonalLoanValidator extends Validator
{
    public boolean isValid()
    {
        //Validation logic.
    }
}
/*
 * Similarly any new type of validation can
 * be accommodated by creating a new subclass
 * of Validator
 */

```

Now using the above validators we can write a LoanApprovalHandler to use the Validator abstraction.

```

public class LoanApprovalHandler
{
    public void approveLoan(Validator validator)
    {
        if ( validator.isValid())
        {
            //Process the loan.
        }
    }
}

```

So to accommodate any type of loan validators we would just have create a subclass of Validator and then pass it to the approveLoan method. That way the class is **CLOSED** for modification but **OPEN** for extension.

Liskov Substitution Principle

Principle : Derived types must be completely substitutable for their base types. It means we must make sure that new derived classes are extending the base classes without changing the behaviour.

Example :

Let us see the violation of this principle.

```
class Bird {
    public void fly(){}
    public void eat(){}
}
class Crow extends Bird {}
class Ostrich extends Bird{
    fly(){
        throw new UnsupportedOperationException();
    }
}

public BirdTest{
    public static void main(String[] args){
        List<Bird> birdList = new ArrayList<Bird>();
        birdList.add(new Bird());
        birdList.add(new Crow());
        birdList.add(new Ostrich());
        letTheBirdsFly ( birdList );
    }
    static void letTheBirdsFly ( List<Bird> birdList ){
        for ( Bird b : birdList ) {
            b.fly();
        }
    }
}
```

As soon as an Ostrich instance is passed, it blows up!!! Here the sub type is not replaceable for the super type. How do we fix such issues ?

By using factoring. Sometimes factoring out the common features into a separate class can help in creating a hierarchy that conforms to LSP.

In the above scenario we can factor out the fly feature into- Flight and NonFlight birds.

```
class Bird{
    public void eat(){}
}
class FlyingBird extends Bird{
    public void fly(){}
}
class NonFlying extends Bird{}
```

Interface Segregation Principle(ISP)

Principle : Clients should not be forced to depend upon interfaces that they don't use.

Example :

If we create an interface called Worker and a method called lunch break, all workers have to implement it. What if the worker is a robot.

Let us consider another example.

Consider there is one interface Travel

Travel has two methods

travelbyBus()

travelByTrain()

Now there is one class Passenger . Passenger will always travel by bus .

if Passenger class implements Travel interface

Passenger implements Passenger {

```
travelbyBus(){  
}
```

```
travelByTrain(){  
}
```

```
}
```

It is forced to implement both the methods travelByBus() and travelByTrain() , though Passenger has nothing to do with travelByTrain().

Now let's read the principle again

Interface Segregation Principle states that, a client should not implement an interface, if it doesn't use that.

Client Passenger should not be forced to implement travelByTrain() method as it never needs it.

Here comes the Interface designing best practice .

An interface design should be very thoughtful. It should always take in consideration all its possible clients and should always make interface design client friendly. So could that be achieved?

Instead of creating one interface for both methods , two separate interfaces could be designed like :

BusTravel{

```
travelByBus();  
}
```

TrainTravel{

```
travelByTrain();  
}
```

Passenger class can implement BusTravel interface and will not be forced to implement anymore method.

Dependency inversion principle

Principle : High-level modules should not depend on low-level modules. Both should depend on abstractions.

Let us consider a bad example

// Dependency Inversion Principle - Bad example

```
class Worker {
    public void work() {
        // ....working
    }
}

class Manager {
    Worker worker;
    public void setWorker(Worker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}

class SuperWorker {
    public void work() {
        //.... working much more
    }
}
```

Below is the code which supports the Dependency Inversion Principle. In this new design a new abstraction layer is added through the IWorker Interface. Now the problems from the above code are solved(considering there is no change in the high level logic):

Manager class doesn't require changes when adding SuperWorkers. Minimized risk to affect old functionality present in Manager class since we don't change it. No need to redo the unit testing for Manager class.

// Dependency Inversion Principle - Good example

```
interface IWorker {
    public void work();
}

class Worker implements IWorker{
    public void work() {
        // ....working
    }
}

class SuperWorker implements IWorker{
    public void work() {
        //.... working much more
    }
}
```

```
class Manager {  
    IWorker worker;  
  
    public void setWorker(IWorker w) {  
        worker = w;  
    }  
  
    public void manage() {  
        worker.work();  
    }  
}
```

1. Stateless and Stateful

A stateless server is a server that treats each request as an independent transaction that is unrelated to any previous request.

The difference is in the nature of the connection they define between a client and a server. Telnet and FTP, for example are stateful protocols.

2. Pass by value and pass by reference

Most methods passed arguments when they are called. An argument may be a constant or a variable.

For example in the expression: `Math.sqrt(x)`; The variable `x` is passed here.

Pass by Reference means the passing the address itself rather than passing the value and pass by value means passing a copy of the value as an argument.

This is simple enough, however there is an important but simple principle at work here. If a variable is passed, the method receives a copy of the variable's value. The value of the original variable cannot be changed within the method. This seems reasonable because the method only has a copy of the value; it does not have access to the original variable. This process is called pass by value.

However, if the variable passed is an object, then the effect is different. We often say things like, "this method returns an object ...", or "this method is passed an object as an argument ..." But this is not quite true, more precisely, we should say, something like "this method returns a reference to an object ..." or "this method is passed a reference to an object as an argument ..."

Generally, objects are never passed to methods or returned by methods. It is always "a reference to an object" that is passed or returned. In general, pass by value refers to passing a constant or a variable holding a primitive data type to a method, and pass by reference refers to passing an object variable to a method. In both cases a copy of the variable is passed to the method. It is a copy of the "value" for a primitive data type variable; it is a copy of the "reference" for an object variable. So, a method receiving an object variable as an argument receives a copy of the reference to the original object.

Here's the clincher: If the method uses that reference to make changes to the object, then the original object is changed. This is reasonable because both the original reference and the copy of the reference "refer to" to same thing – the original object. There is one exception: strings. Since String objects are immutable in Java, a method that is passed a reference to a String object cannot change the original object.

To understand pass by reference lets see the sample program below:


```
public class TestPassByReference {

    public static void main(String[] args) {

        // declare and initialize variables and objects

        int i = 25;

        String s = "Java is fun!";

        StringBuffer sb = new StringBuffer("Hello, world");


        // print variable i and objects s and sb

        System.out.println(i);    // print it (1)

        System.out.println(s);    // print it (2)

        System.out.println(sb);   // print it (3)

        // attempt to change i, s, and sb using methods

        iMethod(i);

        sMethod(s);

        sbMethod(sb);

        // print variable i and objects s and sb (again)

        System.out.println(i);    // print it (7)

        System.out.println(s);    // print it (8)

        System.out.println(sb);   // print it (9)

    }

    public static void iMethod(int iTest) {

        iTest = 9;                // change it

        System.out.println(iTest); // print it (4)

        return;

    }

    public static void sMethod(String sTest) {

        sTest = sTest.substring(8, 11); // change it

        System.out.println(sTest);    // print it (5)

        return;

    }

}
```

```

public static void sbMethod(StringBuffer sbTest) {

    sbTest = sbTest.insert(7, "Java "); // change it

    System.out.println(sbTest);        // print it (6)

    return;

}

}

```

Output of the program :

25

Java is fun!

Hello, world

9

fun

Hello, Java world

25


Java is fun!

Hello, Java world

TestPassByReference begins by declaring and initializing three variables: an int variable named i, a String object variable named s, and a StringBuffer object variable named sb. The values are then printed. Then, each variable is passed as an argument to a method. Within each method, the copy of the variable exists as a local variable. The value of the variable “ or the value of the object referred to by the variable, in the case of the String and StringBuffer object variables ” is changed and printed within each method. The print statements are numbered to show the order of printing. Back in the main() method, the three values are printed again. Have a look at the output and see if it is consistent with our previous discussion.

The pass-by-reference concept is illustrated by the object variables sb and sbTest. In the main() method, a StringBuffer object is instantiated and initialized with "Hello, world" and a reference to it is assigned to the StringBuffer object variable sb.

If Java uses the pass-by reference, why won't a swap function work?

Your question demonstrates a common error made by Java language newcomers. Indeed, even seasoned veterans  find it difficult to keep the terms straight.

Java does manipulate objects by reference, and all object variables are references. However, Java doesn't pass method arguments by reference; it passes them by value.

Take the **badSwap()** method for example:

```

public void badSwap(int var1, int var2)
{
    int temp = var1;
    var1 = var2;
    var2 = temp;
}

```

When **badSwap()** returns, the variables passed as arguments will still hold their original values. The method will also fail if we change the arguments type from **int** to **Object**, since Java passes object references by value as well. Now, here is where it gets tricky:

```

public void tricky(Point arg1, Point arg2)
{
    arg1.x = 100;
    arg1.y = 100;
    Point temp = arg1;
    arg1 = arg2;
    arg2 = temp;
}

public static void main(String [] args)
{
    Point pnt1 = new Point(0,0);
    Point pnt2 = new Point(0,0);
    System.out.println("X: " + pnt1.x + " Y: " + pnt1.y);
    System.out.println("X: " + pnt2.x + " Y: " + pnt2.y);
    System.out.println(" ");
    tricky(pnt1,pnt2);
    System.out.println("X: " + pnt1.x + " Y:" + pnt1.y);
    System.out.println("X: " + pnt2.x + " Y: " + pnt2.y);
}

```

If we execute this **main()** method, we see the following output:

```

X: 0 Y: 0
X: 0 Y: 0
X: 100 Y: 100
X: 0 Y: 0

```

The method successfully alters the value of **pnt1**, even though it is passed by value; however, a swap of **pnt1** and **pnt2** fails! This is the major source of confusion. In the **main()** method, **pnt1** and **pnt2** are nothing more than object references. When you pass **pnt1** and **pnt2** to the **tricky()** method, Java passes the references by value just like any other parameter. This means the references passed to the method are

actually **copies** of the original references. Figure 1 below shows two references pointing to the same object after Java passes an object to a method.

Figure 1. After being passed to a method, an object will have at least two references

Java copies and passes the **reference** by value, not the object. Thus, method manipulation will alter the objects, since the references point to the original objects. But since the references are copies, swaps will fail. As Figure 2 illustrates, the method references swap, but not the original references. Unfortunately, after a method call, you are left with only the unswapped original references. For a swap to succeed outside of the method call, we need to swap the original references, not the copies.

Figure 2. Only the method references are swapped, not the original ones

Java is Pass-by-Value, Dammit!

Introduction

I finally decided to write up a little something about Java's parameter passing. I'm really tired of hearing folks (incorrectly) state "primitives are passed by value, objects are passed by reference".

I'm a compiler guy at heart. The terms "pass-by-value" semantics and "pass-by-reference" semantics have very precise definitions, and they're often horribly abused when folks talk about Java. I want to correct that... The following is how I'd describe these

Pass-by-value

The actual parameter (or argument expression) is fully evaluated and the resulting value is *copied* into a location being used to hold the formal parameter's value during method/function execution. That location is typically a chunk of memory on the runtime stack for the application (which is how Java handles it), but other languages could choose parameter storage differently.

Pass-by-reference

The formal parameter merely acts as an *alias* for the actual parameter. Anytime the method/function uses the formal parameter (for reading or writing), it is actually using the actual parameter.

Java is **strictly** pass-by-value, exactly as in C. Read the Java Language Specification (JLS). It's spelled out, and it's correct

When the method or constructor is invoked ([§5.12](#)), the **values** of the actual argument expressions initialize newly created parameter variables, each of the declared **Type**, before execution of the body of the method or constructor. The **Identifier** that appears in the **DeclaratorId** may be used as a simple name in the body of the method or constructor to refer to the formal parameter.

[In the above, values is my emphasis, not theirs]

In short: Java **has** pointers and is strictly pass-by-value. There's no funky rules. It's simple, clean, and clear. (Well, as clear as the evil C++-like syntax will allow ;)

Note: See the [note at the end of this article](#) for the semantics of remote method invocation (RMI). What is typically called "pass by reference" for remote objects is actually incredibly bad semantics.

The Litmus Test

There's a simple "litmus test" for whether a language supports pass-by-reference semantics:

Can you write a traditional swap(a,b) method/function in the language?

A traditional swap method or function takes two arguments and swaps them such that variables passed into the function are changed outside the function. Its basic structure looks like

Figure 1: (Non-Java) Basic swap function structure

```
swap(Type arg1, Type arg2) {  
    Type temp = arg1;  
    arg1 = arg2;  
    arg2 = temp;  
}
```

If you can write such a method/function in your language such that calling

Figure 2: (Non-Java) Calling the swap function

```
Type var1 = ...;  
Type var2 = ...;  
swap(var1, var2);
```

actually switches the values of the variables var1 and var2, the language supports pass-by-reference semantics.

But you cannot do this in Java!

Now the details...

The problem we're facing here is statements like

In Java, Objects are passed by reference, and primitives are passed by value.

This is half incorrect. Everyone can easily agree that primitives are passed by value; there's no such thing in Java as a pointer/reference to a primitive.

However, *Objects are not passed by reference*. A correct statement would be *Object references are passed by value*.

This may seem like splitting hairs, but it is *far* from it. There is a world of difference in meaning. The following examples should help make the distinction.

In Java, take the case of

Figure 5: (Java) Pass-by-value example

```
public void foo(Dog d) {  
    d = new Dog("Fifi"); // creating the "Fifi" dog  
}
```

```
Dog aDog = new Dog("Max"); // creating the "Max" dog  
// at this point, aDog points to the "Max" dog  
foo(aDog);  
// aDog still points to the "Max" dog
```

the variable passed in (aDog) **is not** modified! After calling foo, aDog **still** points to the "Max" Dog!

Many people mistakenly think/state that something like

Figure 6: (Java) Still pass-by-value...

```
public void foo(Dog d) {  
    d.setName("Fifi");  
}
```

shows that Java does in fact pass objects by reference.

The mistake they make is in the definition of

Figure 7: (Java) Defining a Dog pointer

```
Dog d;
```

itself. When you write that definition, you are defining a *pointer* to a Dog object, *not* a Dog object itself.

On Pointers versus References...

The problem here is that the folks at Sun made a naming mistake.

In programming language design, a "pointer" is a variable that indirectly tracks the location of some piece of data. The value of a pointer is often the memory address of the data you're interested in. Some languages allow you to manipulate that address; others do not.

A "reference" is an alias to another variable. Any manipulation done to the reference variable directly changes the original variable.

Check out the second sentence

of http://java.sun.com/docs/books/jls/third_edition/html/typesValues.html#4.3.1.

"The reference values (often just *references*) are *pointers* to these objects, and a special null reference, which refers to no object"

They emphasize "pointers" in their description... Interesting...

When they originally were creating Java, they had "pointer" in mind (you can see some remnants of this in things like `NullPointerException`).

Sun wanted to push Java as a secure language, and one of Java's advantages was that it does not allow pointer arithmetic as C++ does.

They went so far as to try a different name for the concept, formally calling them "references". A big mistake and it's caused even more confusion in the process.

There's a good explanation of reference variables

at <http://www.cprogramming.com/tutorial/references.html>. (C++ specific, but it says the right thing about the concept of a reference variable.)

The word "reference" in programming language design originally comes from how you pass data to subroutines/functions/procedures/methods. A reference parameter is an alias to a variable passed as a parameter.

In the end, Sun made a naming mistake that's caused confusion. Java has pointers, and if you accept that, it makes the way Java behaves make much more sense.

Calling Methods

Calling

Figure 8: (Java) Passing a pointer by value

```
foo(d);
```

passes the **value of d** to foo; it does not pass the object that d points to!

The value of the pointer being passed is similar to a memory address. Under the covers it may be a tad different, but you can think of it in exactly the same way. The value uniquely identifies some object on the heap.

However, it makes no difference how pointers are **implemented** under the covers. You program with them **exactly** the same way in Java as you would in C or C++. The syntax is just slightly different (another poor choice in Java's design; they should have used the same `->` syntax for de-referencing as C++).

In Java,

Figure 9: (Java) A pointer

```
Dog d;
```

is **exactly** like C++'s

Figure 10: (C++) A pointer

```
Dog *d;
```

And using

Figure 11: (Java) Following a pointer and calling a method

```
d.setName("Fifi");
```

is exactly like C++'s

Figure 12: (C++) Following a pointer and calling a method

```
d->setName("Fifi");
```

To sum up: Java **has** pointers, and the **value** of the **pointer** is passed in. There's no way to actually pass an object itself as a parameter. You can only pass a pointer to an object.

Keep in mind, when you call

Figure 13: (Java) Even more still passing a pointer by value

```
foo(d);
```

you're not passing an object; you're passing a **pointer** to the object.

For a slightly different (but still correct) take on this issue, please see <http://www-106.ibm.com/developerworks/library/j-praxis/pr1.html>. It's from Peter Haggard's excellent book, *Practical Java*.)

A Note on Remote Method Invocation (RMI)

When passing parameters to remote methods, things get a bit more complex. First, we're (usually) dealing with passing data between two independent virtual machines, which might be on separate physical machines as well. Passing the value of a pointer wouldn't do any good, as the target virtual machine doesn't have access to the caller's heap.

You'll often hear "pass by value" and "pass by reference" used with respect to RMI. These terms have more of a "logical" meaning, and really aren't correct for the intended use.

Here's what is usually meant by these phrases with regard to RMI. Note that this is *not* proper usage of "pass by value" and "pass by reference" semantics:

RMI Pass-by-value

The actual parameter is *serialized* and passed using a network protocol to the target remote object. Serialization essentially "squeezes" the data out of an object/primitive. On the receiving end, that data is used to build a "clone" of the original object or primitive. Note that this process can be rather expensive if the actual parameters point to large objects (or large graphs of objects).

This isn't quite the right use of "pass-by-value"; I think it should really be called something like "pass-by-memento". (See "Design Patterns" by Gamma et al for a description of the Memento pattern).

RMI Pass-by-reference

The actual parameter, which *is itself a remote object*, is represented by a proxy. The proxy keeps track of where the actual parameter lives, and anytime the target method uses the formal parameter, *another remote method invocation occurs* to "call back" to the actual parameter. This can be useful if the actual parameter points to a large object (or graph of objects) and there are few call backs.

This isn't quite the right use of "pass-by-reference" (again, you cannot change the actual parameter itself). I think it should be called something like "pass-by-proxy". (Again, see "Design Patterns" for descriptions of the Proxy pattern).

Follow up from stackoverflow.com

I posted the following as some clarification when a discussion on this article arose on <http://stackoverflow.com>.

The Java Spec says that everything in java is pass-by-value. There is no such thing as "pass-by-reference" in java.

The key to understanding this is that something like

Figure 14: (Java) Not a Dog; a pointer to a Dog

```
Dog myDog;
```

is not a Dog; it's actually a pointer to a Dog.

What that means, is when you have

Figure 15: (Java) Passing the Dog's location

```
Dog myDog = new Dog("Rover");  
foo(myDog);
```

you're essentially passing the address of the created Dog object to the foo method. (I say essentially b/c java pointers aren't direct addresses, but it's easiest to think of them that way)

Suppose the Dog object resides at memory address 42. This means we pass 42 to the method.

If the Method were defined as

Figure 16: (Java) Looking at the called method in detail

```
public void foo(Dog someDog) {
```



```
someDog.setName("Max"); // AAA
someDog = new Dog("Fifi"); // BBB
someDog.setName("Rowlf"); // CCC
}
```

Let's look at what's happening.

the parameter someDog is set to the value 42

at line "AAA"

someDog is followed to the Dog it points to (the Dog object at address 42) that Dog (the one at address 42) is asked to change his name to Max

at line "BBB"

a new Dog is created. Let's say he's at address 74 we assign the parameter someDog to 74

at line "CCC"

someDog is followed to the Dog it points to (the Dog object at address 74) that Dog (the one at address 74) is asked to change his name to Rowlf then, we return

Now let's think about what happens outside the method:

Did myDog change?

There's the key.

Keeping in mind that myDog is a pointer, and not an actual Dog, the answer is NO. myDog still has the value 42; it's still pointing to the original Dog.

It's perfectly valid to follow an address and change what's at the end of it; that does not change the variable, however.

Java works exactly like C. You can assign a pointer, pass the pointer to a method, follow the pointer in the method and change the data that was pointed to. However, you cannot change where that pointer points.

In C++, Ada, Pascal and other languages that support pass-by-reference, you can actually change the variable that was passed.

If Java had pass-by-reference semantics, the foo method we defined above would have changed where myDog was pointing when it assigned someDog on line BBB.

Think of reference parameters as being aliases for the variable passed in. When that alias is assigned, so is the variable that was passed in.

Cup Size -- a story about variables

The Coffee Corral coffeehouse at the Ranch is famous around these parts for its unique cup collection. If you really need to understand the way values and objects are used in Java, you gotta start here.

I like cups. Big cups, little cups. Painted tea cups, huge cappuccino bowls, cups with logos from the UCLA coffeehouse. Cups I painted myself at the "All Fired Up" pottery shop. Cups with curvy, sexy handles. Metallic cups that I now know must never, ever go in the microwave.

So when I think of variables, I naturally think of... cups.

A variable is just a cup. It has a size, and it can hold something.

In Java, cups come in two main styles: primitive and reference.

Primitive cups hold primitive values.

Reference cups hold remote controls to objects.

We'll start with primitives. Primitive cups are like the cups they have at the coffeehouse. If you're familiar with Starbucks' you know what I mean. They come in different sizes, and each size has a name like short, tall, grande. As in, "I'd like a grande mocha java with extra whipped cream. Oh, and use non-fat milk please")

Our coffeehouse has a picture of the cups on the counter, so customers know what to order. It looks like this:

In Java, integer primitives comes in different sizes, and those sizes have names. They look like this:

These cups hold a value.

So instead of saying, "I'd like a tall French Roast", you say to the compiler, "I'd like an int with the number 90 please." And that's what you get. (you also have to give your cup a name, but we'll get to that later.)

The number 90 is dropped into your int-sized cup.

But what about floating point numbers? (the things with decimal points)
They get their own cups too.

And there's another cup for booleans, that can store values of true or false. And a cup for chars, that store single characters like the letter 'c' or 'z'.

In Java, **each of these cups (float, char, long, etc.) is a specific size**. Byte is the smallest, double and long are the largest. Rather than measure in milliliters (or ounces as we do in the US) Java variables have a size measured in bits:

byte - 8 bits
short - 16 bits
int - 32 bits
long - 64 bits

All of these integer types are SIGNED. The leftmost bit represents the sign (positive or negative) and is NOT part of the value. So with a byte, for instance, you don't get the whole 8 bits to represent your value. You get 7. This gives you a range, for bytes, of :
(-2 to the 7th) through (2 to the 7th) -1. Why that little -1 on the end? Because zero is in there, and zero counts as negative. Works the same way with the others.

float - 32 bits
double - 64 bits

Floating point numbers are in the IEEE 754 standard. If that means anything to you, great. If it doesn't, well, then, you'll just have to struggle through the long technical dissertation on floating point numbers which I feel compelled to insert here. What the heck, I'll skip it.

We rejoin our primitive variables, already in progress.

char - 16 bits, UNSIGNED

(Unicode format -- for English, it maps perfectly to ASCII with the high 8-bits just hanging out as a bunch of zeros)

boolean - hmmm... you're not supposed to ask. It holds a value of true or false, but it's really stored as a numeric value, probably in a byte-sized cup. Try not to think about the size; all you know is that it holds a boolean.

Let's get to the Really Interesting Cups... **REFERENCES**

In Java, if you want to stick an object in a variable, remember that the object is created out on the garbage-collectible heap. Always. So it's not IN the variable. There aren't giant, expandable cups which can be made big enough to hold any object. And unlike C/C++, there aren't cups which hold the exact memory location of the object.

In Java, **objects are created on the heap**, and a REFERENCE to the object is stored in the cup. Think of it as a remote control to a specific type of object.

At the CoffeeCorral, a customer can ask for a remote control to a TV They ask for it like this:

"I'd like a reference to a new Sony32 television please, and name it TV." which in Java looks like:

But notice the NAME written on the cup. **The cups have unique names**. Imagine if you had a pile of remote controls, and nobody knew which one controlled which TV. So we make people put names on their remote control cups. But the name is not on the object -- the object has a unique ID, like a serial number, but that isn't the same as what you name the reference! The reference name (the name on the cup) is YOUR choice.

It's the same for primitives -- you name the cups like this:

"I'd like a byte with the number 7, and name it x please".

For object references, you can also ask for a self-serve remote control... a remote control where you don't have a television object picked out yet. You still get the remote in the cup, but you "program" that remote for a specific television later.

In Java you say:

```
Sony32 tv; // declare but don't initialize with an actual Sony32 object.
```

It's like saying, "I'd like a reference to a Sony32 television, and name the remote 'tv', but I'll pick out the actual television later.". You get the cup, you get the remote, but there's no object and the remote isn't controlling ANYTHING.

In Java, a remote which refers to nothing is a reference with a value of null.

So what's REALLY inside that cup? What's a remote control?

In Java, remote controls are called references. They store a value which the Java Virtual Machine (JVM) uses to get to your object. It sure looks and feels a lot like a pointer, and it might very well be a pointer to a pointer, or...

You Can't Know. It's an implementation detail that you, as a programmer, can't access. Don't even think about it. There's no way to use that value other than to access the methods and variables of the actual object the reference refers to. That's part of what makes Java safer than C/C++. You can not go directly to any arbitrary memory location. The JVM allocates memory on your behalf, for your object, and stores an address-like thing in the reference cup (which is most likely a 32-bit cup, but not guaranteed to be).

If this depresses you, take a deep breath, have a nice big cup of hot tea and get over it. When you start doing CORBA you'll be grateful beyond belief that you don't have to take care of the memory.

This story continues with : [Pass-By-Value Please](#).

Pass-by-Value Please (Cup Size continued)

If you haven't read the [Cup Size story](#), this one won't make sense. Or it will make sense, but you'll think it's really stupid. Or you won't think it's stupid but you'll find yourself... never mind, just go read it now and then come back.

I really care about my cups.

I don't want just anybody putting something in my cups. If I have something in a cup, I just want it to stay that way until I decide to change it! So back off!

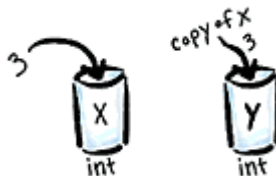
So if a Java variable is a cup, with a value in it, **what does it mean to "pass" a variable** to a method? And are primitives and references treated the same way?

We'll get there, but first let's start with simple assignment.

What does it mean to say:

1) `int x = 3;`

2) `int y = x;`



In line 1, a cup called x, of size int, is created and given the value 3.

In line 2, a cup called y, of size int, is created and given the value... 3.

The x variable is not affected!

Java COPIES the value of x (which is 3) and puts that COPY into y.

This is PASS-BY-VALUE. Which you can think of as PASS-BY-COPY. The value is copied, and that's what gets shoved into the new cup. You don't stuff one cup into another one.

Saying `int y = x` does NOT mean "put the x cup into y". It means "copy the value inside x and put that copy into y".

If I later change y:

```
y = 34;
```

Is x affected? Of course not. The x cup is still sitting there, all happy.

If I later change x:

```
x = 90;
```

Is y affected? Nope. They are disconnected from one another once the assignment was made (the COPY was made).

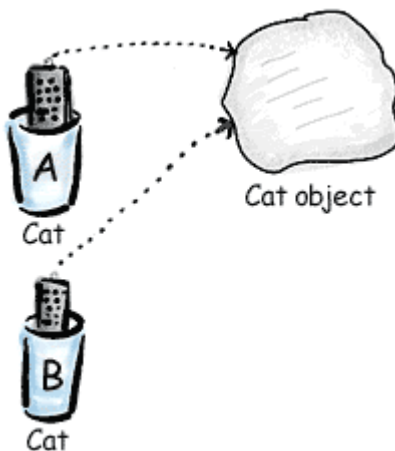
SO... what about Reference Variables (remote controls)? How does THAT work?

Not so tricky, in fact the rule is the same.

References do the same thing. You get a copy of the reference.

So if I say:

```
Cat A = new Cat();  
Cat B = A;
```



The remote control in A is copied. Not the object it refers to.

You've still got just one Cat object.

But now you have **two different references** (remote controls) controlling the **same** Cat object.

NOW let's look at passing values to methods

Java is pass-by-value.

Always.

That means **"copy the value, and pass the copy."**

For primitives, it's easy:

```
int x = 5;  
doStuff(x); // pass a COPY of x (the value 5) to the doStuff method
```

The doStuff method looks like this:

```
void doStuff(int y) {  
    // use y in some way  
}
```

A copy of the value in x, which is 5, is passed into the doStuff() method.

The doStuff() method has its own new cup, called y, waiting.

The y cup is a new, different cup. With a copy of what was in x at the time it was passed.

From this point on, y and x have no affect on each other. If you change y, you don't touch x.

```
void doStuff(int y) {  
    y = 27; // this does NOT affect x  
}
```

And vice-versa. If you change x, you don't change y.

The only part x had in this whole business was to simply copy its value and send that copy into the doStuff() method.

How does pass-by-value work with references?

Way too many people say "Java passes primitive by value and objects by reference". This is not the way it should be stated. Java passes everything by value. With primitives, you get a copy of the contents. With references you get a copy of the contents.

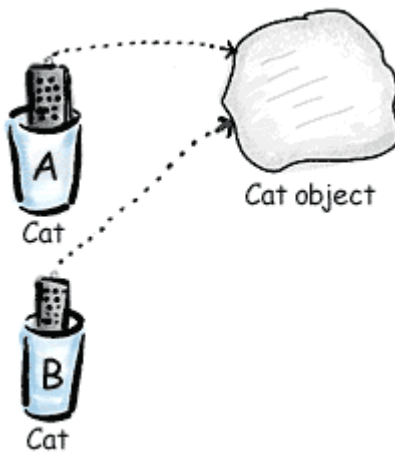
But what is the contents of a reference?

The remote control. The means to control / access the object.

When you pass an object reference into a method, you are passing a COPY of the REFERENCE. A clone of the remote control. The object is still sitting out there, waiting for someone to use a remote. The object doesn't care how many remotes are "programmed" to control it. Only the garbage collector cares (and you, the programmer).

So when you say:

```
Cat A = new Cat();  
doStuff(A);  
  
void doStuff(Cat B) {  
    // use B in some way  
}
```



There is still just ONE Cat object. But now TWO remote controls (references) can access that same Cat object.

So now, anything that B does to the Cat, will affect the Cat that A refers to, but it won't affect the A cup!

You can change the Cat, using your new B reference (copied directly from A), but you can't change A.

What the heck does that mean?

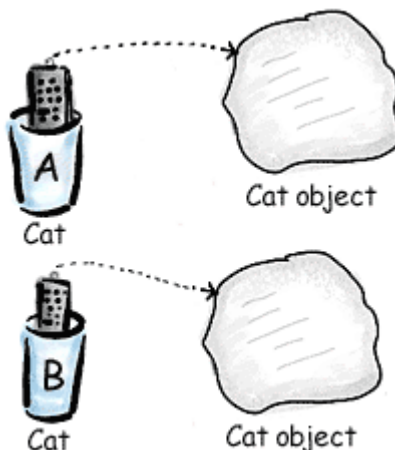
You can change the object A refers to, but you can't take the A reference variable and do something to it -- like redirect it to reference a different object, or null.

So if you change the B reference (not the Cat object B refers to, but the B reference itself) you don't change A. And the opposite is true.

So...

```
Cat A = new Cat();
doStuff(A);
void doStuff(Cat B) {
    B = new Cat(); //did NOT affect the A reference
}
```

Doing this simply "points" B to control a different object. A is still happy.



So repeat after me:

Java is pass-by-value.

(OK, once again... with feeling.)

Java is pass-by-value.

For primitives, you pass a copy of the actual value.

For references to objects, you pass a copy of the reference (the remote control).

You never pass the object. All objects are stored on the heap. Always.

Now go have an extra big cup of coffee and write some code.

What is Shallow Copy?

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

What is Deep Copy?

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

Well, here we are with what shallow copy and deep copy are and obviously the difference between them. Now lets see how to implement them in java.

How to implement shallow copy in java?

Here is an example of Shallow Copy implementation

```
class Subject {  
  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String s) {  
        name = s;  
    }  
  
    public Subject(String s) {  
        name = s;  
    }  
}  
  
class Student implements Cloneable {  
    //Contained object  
    private Subject subj;  
  
    private String name;  
  
    public Subject getSubj() {  
        return subj;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String s) {  
        name = s;  
    }  
}
```

```

public Student(String s, String sub) {
    name = s;
    subj = new Subject(sub);
}

public Object clone() {
    //shallow copy
    try {
        return super.clone();
    } catch (CloneNotSupportedException e) {
        return null;
    }
}
}

public class CopyTest {

    public static void main(String[] args) {
        //Original Object
        Student stud = new Student("John", "Algebra");

        System.out.println("Original Object: " + stud.getName() + " - "
            + stud.getSubj().getName());

        //Clone Object
        Student clonedStud = (Student) stud.clone();

        System.out.println("Cloned Object: " + clonedStud.getName() + " - "
            + clonedStud.getSubj().getName());

        stud.setName("Dan");
        stud.getSubj().setName("Physics");

        System.out.println("Original Object after it is updated: "
            + stud.getName() + " - " + stud.getSubj().getName());

        System.out.println("Cloned Object after updating original object: "
            + clonedStud.getName() + " - " + clonedStud.getSubj().getName());

    }
}

```

Output is:

Original Object: John - Algebra

Cloned Object: John - Algebra

Original Object after it is updated: Dan - Physics

Cloned Object after updating original object: John - Physics

In this example, all I did is, implement the class that you want to copy with Clonable interface and override clone() method of Object class and call super.clone() in it. If you observe, the changes made to "name" field of original object (Student class) is not reflected in cloned object but the changes made to "name" field of contained object (Subject class) is reflected in cloned object. This is because the cloned object carries the memory address of the

Subject object but not the actual values. Hence any updates on the Subject object in Original object will reflect in Cloned object.

How to implement deep copy in java?

Here is an example of Deep Copy implementation. This is the same example of Shallow Copy implementation and hence I didnt write the Subject and CopyTest classes as there is no change in them.

```
class Student implements Cloneable {
    //Contained object
    private Subject subj;

    private String name;

    public Subject getSubj() {
        return subj;
    }

    public String getName() {
        return name;
    }

    public void setName(String s) {
        name = s;
    }

    public Student(String s, String sub) {
        name = s;
        subj = new Subject(sub);
    }

    public Object clone() {
        //Deep copy
        Student s = new Student(name, subj.getName());
        return s;
    }
}
```

Output is:

Original Object: John - Algebra

Cloned Object: John - Algebra

Original Object after it is updated: Dan - Physics

Cloned Object after updating original object: John - Algebra

Well, if you observe here in the "Student" class, you will see only the change in the "clone()" method. Since its a deep copy, you need to create an object of the cloned class. Well if you have references in the Subject class, then you need to implement Cloneable interface in Subject class and override clone method in it and this goes on and on.

There is an alternative way for deep copy.

Yes, there is. You can do deep copy through serialization. What does serialization do? It writes out the whole object graph into a persistant store and read it back when needed, which means you will get a copy of the whole object

graph when you read it back. This is exactly what you want when you deep copy an object. Note, when you deep copy through serialization, you should make sure that all classes in the object's graph are serializable. Let me explain you this alternative way with an example. If you want to know about serialization first, check it out [here](#).

```
public class ColoredCircle implements Serializable
{
    private int x;
    private int y;

    public ColoredCircle(int x, int y){
        this.x = x;
        this.y = y;
    }

    public int getX(){
        return x;
    }

    public void setX(int x){
        this.x = x;
    }

    public int getY(){
        return y;
    }

    public void setY(int y){
        this.y = y;
    }
}

public class DeepCopy
{
    static public void main(String[] args)
    {
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;

        try
        {
            // create original serializable object
            ColoredCircle c1 = new ColoredCircle(100,100);
            // print it
            System.out.println("Original = " + c1);

            ColoredCircle c2 = null;

            // deep copy
            ByteArrayOutputStream bos = new ByteArrayOutputStream();
            oos = new ObjectOutputStream(bos);
            // serialize and pass the object
            oos.writeObject(c1);
            oos.flush();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

```

        ByteArrayInputStream bin =
            new ByteArrayInputStream(bos.toByteArray());
        ois = new ObjectInputStream(bin);
        // return the new object
        c2 = ois.readObject();

        // verify it is the same
        System.out.println("Copied  = " + c2);
        // change the original object's contents
        c1.setX(200);
        c1.setY(200);
        // see what is in each one now
        System.out.println("Original = " + c1);
        System.out.println("Copied  = " + c2);
    }
    catch(Exception e)
    {
        System.out.println("Exception in main = " + e);
    }
    finally
    {
        oos.close();
        ois.close();
    }
}
}

```

The output is:

```

Original = x=100,y=100
Copied  = x=100,y=100
Original = x=200,y=200
Copied  = x=100,y=100

```

All you need to do here is:

- Ensure that all classes in the object's graph are serializable.
- Create input and output streams.
- Use the input and output streams to create object input and object output streams.
- Pass the object that you want to copy to the object output stream.
- Read the new object from the object input stream and cast it back to the class of the object you sent.

In this example, I have created a ColoredCircle object, c1 and then serialized it (write it out to `ByteArrayOutputStream`). Then I deserialized the serialized object and saved it in c2. Later I modified the original object, c1. Then if you see the result, c1 is different from c2. c2 is deep copy of first version of c1. So its just a copy and not a reference. Now any modifications to c1 wont affect c2, the deep copy of first version of c1.

Well this approach has got its own limitations and issues:

As you cannot serialize a transient variable, using this approach you cannot copy the transient variables. Another issue is dealing with the case of a class whose object's instances within a virtual machine must be controlled. This is a special case of the Singleton pattern, in which a class has only one object within a VM. As discussed above, when you serialize an object, you create a totally new object that will not be unique. To get around this default behavior you can use the `readResolve()` method to force the stream to return an appropriate object rather than the one that was serialized. In this particular case, the appropriate object is the same one that was serialized.

Next one is the performance issue. Creating a socket, serializing an object, passing it through the socket, and then deserializing it is slow compared to calling methods in existing objects. I say, there will be vast difference in the performance. If your code is performance critical, I suggest don't go for this approach. It takes almost 100 times more time to deep copy the object than the way you do by implementing Clonable interface.

When to do shallow copy and deep copy?

It's very simple that if the object has only primitive fields, then obviously you will go for shallow copy but if the object has references to other objects, then based on the requirement, shallow copy or deep copy should be chosen. What I mean here is, if the references are not modified anytime, then there is no point in going for deep copy. You can just opt shallow copy. But if the references are modified often, then you need to go for deep copy. Again there is no hard and fast rule, it all depends on the requirement.

Finally let's have a word about rarely used option - Lazy copy

A lazy copy is a combination of both shallow copy and deep copy. When initially copying an object, a (fast) shallow copy is used. A counter is also used to track how many objects share the data. When the program wants to modify the original object, it can determine if the data is shared (by examining the counter) and can do a deep copy at that time if necessary.

Lazy copy looks to the outside just as a deep copy but takes advantage of the speed of a shallow copy whenever possible. It can be used when the references in the original object are not modified often. The downside are rather high but constant base costs because of the counter. Also, in certain situations, circular references can also cause problems.

<http://javarevisited.blogspot.in/2012/05/how-to-use-threadlocal-in-java-benefits.html>

ThreadLocal in Java is another way to achieve thread-safety apart from writing immutable classes. If you have been writing multi-threaded or concurrent code in Java then you must be familiar with cost of synchronization or locking which can greatly affect Scalability of application, but there is no choice other than synchronize if you are sharing objects between multiple threads. ThreadLocal in Java is a different way to achieve thread-safety, it doesn't address synchronization requirement, instead it eliminates sharing by providing explicitly copy of Object to each thread. Since Object is no more shared there is no requirement of Synchronization which can improve scalability and performance of application. In this Java ThreadLocal tutorial we will see important points about ThreadLocal in Java, when to use ThreadLocal in Java and a simple Example of ThreadLocal in Java program.

When to use ThreadLocal in Java

Many Java Programmer question where to use ThreadLocal in Java and some even argue benefit of ThreadLocal variable, but ThreadLocal has many genuine use cases and that's why its added in to standard Java Platform Library. I agree though until you are not in concurrent programming, you will rarely use ThreadLocal. below are some well know usage of ThreadLocal class in Java:

- 1) ThreadLocal are fantastic to implement Per Thread Singleton classes or per thread context information like transaction id.

- 2) You can wrap any non Thread Safe object in ThreadLocal and suddenly its uses becomes Thread-safe, as its only being used by Thread Safe. One of the classic example of ThreadLocal is sharing SimpleDateFormat. Since SimpleDateFormat is not thread safe, having a global formatter may not work but having per Thread formatter will certainly work.

- 3) ThreadLocal provides another way to extend Thread. If you want to preserve or carry information from one method call to another you can carry it by using ThreadLocal. This can provide immense flexibility as you don't need to modify any method.

On basic level ThreadLocal provides Thread Confinement which is extension of local variable. while local variable only accessible on block they are declared, ThreadLocal are visible only in Single Thread. No two Thread can see each others ThreadLocal variable. Real Life example of ThreadLocal are in J2EE application servers which uses java ThreadLocal variable to keep track of transaction and security Context. It makes lot of sense to share heavy object like Database Connection as ThreadLocal in order to avoid excessive creation and cost of locking in case of sharing global instance.

Java ThreadLocal Example – Code

```
import java.io.IOException;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 *
 * @author
 */
public class ThreadLocalTest {

    public static void main(String args[]) throws IOException {
        Thread t1 = new Thread(new Task());
        Thread t2 = new Thread( new Task());

        t1.start();
        t2.start();

    }

    /**
     * Thread safe format method because every thread will use its own DateFormat
     */
    public static String threadSafeFormat(Date date){
        DateFormat formatter = PerThreadFormatter.getDateFormatter();
        return formatter.format(date);
    }

}

/**
 * Thread Safe implementation of SimpleDateFormat
 * Each Thread will get its own instance of SimpleDateFormat which will not be shared
 * between other threads. *
 */
class PerThreadFormatter {

    private static final ThreadLocal<SimpleDateFormat> dateFormatHolder = new
ThreadLocal<SimpleDateFormat>() {

        /**
         * initialValue() is called
         */
        @Override
        protected SimpleDateFormat initialValue() {
            System.out.println("Creating SimpleDateFormat for Thread : " +
Thread.currentThread().getName());
            return new SimpleDateFormat("dd/MM/yyyy");
        }
    };
};
```



```

    /*
     * Every time there is a call for DateFormat, ThreadLocal will return calling
     * Thread's copy of SimpleDateFormat
     */
    public static DateFormat getDateFormatter() {
        return dateFormatHolder.get();
    }
}

class Task implements Runnable{

    @Override
    public void run() {
        for(int i=0; i<2; i++){
            System.out.println("Thread: " + Thread.currentThread().getName() + " Formatted
Date: " + ThreadLocalTest.threadSafeFormat(new Date()) );
        }
    }
}

```

Output:

```

Creating SimpleDateFormat for Thread : Thread-0
Creating SimpleDateFormat for Thread : Thread-1
Thread: Thread-1 Formatted Date: 30/05/2012
Thread: Thread-1 Formatted Date: 30/05/2012
Thread: Thread-0 Formatted Date: 30/05/2012
Thread: Thread-0 Formatted Date: 30/05/2012

```

If you look the output of above program than you will find that when different thread calls getFormatter() method of ThreadLocal class than its call its initialValue() method which creates exclusive instance of SimpleDateFormat for that Thread. Since SimpleDateFormat is not shared between thread and essentially local to the thread which creates its own threadSafeFormat() method is completely thread-safe.

Important points on Java ThreadLocal Class

1. ThreadLocal in Java is introduced on JDK 1.2 but it later generalized in JDK 1.4 to introduce type safety on ThreadLocal variable.
2. ThreadLocal can be associated with Thread scope, all the code which is executed by Thread has access to ThreadLocal variables but two threads can not see each other's ThreadLocal variable.
3. Each thread holds an exclusive copy of ThreadLocal variable which becomes eligible to Garbage collection after thread finished or died, normally or due to any Exception, Given those ThreadLocal variable doesn't have any other live references.
4. ThreadLocal variables in Java are generally private static fields in Classes and maintain its state inside Thread.

We saw how ThreadLocal in Java opens another avenue for thread-safety. Though concept of thread-safety by confining object to Thread is there from JDK 1.0 and many programmer has their own custom ThreadLocal classes, having ThreadLocal in Java API makes it a lot more easy and standard. Think about ThreadLocal variable while designing concurrency in your application. Don't misunderstand that ThreadLocal is alternative of Synchronization, it all depends upon design. If design allows each thread to have their own copy of object than ThreadLocal is there to use.

What is a thread-local variable?

A thread-local variable effectively provides a separate copy of its value for each thread that uses it. Each thread can see only the value associated with that thread, and is unaware that other threads may be using or modifying their own copies.

Because thread-local variables are implemented through a class, rather than as part of the Java language itself, the syntax for using thread-local variables is a bit more clumsy than for language dialects where thread-local variables are built in. To create a thread-local variable, you instantiate an object of class `ThreadLocal`. The `ThreadLocal` class behaves much like the various `Reference` classes in `java.lang.ref`; it acts as an indirect handle for storing or retrieving a value. Listing 1 shows the `ThreadLocal` interface.

Listing 1. The `ThreadLocal` interface

```
public class ThreadLocal {  
  
    public Object get();  
  
    public void set(Object newValue);  
  
    public Object initialValue();  
  
}
```

The `get()` accessor retrieves the current thread's value of the variable; the `set()` accessor modifies the current thread's value. The `initialValue()` method is an optional method that lets you set the initial value of the variable if it has not yet been used in this thread; it allows for a form of lazy initialization. How `ThreadLocal` behaves is best illustrated by an example implementation. Listing 2 shows one way to implement `ThreadLocal`. It isn't a particularly good implementation (although it is quite similar to the initial implementation), as it would likely perform poorly, but it illustrates clearly how `ThreadLocal` behaves.

Using `ThreadLocal` to implement a per-thread Singleton

Thread-local variables are commonly used to render stateful Singleton or shared objects thread-safe, either by encapsulating the entire unsafe object in a `ThreadLocal` or by encapsulating the object's thread-specific state in a `ThreadLocal`. For example, in an application that is tightly tied to a database, many methods may need to access the database. It could be inconvenient to include a `Connection` as an argument to every method in the system -- a sloppier, but significantly more convenient technique would be to access the connection with a Singleton. However, multiple threads cannot safely share a JDBC `Connection`. By using a `ThreadLocal` in our Singleton, as shown in Listing 3, we can allow any class in our program to easily acquire a reference to a per-thread `Connection`. In this way, we can think of a `ThreadLocal` as allowing us to create a per-thread-singleton.

Listing 3. Storing a JDBC `Connection` in a per-thread Singleton

```

public class ConnectionDispenser {
    private static class ThreadLocalConnection extends ThreadLocal {
        public Object initialValue() {
            return DriverManager.getConnection(ConfigurationSingleton.getDbUrl());
        }
    }

    private static ThreadLocalConnection conn = new ThreadLocalConnection();

    public static Connection getConnection() {
        return (Connection) conn.get();
    }
}

```

ThreadLocal's less thread-safe cousin, InheritableThreadLocal

The ThreadLocal class has a relative, InheritableThreadLocal, which functions in a similar manner, but is suitable for an entirely different sort of application. When a thread is created, if it holds values for any InheritableThreadLocal objects, these values are automatically passed on to the child process as well. If a child process calls get() on an InheritableThreadLocal, it sees the same object as the parent would. To preserve thread-safety, you should use InheritableThreadLocal only for immutable objects (objects whose state will not ever be changed once created), because the object is shared between multiple threads. InheritableThreadLocal is useful for passing data from a parent thread to a child thread, such as a user id, or a transaction id, but not for stateful objects like JDBC Connections.

The benefits of ThreadLocal

ThreadLocal offers a number of benefits. It is often the easiest way to render a stateful class thread-safe, or to encapsulate non-thread-safe classes so that they can safely be used in multithreaded environments. Using ThreadLocal allows us to bypass the complexity of determining when to synchronize in order to achieve thread-safety, and it improves scalability because it doesn't require any synchronization. In addition to simplicity, using ThreadLocal to store a per-thread-singleton or per-thread context information has a valuable documentation perk -- by using a ThreadLocal, it's clear that the object stored in the ThreadLocal is not shared between threads, simplifying the task of determining whether a class is thread-safe or not.

<http://veerasundar.com/blog/2010/11/java-thread-local-how-to-use-and-code-sample/>

Thread Local is an interesting and useful concept, yet most of the Java developers are not aware of how to use that. In this post, I'll explain what is Thread Local and when to use it, with an example code. Since it'll be little tough to understand this concept at first, I'll keep the explanation as simple as possible (corollary: you shouldn't use this code as it is in a production environment. Grasp the concept and improve upon it!)

What is Thread Local?

Thread Local can be considered as a scope of access, like a request scope or session scope. It's a thread scope. You can set any object in Thread Local and this object will be global and local to the specific thread which is accessing this object. Global and local!!? Let me explain:

Values stored in Thread Local are global to the thread, meaning that they can be accessed from anywhere inside that thread. If a thread calls methods from several classes, then all the methods can see the Thread Local variable set by other methods (because they are executing in same thread). The value need not be passed explicitly. It's like how you use global variables.

Values stored in Thread Local are local to the thread, meaning that each thread will have it's own Thread Local variable. One thread can not access/modify other thread's Thread Local variables.

Well, that's the concept of Thread Local. I hope you understood it (if not, please leave a comment).

When to use Thread Local?

We saw what is thread local in the above section. Now let's talk about the use cases. i.e. when you'll be needing something like Thread Local.

I can point out one use case where I used thread local. Consider you have a Servlet which calls some business methods. You have a requirement to generate a unique transaction id for each and every request this servlet process and you need to pass this transaction id to the business methods, for logging purpose. One solution would be passing this transaction id as a parameter to all the business methods. But this is not a good solution as the code is redundant and unnecessary.

To solve that, you can use Thread Local. You can generate a transaction id (either in servlet or better in a filter) and set it in the Thread Local. After this, what ever the business method, that this servlet calls, can access the transaction id from the thread local.

This servlet might be servicing more that one request at a time. Since each request is processed in separate thread, the transaction id will be unique to each thread (local) and will be accessible from all over the thread's execution (global).

Got it!?

How to use Thread Local?

Java provides an ThreadLocal object using which you can set/get thread scoped variables. Below is a code example demonstrating what I'd explained above.

Lets first have the Context.java file which will hold the transactionId field.

```
package com.veerasundar;

public class Context {

    private String transactionId = null;

    /* getters and setters here */

}
```

Now create the MyThreadLocal.java file which will act as a container to hold our context object.

```

package com.veerasundar;

/**
 * this class acts as a container to our thread local variables.
 * @author vsundar
 */
public class MyThreadLocal {

    public static final ThreadLocal userThreadLocal = new ThreadLocal();

    public static void set(Context user) {
        userThreadLocal.set(user);
    }

    public static void unset() {
        userThreadLocal.remove();
    }

    public static Context get() {
        return userThreadLocal.get();
    }
}

```

In the above code, you are creating a ThreadLocal object as a static field which can be used by rest of the code to set/get thread local variables.

Let's create our main class file which will generate and set the transaction ID in thread local and then call the business method.

```

package com.veerasundar;

public class ThreadLocalDemo extends Thread {

    public static void main(String args[]) {

        Thread threadOne = new ThreadLocalDemo();
        threadOne.start();

        Thread threadTwo = new ThreadLocalDemo();
        threadTwo.start();
    }

    @Override
    public void run() {
        // sample code to simulate transaction id
        Context context = new Context();
        context.setTransactionId(getName());

        // set the context object in thread local to access it somewhere else
        MyThreadLocal.set(context);

        /* note that we are not explicitly passing the transaction id */
        new BusinessService().businessMethod();
        MyThreadLocal.unset();
    }
}

```

Finally, here's the code for the BusinessService.java which will read from thread local and use the value.

```
public class BusinessService {  
    public void businessMethod() {  
        // get the context from thread local  
        Context context = MyThreadLocal.get();  
        System.out.println(context.getTransactionId());  
    }  
}
```

When you run the ThreadLocalDemo file, you'll get the below output:

Thread-0

Thread-1

As you might see, even though we are not explicitly passing the transaction id, the value can be accessed from the business method and printed on the console. Adding to it, the transaction ID differs for each thread (0 and 1).

<http://docs.oracle.com/javase/6/docs/api/java/lang/ThreadLocal.html>

This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

For example, the class below generates unique identifiers local to each thread. A thread's id is assigned the first time it invokes UniqueThreadIdGenerator.getCurrentThreadId() and remains unchanged on subsequent calls.

```
import java.util.concurrent.atomic.AtomicInteger;

public class UniqueThreadIdGenerator {

    private static final AtomicInteger uniqueId = new AtomicInteger(0);

    private static final ThreadLocal < Integer > uniqueNum =
        new ThreadLocal < Integer > () {
            @Override protected Integer initialValue() {
                return uniqueId.getAndIncrement();
            }
        };

    public static int getCurrentThreadId() {
        return uniqueId.get();
    }
} // UniqueThreadIdGenerator
```

<http://stackoverflow.com/questions/817856/when-and-how-should-i-use-a-threadlocal-variable>

ThreadLocal Usage

One possible (and common) use is when you have some object that is not thread-safe, but you want to avoid synchronizing access to that object (I'm looking at you, SimpleDateFormat). Instead, give each thread its own instance of the object.

Another alternative to synchronization or threadlocal is to make the variable a local variable. Local vars are always thread safe. I assume that it is bad practice to make DateFormats local because they are expensive to create.

<http://thecafetechno.com/tutorials/interview-questions/what-is-threadlocal-class-what-are-threadlocal-variables/>

ThreadLocal class allows you to put local data on a thread, so that every module running in the thread can access it. ThreadLocal class was introduced in JDK 1.2, but hasn't been used much, maybe because of a first, rather poor implementation, performance-wise.

According to Java Docs:

This class provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its get or set method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread (e.g., a user ID or Transaction ID).

A thread-local variable effectively provides a separate copy of its value for each thread that uses it.

Thread-local instances are typically private static fields in classes that wish to associate state with a thread.

In case when multiple threads access a thread-local instance, separate copy of thread-local variable is maintained for each thread.

ThreadLocal Class contains these methods:

MethodPurpose

Object get()	Returns the value in the current thread's copy of this thread-local variable.
set(Object)	Sets the current thread's copy of this thread-local variable to the specified value.
Object initialValue()	Returns the current thread's "initial value" for this thread-local variable.
remove()	Was introduced in 1.5 - Removes the current thread's value for this thread-local variable.

An example: A servlet is executed in a thread, but since many users may use the same servlet at the same time, many threads will be running the same servlet code concurrently. If the servlet uses a ThreadLocal object, it can hold data local to each thread. The user ID is a good example of what could be stored in the ThreadLocal object. I like to think of this object as a hash map where a kind of thread ID is used as the key.

```
package com.thecafetechno;
public class ThreadLocalManager {
    public static final ThreadLocal myThreadLocal = new ThreadLocal();
}

package com.thecafetechno;
public class TestThreadLocal implements Runnable {
    private String value;
    private long delayTime;
    private long sleepTime;

    public TestThreadLocal(String value, long delayTime, long sleepTime) {
        this.value = value;
        this.delayTime = delayTime;
        this.sleepTime = sleepTime;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(this.delayTime);
            System.out.println "[" + this + "] is setting myThreadLocal ["
                + ThreadLocalManager.myThreadLocal + "] the value : "
                + this.value);
            ThreadLocalManager.myThreadLocal.set(this.value);
            Thread.sleep(this.sleepTime);
            System.out.println "[" + this + "] is accessing myThreadLocal ["
                + ThreadLocalManager.myThreadLocal + "] value : "
                + ThreadLocalManager.myThreadLocal.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        TestThreadLocal test1 = new TestThreadLocal("Value1", 0, 300);
        System.out.println("Creating test1 : " + test1);
        TestThreadLocal test2 = new TestThreadLocal("Value2", 100, 600);
        System.out.println("Creating test2 : " + test2);
        Thread t1 = new Thread(test1);
        Thread t2 = new Thread(test2);

        t1.start();
        t2.start();
    }
}
```

Although the execution of both threads is tangled, the output message clearly shows that the ThreadLocal value is completely isolated between threads t1 & t2.

Annotations in Java

While creating an annotation in java , two things are important, one is **@Target** and **@Retention**. An example is given below.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Food {
    FoodType type();
}
```

@Target : As per javadocs, Indicates the kinds of program element to which an annotation type is applicable.

The following constants are used for @Target ElementType.

- **ANNOTATION_TYPE** - Annotation type declaration
- **CONSTRUCTOR** - Constructor declaration
- **FIELD** - Field declaration (includes enum constants)
- **LOCAL_VARIABLE** - Local variable declaration
- **METHOD** - Method declaration
- **PACKAGE** - Package declaration
- **PARAMETER** - Parameter declaration
- **TYPE** - Class, interface (including annotation type), or enum declaration

Let's say the annotation to which you specify the ElementType is called YourAnnotation:

ANNOTATION_TYPE - Annotation type declaration. **Note:** This goes on other annotations

```
@YourAnnotation
```

```
public @interface AnotherAnnotation {..}
```

CONSTRUCTOR - Constructor declaration

```
public class SomeClass {
    @YourAnnotation
    public SomeClass() {..}
}
```

FIELD - Field declaration (includes enum constants)

```
@YourAnnotation
```

```
private String someField;
```

LOCAL_VARIABLE - Local variable declaration. **Note:** This can't be read at runtime, so it is used only for compile-time things, like the @SuppressWarnings annotation.

```
public void someMethod() {
    @YourAnnotation int a = 0;
}
```

METHOD - Method declaration

```
@YourAnnotation
```

```
public void someMethod() {..}
```

PACKAGE - Package declaration. **Note:** This can be used only in package-info.java.

```
@YourAnnotation
```

```
package org.yourcompany.somepackage;
```

PARAMETER - Parameter declaration

```
public void someMethod(@YourAnnotation param) {..}
```

TYPE - Class, interface (including annotation type), or enum declaration

```
@YourAnnotation
```

```
public class SomeClass {..}
```

You can specify multiple `ElementType`s for a given annotation. E.g.:

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})
```

Retention Policy

As per javadocs, Indicates how long annotations with the annotated type are to be retained. If no Retention annotation is present on an annotation type declaration, the retention policy defaults to `RetentionPolicy.CLASS`.

- **`RetentionPolicy.SOURCE`**: Discard during the compile. These annotations don't make any sense after the compile has completed, so they aren't written to the bytecode.
Example: `@Override`, `@SuppressWarnings`
- **`RetentionPolicy.CLASS`**: Discard during class load. Useful when doing bytecode-level post-processing. Somewhat surprisingly, this is the default.
- **`RetentionPolicy.RUNTIME`**: Do not discard. The annotation should be available for reflection at runtime.
Example: `@Deprecated`

WHAT IS RETENTION POLICY IN JAVA ANNOTATIONS?

Description:

- A retention policy determines at what point annotation should be discarded.
- Java defined 3 types of retention policies through `java.lang.annotation.RetentionPolicy` enumeration. It has `SOURCE`, `CLASS` and `RUNTIME`.
- Annotation with retention policy `SOURCE` will be retained only with source code, and discarded during compile time.
- Annotation with retention policy `CLASS` will be retained till compiling the code, and discarded during runtime.
- Annotation with retention policy `RUNTIME` will be available to the JVM through runtime.
- The retention policy will be specified by using java built-in annotation `@Retention`, and we have to pass the retention policy type.
- **The default retention policy type is `CLASS`.**

An Example of custom annotation is given below.

Food.java

```
package com.ddlab.rnd;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
//@Target(value={ElementType.TYPE})//You can provide multiple type
@Target(ElementType.TYPE)
public @interface Food {
    FoodType type();
}
```

FoodType.java

```
package com.ddlab.rnd;
public enum FoodType {
    VEG, NON_VEG
}
```

Cow.java

```
package com.ddlab.rnd;
@Food(type=FoodType.VEG)
public class Cow {
    @Override
    public String toString() {
        return "Cow";
    }
}
```

Lion.java

```
package com.ddlab.rnd;
@Food(type = FoodType.NON_VEG )
public class Lion {

    @Override
    public String toString() {
        return "Lion";
    }
}
```

TestAnnotation.java

```
package com.ddlab.rnd;
public class TestAnnotations {

    public static void processObject( Object obj ) {

        boolean flag = obj.getClass().isAnnotationPresent(Food.class);
        if( flag ) {
            Food food = obj.getClass().getAnnotation(Food.class);
            FoodType foodType = food.type();
            if( foodType == FoodType.VEG ) {
                System.out.println(obj+" is herbivorous ..");
            }
            else {
                System.out.println(obj+" is carnivorous ..");
            }
        }
    }

    public static void main(String[] args) {
        Cow cow = new Cow();
        Lion lion = new Lion();
        processObject(cow);
        processObject(lion);
    }
}
```

Inner class in java

<http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/inner-classes.html>

There are four types of inner classes: [member](#), [static member](#), [local](#), and [anonymous](#).

A member class is defined at the top level of the [class](#). It may have the same [access](#) modifiers as variables (public, protected, package, static, final), and is accessed in much the same way as [variables](#) of that class.

```
public class OuterClass {
    int outerVariable = 100;

    class MemberClass {
        int innerVariable = 20;

        int getSum(int parameter) {
            return innerVariable + outerVariable + parameter;
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        MemberClass inner = outer.new MemberClass();
        System.out.println(inner.getSum(3));
        outer.run();
    }

    void run() {
        MemberClass localInner = new MemberClass();
        System.out.println(localInner.getSum(5));
    }
}
```

A **static member class** is defined like a member class, but with the keyword [static](#). Despite its position inside another class, a static member class is actually an "outer" class--it has no special access to names in its containing class. To refer to the static inner class from a class outside the containing class, use the syntax **OuterClassName.InnerClassName**. A static member class may contain static fields and methods.

```
public class OuterClass {
    int outerVariable = 100;
    static int staticOuterVariable = 200;

    static class StaticMemberClass {
        int innerVariable = 20;

        int getSum(int parameter) {
            // Cannot access outerVariable here
            return innerVariable + staticOuterVariable + parameter;
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        StaticMemberClass inner = new StaticMemberClass();
        System.out.println(inner.getSum(3));
        outer.run();
    }

    void run() {
        StaticMemberClass localInner = new StaticMemberClass();
        System.out.println(localInner.getSum(5));
    }
}
```

A **local inner class** is defined within a method, and the usual scope rules apply to it. It is only accessible within that method, therefore access restrictions (**public**, **protected**, **package**) do not apply. However, because objects (and their methods) created from this class may persist after the method returns, a local inner class *may not refer to parameters or non-**final** local variables of the method.*

```
public class OuterClass {
    int outerVariable = 10000;
    static int staticOuterVariable = 2000;

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        System.out.println(outer.run());
    }

    Object run() {
        int localVariable = 666;
        final int finalLocalVariable = 300;

        class LocalClass {
            int innerVariable = 40;

            int getSum(int parameter) {
                // Cannot access localVariable here
                return outerVariable + staticOuterVariable +
                    finalLocalVariable + innerVariable + parameter;
            }
        }
        LocalClass local = new LocalClass();
        System.out.println(local.getSum(5));
        return local;
    }
}
```

An **anonymous inner class** is one that is declared and used to create one object (typically as a parameter to a method), all within a single statement.

An anonymous inner class may extend a class:

```
new SuperClass(parameters){ class body }
```

Here, **SuperClass** is not the name of the class being defined, but rather the name of the class being extended. The **parameters** are the parameters to the constructor for that superclass.

An anonymous inner class may implement an interface:

```
new Interface(){ class body }
```

Because anonymous inner classes are almost always used as event listeners, the example below uses an anonymous inner class as a button listener.

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

public class OuterClass extends JFrame {

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        JButton button = new JButton("Don't click me!");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                System.out.println("Ouch!");
            }
        });
        outer.add(button);
        outer.pack();
        outer.setVisible(true);
    }
}

```

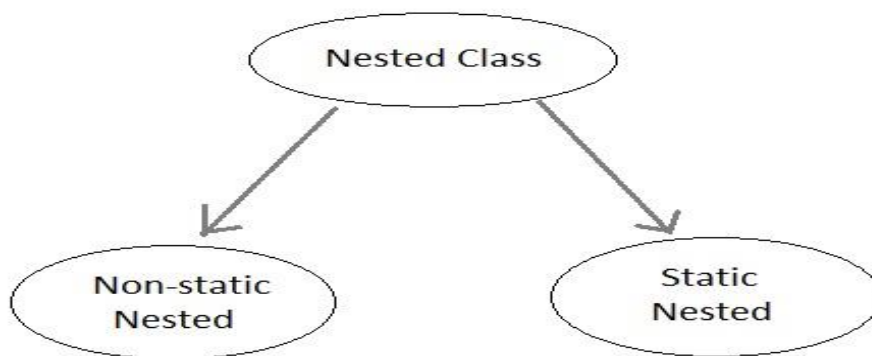
STYLE

Because anonymous inner classes occur within a method, they break up the flow and add several lines to the method. Consequently, the actual code within an anonymous inner class should be kept very short--usually a single method call.

<http://www.studytonight.com/java/nested-classes.php>

Nested Class

A class within another class is known as Nested class. The scope of the nested is bounded by the scope of its enclosing class.



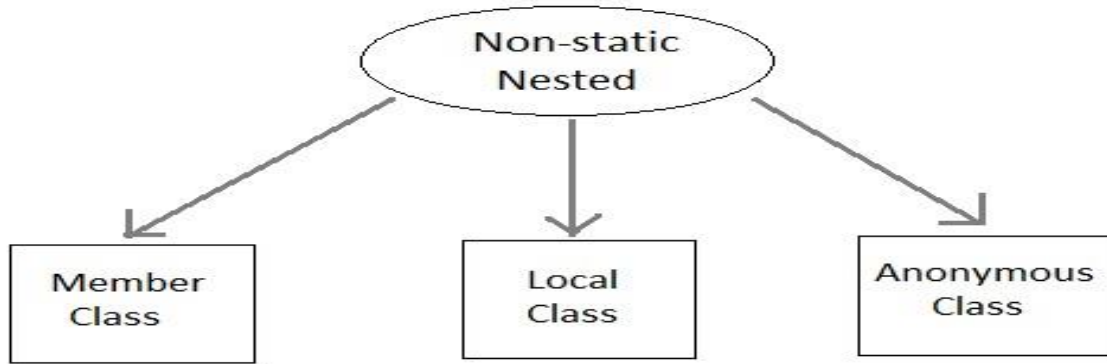
Static Nested Class

A static nested class is the one that has **static** modifier applied. Because it is static it cannot refer to non-static members of its enclosing class directly. Because of this restriction static nested class is seldom used.

Non-static Nested class

Non-static Nested class is most important type of nested class. It is also known as **Inner** class. It has access to all variables and methods of **Outer** class and may refer to them directly. But the reverse is not true, that is, **Outer** class cannot directly access members of **Inner** class.

One more important thing to notice about an **Inner** class is that it can be created only within the scope of **Outer** class. Java compiler generates an error if any code outside **Outer** class attempts to instantiate **Inner** class.



Example of Inner class

```
class Outer
{
    public void display()
    {
        Inner in=new Inner();
        in.show();
    }

    class Inner
    {
        public void show()
        {
            System.out.println("Inside inner");
        }
    }
}

class Test
{
    public static void main(String[] args)
    {
        Outer ot=new Outer();
        ot.display();
    }
}
```

Output:

Inside inner

Example of Inner class inside a method

```
class Outer {
    int count;
    public void display() {
        for(int i=0;i<5;i++)
        {
            class Inner        //Inner class defined inside for loop
            {
                public void show()
                {
                    System.out.println("Inside inner "+(count++));
                }
            }
            Inner in=new Inner();
            in.show();
        }
    }
}

class Test {
    public static void main(String[] args) {
        Outer ot=new Outer();
        ot.display();
    }
}
```

Output:

Inside inner 0
Inside inner 1
Inside inner 2
Inside inner 3
Inside inner 4

Example of Inner class instantiated outside Outer class

```
class Outer {
    int count;
    public void display() {
        Inner in=new Inner();
        in.show();
    }
    class Inner {
        public void show() {
            System.out.println("Inside inner "+(++count));
        }
    }
}

class Test {
    public static void main(String[] args)
    {
        Outer ot=new Outer();
        Outer.Inner in= ot.new Inner();
        in.show();
    }
}
```

Output

Inside inner 1

Anonymous class

A class without any name is called Anonymous class.

```
interface Animal
{
    void type();
}
public class ATest {
    public static void main(String args[])
    {
        Animal an = new Animal(){           //Anonymous class created
            public void type()
            {
                System.out.println("Anonymous animal");
            }
        };
        an.type();
    }
}
```

Output

Anonymous animal

Here a class is created which implements **Animal** interface and its name will be decided by the compiler. This anonymous class will provide implementation of **type()** method.

<http://stackoverflow.com/questions/70324/java-inner-class-and-static-nested-class>

Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are simply called static nested classes. Non-static nested classes are called inner classes.

Static nested classes are accessed using the enclosing class name:

`OuterClass.StaticNestedClass`

For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```

Objects that are instances of an inner class exist within an instance of the outer class. Consider the following classes:

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance. To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

see: [Java Tutorial - Nested Classes](#)

For completeness note that there is also such a thing as an [inner class without an enclosing instance](#):

```
class A {
    int t() { return 1; }
    static A a = new A() { int t() { return 2; } };
}
```

Here, `new A() { ... }` is an *inner class defined in a static context* and does not have an enclosing instance.

There are three reasons you might create a nested class:

- organization: sometimes it seems most sensible to sort a class into the namespace of another class, especially when it won't be used in any other context
- access: nested classes have special access to the variables/fields of their containing classes (precisely which variables/fields depends on the kind of nested class, whether inner or static).
- convenience: having to create a new file for every new type is bothersome, again, especially when the type will only be used in one context

There are **four kinds of nested class in Java**. In brief, they are:

- **static class**: declared as a static member of another class
- **inner class**: declared as an instance member of another class
- **local inner class**: declared inside an instance method of another class
- **anonymous inner class**: like a local inner class, but written as an expression which returns a one-off object

In more detail:

static classes

Static classes are the easiest kind to understand because they have nothing to do with instances of the containing class.

A static class is a class declared as a static member of another class. Just like other static members, such a class is really just a hanger on that uses the containing class as its namespace, e.g. the class `Goat` declared as a static member of class `Rhino` in the package `pizza` is known by the name `pizza.Rhino.Goat`.

```
package pizza;

public class Rhino {

    ...

    public static class Goat {

        ...

    }

}
```

Frankly, static classes are a pretty worthless feature because classes are already divided into namespaces by packages. The only real conceivable reason to create a static class is that such a class has access to its containing class's private static members, but I find this to be a pretty lame justification for the static class feature to exist.

inner classes

An inner class is a class declared as a non-static member of another class:

```
package pizza;

public class Rhino {

    public class Goat {

        ...

    }

    private void jerry() {
        Goat g = new Goat();
    }

}
```

Like with a static class, the inner class is known as qualified by its containing class name, `pizza.Rhino.Goat`, but inside the containing class, it can be known by its simple name. However, every instance of an inner class

is tied to a particular instance of its containing class: above, the *Goat* created in *jerry*, is implicitly tied to the *Rhino* instance *this* in *jerry*. Otherwise, we make the associated *Rhino* instance explicit when we instantiate *Goat*:

```
Rhino rhino = new Rhino();
Rhino.Goat goat = rhino.new Goat();
```

(Notice you refer to the inner type as just *Goat* in the weird *new* syntax: Java infers the containing type from the *rhino* part. And, yes *new rhino.Goat()* would have made more sense to me too.)

So what does this gain us? Well, the inner class instance has access to the instance members of the containing class instance. These enclosing instance members are referred to inside the inner class *via* just their simple names, not *via this* (*this* in the inner class refers to the inner class instance, not the associated containing class instance):

```
public class Rhino {

    private String barry;

    public class Goat {
        public void colin() {
            System.out.println(barry);
        }
    }
}
```

In the inner class, you can refer to *this* of the containing class as *Rhino.this*, and you can use *this* to refer to its members, e.g. *Rhino.this.barry*.

local inner classes

A local inner class is a class declared in the body of a method. Such a class is only known within its containing method, so it can only be instantiated and have its members accessed within its containing method. The gain is that a local inner class instance is tied to and can access the final local variables of its containing method. When the instance uses a final local of its containing method, the variable retains the value it held at the time of the instance's creation, even if the variable has gone out of scope (this is effectively Java's crude, limited version of closures).

Because a local inner class is neither the member of a class or package, it is not declared with an access level. (Be clear, however, that its own members have access levels like in a normal class.)

If a local inner class is declared in an instance method, an instantiation of the inner class is tied to the instance held by the containing method's *this* at the time of the instance's creation, and so the containing class's instance members are accessible like in an instance inner class. A local inner class is instantiated simply *via* its name, e.g. local inner class *Cat* is instantiated as *new Cat()*, not *new this.Cat()* as you might expect.

anonymous inner classes

An anonymous inner class is a syntactically convenient way of writing a local inner class. Most commonly, a local inner class is instantiated at most just once each time its containing method is run. It would be nice, then, if we could combine the local inner class definition and its single instantiation into one convenient syntax form, and it would also be nice if we didn't have to think up a name for the class (the fewer unhelpful names your code contains, the better). An anonymous inner class allows both these things:

```
new *ParentClassName*(*constructorArgs*) {*members*}
```

This is an expression returning a new instance of an unnamed class which extends *ParentClassName*. You cannot supply your own constructor; rather, one is implicitly supplied which simply calls the super constructor, so the arguments supplied must fit the super constructor. (If the parent contains multiple constructors, the "simplest" one is called, "simplest" as determined by a rather complex set of rules not worth bothering to learn in detail--just pay attention to what NetBeans or Eclipse tell you.)

Alternatively, you can specify an interface to implement:

```
new *InterfaceName*() {*members*}
```

Such a declaration creates a new instance of an unnamed class which extends `Object` and implements *InterfaceName*. Again, you cannot supply your own constructor; in this case, Java implicitly supplies a no-arg, do-nothing constructor (so there will never be constructor arguments in this case). Even though you can't give an anonymous inner class a constructor, you can still do any setup you want using an initializer block (a `{}` block placed outside any method).

Be clear that an anonymous inner class is simply a less flexible way of creating a local inner class with one instance. If you want a local inner class which implements multiple interfaces or which implements interfaces while extending some class other than *Object* or which specifies its own constructor, you're stuck creating a regular named local inner class.

First to get the terms right:

- A nested class is a class which is contained in another class at the source code level.
- It is static if you declare it with the **static** modifier.
- A non-static nested class is called inner class. (I stay with non-static nested class.)

Martin's answer is right so far. However, the actual question is: What is the purpose of declaring a nested class static or not?

You use **static nested classes** if you just want to keep your classes together if they belong topically together or if the nested class is exclusively used in the enclosing class. There is no semantic difference between a static nested class and every other class.

Non-static nested classes are a different beast. Similar to anonymous inner classes, such nested classes are actually closures. That means they capture their surrounding scope and their enclosing instance and make that accessible. Perhaps an example will clarify that. See this stub of a `Container`:

```
public class Container {
    public class Item{
        Object data;
        public Container getContainer(){
            return Container.this;
        }
        public Item(Object data) {
            super();
            this.data = data;
        }
    }

    public static Item create(Object data){
        // does not compile since no instance of Container is available
        return new Item(data);
    }
    public Item createSubItem(Object data){
        // compiles, since 'this' Container is available
        return new Item(data);
    }
}
```

In this case you want to have a reference from a child item to the parent container. Using a non-static nested class, this works without some work. You can access the enclosing instance of `Container` with the syntax `Container.this`.

More hardcore explanations following:

If you look at the Java bytecodes the compiler generates for an (non-static) nested class it might become even clearer:

```
// class version 49.0 (49)
// access flags 33
```

```

public class Container$Item {
    // compiled from: Container.java
    // access flags 1
    public INNERCLASS Container$Item Container Item
    // access flags 0
    Object data
    // access flags 4112
    final Container this$0

    // access flags 1
    public getContainer() : Container
    L0
        LINENUMBER 7 L0
        ALOAD 0: this
        GETFIELD Container$Item.this$0 : Container
        ARETURN
    L1
        LOCALVARIABLE this Container$Item L0 L1 0
        MAXSTACK = 1
        MAXLOCALS = 1
    // access flags 1
    public <init>(Container, Object) : void
    L0
        LINENUMBER 12 L0
        ALOAD 0: this
        ALOAD 1
        PUTFIELD Container$Item.this$0 : Container
    L1
        LINENUMBER 10 L1
        ALOAD 0: this
        INVOKESPECIAL Object.<init>() : void
    L2
        LINENUMBER 11 L2
        ALOAD 0: this
        ALOAD 2: data
        PUTFIELD Container$Item.data : Object
        RETURN
    L3
        LOCALVARIABLE this Container$Item L0 L3 0
        LOCALVARIABLE data Object L0 L3 2
        MAXSTACK = 2
        MAXLOCALS = 3
}

```

As you can see the compiler creates a hidden field `Container this$0`. This is set in the constructor which has an additional parameter of type `Container` to specify the enclosing instance. You can't see this parameter in the source but the compiler implicitly generates it for a nested class.

Martin's example

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

would so be compiled to a call of something like (in bytecodes)

```
new InnerClass(outerObject)
```

For the sake of completeness:

An anonymous class **is** a perfect example of a non-static nested class which just has no name associated with it and can't be referenced later.

There are 4 different types of inner classes you can use in Java. The following gives their name and examples.

1. Static Nested Classes

```
class Outer {
    static class Inner {
        void go() {
            System.out.println("Inner class reference is: " + this);
        }
    }
}

public class Test {
    public static void main(String[] args) {
        Outer.Inner n = new Outer.Inner();
        n.go();
    }
}
```

```
Inner class reference is: Outer$Inner@19e7ce87
```

2. Member Inner Class

Member class is instance-specific. It has access to all methods, fields, and the Outer's this reference.

```
public class Outer {
    private int x = 100;

    public void makeInner(){
        Inner in = new Inner();
        in.seeOuter();
    }

    class Inner{
        public void seeOuter(){
            System.out.println("Outer x is " + x);
            System.out.println("Inner class reference is " + this);
            System.out.println("Outer class reference is " + Outer.this);
        }
    }

    public static void main(String [] args){
        Outer o = new Outer();
        Inner i = o.new Inner();
        i.seeOuter();
    }
}
```

```
Outer x is 100
```

```
Inner class reference is Outer$Inner@4dfd9726
```

```
Outer class reference is Outer@43ce67ca
```

3. Method-Local Inner Classes

```
public class Outer {  
    private String x = "outer";  
  
    public void doStuff() {  
        class MyInner {  
            public void seeOuter() {  
                System.out.println("x is " + x);  
            }  
        }  
  
        MyInner i = new MyInner();  
        i.seeOuter();  
    }  
  
    public static void main(String[] args) {  
        Outer o = new Outer();  
        o.doStuff();  
    }  
}
```

```
x is outer
```

```
public class Outer {  
    private static String x = "static outer";  
  
    public static void doStuff() {  
        class MyInner {  
            public void seeOuter() {  
                System.out.println("x is " + x);  
            }  
        }  
  
        MyInner i = new MyInner();  
        i.seeOuter();  
    }  
  
    public static void main(String[] args) {  
        Outer.doStuff();  
    }  
}
```

```
x is static outer
```

4. Anonymous Inner Classes

This is frequently used when you add an action listener to a widget in a GUI application.

```
button.addActionListener(new ActionListener(){  
    public void actionPerformed(ActionEvent e){  
        comp.setText("Button has been clicked");  
    }  
});
```

instanceof Vs getClass()

<http://stackoverflow.com/questions/4989818/instanceof-vs-getclass>

The reason that the performance of `instanceof` and `getClass() == ...` is different is that they are doing different things.

- `instanceof` tests whether the object reference on the left-hand side (LHS) is an instance of the type on the right-hand side (RHS) *or some subtype*.
- `getClass() == ...` tests whether the types are identical.

So the recommendation is to ignore the performance issue and use the alternative that gives you the answer that you need.

And yes, overuse of either of them is "design smell". If you are not careful, you end up with a design where the addition of new subclasses results in a significant amount of code reworking. In most situations, the preferred approach is to use polymorphism.

<http://stackoverflow.com/questions/596462/any-reason-to-prefer-getclass-over-instanceof-when-generating-equals>

The reason that I favor the `instanceof` approach is that when you use the `getClass` approach, you have the restriction that objects are only equal to other objects of the same class, the same run time type. If you extend a class and add a couple of innocuous methods to it, then check to see whether some object of the subclass is equal to an object of the super class, even if the objects are equal in all important aspects, you will get the surprising answer that they aren't equal. In fact, this violates a strict interpretation of the *Liskov substitution principle*, and can lead to very surprising behavior. In Java, it's particularly important because most of the collections (`HashTable`, etc.) are based on the `equals` method. If you put a member of the super class in a hash table as the key and then look it up using a subclass instance, you won't find it, because they are not equal.

Angelika Langers [Secrets of equals](#) gets into that with a long and detailed discussion for a few common and well-known examples, including by Josh Bloch and Barbara Liskov, discovering a couple of problems in most of them. She also gets into the `instanceof` vs `getClass`. Some quote from it

Conclusions

Having dissected the four arbitrarily chosen examples of implementations of `equals()` , what do we conclude?

First of all: there are two substantially different ways of performing the check for type match in an implementation of `equals()` . A class can allow mixed-type comparison between super- and subclass objects by means of the `instanceof` operator, or a class can treat objects of different type as non-equal by means of the `getClass()` test. The examples above illustrated nicely that implementations of `equals()` using `getClass()` are generally more robust than those implementations using `instanceof` .

The `instanceof` test is correct only for final classes or if at least method `equals()` is final in a superclass. The latter essentially implies that no subclass must extend the superclass's state, but can only add functionality or fields that are irrelevant for the object's state and behavior, such as transient or static fields.

Implementations using the `getClass()` test on the other hand always comply to the `equals()` contract; they are correct and robust. They are, however, semantically very different from implementations that use the `instanceof` test. Implementations using `getClass()` do not allow comparison of sub- with superclass objects, not even when the subclass does not add any fields and would not even want to override `equals()` . Such a "trivial" class extension would for instance be the addition of a debug-print method in a subclass defined for exactly this "trivial" purpose. If the superclass prohibits mixed-type comparison via the `getClass()` check, then the trivial extension would not be comparable to its superclass. Whether or not this is a problem fully depends on the semantics of the class and the purpose of the extension.

Java 7 new features

Binary Literals

In Java SE 7, the integral types (`byte`, `short`, `int`, and `long`) can also be expressed using the binary number system. To specify a binary literal, add the prefix `0b` or `0B` to the number. The following examples show binary literals:

```
// An 8-bit 'byte' value:
byte aByte = (byte)0b00100001;

// A 16-bit 'short' value:
short aShort = (short)0b1010000101000101;

// Some 32-bit 'int' values:
int anInt1 = 0b10100001010001011010000101000101;
int anInt2 = 0b101;
int anInt3 = 0B101; // The B can be upper or lower case.

// A 64-bit 'long' value. Note the "L" suffix:
long aLong = 0b101000010100010110100001010001011010000101000101010000101000101L;
```

Binary literals can make relationships among data more apparent than they would be in hexadecimal or octal. For example, each successive number in the following array is rotated by one bit:

```
public static final int[] phases = {
    0b00110001,
    0b01100010,
    0b11000100,
    0b10001001,
    0b00010011,
    0b00100110,
    0b01001100,
    0b10011000
}
```

Example is given below.

```
int anInt1 = 0b10100001010001011010000101000101;
int anInt2 = 0b101;
int anInt3 = 0B101; // The B can be upper or lower case.
```

```
System.out.println(anInt1); // -1589272251
System.out.println(anInt2); // 5
System.out.println(anInt3); // 5
```

Underscores in Numeric Literals

In Java SE 7 and later, any number of underscore characters (`_`) can appear anywhere between digits in a numerical literal. This feature enables you, for example, to separate groups of digits in numeric literals, which can improve the readability of your code.

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

```
System.out.println(creditCardNumber); // 1234567890123456
System.out.println(pi); // 3.1415
System.out.println(maxLong); // 9223372036854775807
```

Strings in switch Statements

In the JDK 7 release, you can use a `String` object in the expression of a `switch` statement:

```
public String getDayOfWeekWithSwitchStatement(String dayOfWeekArg) {
    String typeOfDay;
    switch (dayOfWeekArg) {
        case "Monday":
            typeOfDay = "Start of work week";
            break;
        case "Tuesday":
        case "Wednesday":
        case "Thursday":
            typeOfDay = "Midweek";
            break;
        case "Friday":
            typeOfDay = "End of work week";
            break;
        case "Saturday":
        case "Sunday":
            typeOfDay = "Weekend";
            break;
        default:
            throw new IllegalArgumentException("Invalid day of the week: " + dayOfWeekArg);
    }
    return typeOfDay;
}
```

The `switch` statement compares the `String` object in its expression with the expressions associated with each `case` label as if it were using the `String.equals` method; consequently, the comparison of `String` objects in `switch` statements is case sensitive.

The try-with-resources Statement

The `try-with-resources` statement is a `try` statement that declares one or more resources. A *resource* is an object that must be closed after the program is finished with it. The `try-with-resources` statement ensures that each resource is closed at the end of the statement. Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, can be used as a resource.

The following example reads the first line from a file. It uses an instance of `BufferedReader` to read data from the file. `BufferedReader` is a resource that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

In this example, the resource declared in the `try-with-resources` statement is a `BufferedReader`. The declaration statement appears within parentheses immediately after the `try` keyword. The class `BufferedReader`, in Java SE 7 and later, implements the interface `java.lang.AutoCloseable`. Because the `BufferedReader` instance is declared in a `try-with-resources` statement, it will be closed regardless of whether the `try` statement completes normally or abruptly (as a result of the method `BufferedReader.readLine` throwing an `IOException`).

Prior to Java SE 7, you can use a `finally` block to ensure that a resource is closed regardless of whether the `try` statement completes normally or abruptly. The following example uses a `finally` block instead of a `try-with-resources` statement:

```
static String readFirstLineFromFileWithFinallyBlock(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

However, in this example, if the methods `readLine` and `close` both throw exceptions, then the method `readFirstLineFromFileWithFinallyBlock` throws the exception thrown from the `finally` block; the exception thrown from the `try` block is suppressed. In contrast, in the example `readFirstLineFromFile`, if exceptions are thrown from both the `try` block and the `try-with-resources` statement, then the method `readFirstLineFromFile` throws the exception thrown from the `try` block; the exception thrown from the `try-with-resources` block is suppressed. In Java SE 7 and later, you can retrieve suppressed exceptions; see the section [Suppressed Exceptions](#) for more information.

You may declare one or more resources in a `try-with-resources` statement. The following example retrieves the names of the files packaged in the zip file `zipFileName` and creates a text file that contains the names of these files:

```
public static void writeToFileZipFileContents(String zipFileName, String outputFileName)
    throws java.io.IOException {

    java.nio.charset.Charset charset = java.nio.charset.Charset.forName("US-ASCII");
    java.nio.file.Path outputPath = java.nio.file.Paths.get(outputFileName);

    // Open zip file and create output file with try-with-resources statement

    try (
        java.util.zip.ZipFile zf = new java.util.zip.ZipFile(zipFileName);
        java.io.BufferedWriter writer = java.nio.file.Files.newBufferedWriter(outputPath, charset)
    ) {

        // Enumerate each entry

        for (java.util.Enumeration entries = zf.entries(); entries.hasMoreElements();) {

            // Get the entry name and write it to the output file

            String newLine = System.getProperty("line.separator");
            String zipEntryName = ((java.util.zip.ZipEntry)entries.nextElement()).getName() + newLine;
            writer.write(zipEntryName, 0, zipEntryName.length());
        }
    }
}
```

In this example, the `try-with-resources` statement contains two declarations that are separated by a semicolon: `ZipFile` and `BufferedWriter`. When the block of code that directly follows it terminates, either normally or because of an exception, the `close` methods of the `BufferedWriter` and `ZipFile` objects are automatically called in this order. Note that the `close` methods of resources are called in the *opposite* order of their creation.

The following example uses a `try-with-resources` statement to automatically close a `java.sql.Statement` object:

```
public static void viewTable(Connection con) throws SQLException {

    String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";

    try (Statement stmt = con.createStatement()) {

        ResultSet rs = stmt.executeQuery(query);

        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + ", " + supplierID + ", " + price +
                               ", " + sales + ", " + total);
        }

    } catch (SQLException e) {
        JDBCTutorialUtilities.printStackTrace(e);
    }
}
```

The resource `java.sql.Statement` used in this example is part of the JDBC 4.1 and later API.

Note: A `try-with-resources` statement can have `catch` and `finally` blocks just like an ordinary `try` statement. In a `try-with-resources` statement, any `catch` or `finally` block is run after the resources declared have been closed.

Catching Multiple Exception Types and Rethrowing Exceptions with Improved Type Checking

Handling More Than One Type of Exception

In Java SE 7 and later, a single `catch` block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

Consider the following example, which contains duplicate code in each of the `catch` blocks:

```
catch (IOException ex) {
    logger.log(ex);
    throw ex;
}
catch (SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

In releases prior to Java SE 7, it is difficult to create a common method to eliminate the duplicated code because the variable `ex` has different types.

The following example, which is valid in Java SE 7 and later, eliminates the duplicated code:

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

The `catch` clause specifies the types of exceptions that the block can handle, and each exception type is separated with a vertical bar (`|`).

Note: If a `catch` block handles more than one exception type, then the `catch` parameter is implicitly `final`. In this example, the `catch` parameter `ex` is `final` and therefore you cannot assign any values to it within the `catch` block.

Bytecode generated by compiling a `catch` block that handles multiple exception types will be smaller (and thus superior) than compiling many `catch` blocks that handle only one exception type each. A `catch` block that handles multiple exception types creates no duplication in the bytecode generated by the compiler; the bytecode has no replication of exception handlers.

Rethrowing Exceptions with More Inclusive Type Checking

The Java SE 7 compiler performs more precise analysis of rethrown exceptions than earlier releases of Java SE. This enables you to specify more specific exception types in the `throws` clause of a method declaration.

Consider the following example:

```
static class FirstException extends Exception { }
static class SecondException extends Exception { }

public void rethrowException(String exceptionName) throws Exception {
    try {
        if (exceptionName.equals("First")) {
            throw new FirstException();
        } else {
            throw new SecondException();
        }
    } catch (Exception e) {
        throw e;
    }
}
```

This example's `try` block could throw either `FirstException` or `SecondException`. Suppose you want to specify these exception types in the `throws` clause of the `rethrowException` method declaration. In releases prior to Java SE 7, you cannot do so. Because the exception parameter of the `catch` clause, `e`, is type `Exception`, and the `catch` block rethrows the exception parameter `e`, you can only specify the exception type `Exception` in the `throws` clause of the `rethrowException` method declaration.

However, in Java SE 7, you can specify the exception types `FirstException` and `SecondException` in the `throws` clause in the `rethrowException` method declaration. The Java SE 7 compiler can determine that the exception thrown by the statement `throw e` must have come from the `try` block, and the only exceptions thrown by the `try` block can be `FirstException` and `SecondException`. Even though the exception parameter of the `catch` clause, `e`, is type `Exception`, the compiler can determine that it is an instance of either `FirstException` or `SecondException`:

```

public void rethrowException(String exceptionName)
throws FirstException, SecondException {
    try {
        // ...
    }
    catch (Exception e) {
        throw e;
    }
}

```

This analysis is disabled if the `catch` parameter is assigned to another value in the `catch` block. However, if the `catch` parameter is assigned to another value, you must specify the exception type `Exception` in the `throws` clause of the method declaration.

In detail, in Java SE 7 and later, when you declare one or more exception types in a `catch` clause, and rethrow the exception handled by this `catch` block, the compiler verifies that the type of the rethrown exception meets the following conditions:

- The `try` block is able to throw it.
- There are no other preceding `catch` blocks that can handle it.
- It is a subtype or supertype of one of the `catch` clause's exception parameters.

The Java SE 7 compiler allows you to specify the exception types `FirstException` and `SecondException` in the `throws` clause in the `rethrowException` method declaration because you can rethrow an exception that is a supertype of any of the types declared in the `throws`.

In releases prior to Java SE 7, you cannot throw an exception that is a supertype of one of the `catch` clause's exception parameters. A compiler from a release prior to Java SE 7 generates the error, "unreported exception `Exception`; must be caught or declared to be thrown" at the statement `throw e`. The compiler checks if the type of the exception thrown is assignable to any of the types declared in the `throws` clause of the `rethrowException` method declaration. However, the type of the `catch` parameter `e` is `Exception`, which is a supertype, not a subtype, of `FirstException` and `SecondException`.

Type Inference for Generic Instance Creation

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (`<>`) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the *diamond*.

For example, consider the following variable declaration:

```
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

In Java SE 7, you can substitute the parameterized type of the constructor with an empty set of type parameters (`<>`):

```
Map<String, List<String>> myMap = new HashMap<>();
```

Note that to take advantage of automatic type inference during generic class instantiation, you must specify the diamond. In the following example, the compiler generates an unchecked conversion warning because the `HashMap()` constructor refers to the `HashMap` raw type, not the `Map<String, List<String>>` type:

```
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

Java SE 7 supports limited type inference for generic instance creation; you can only use type inference if the parameterized type of the constructor is obvious from the context. For example, the following example does not compile:

```
List<String> list = new ArrayList<>();
list.add("A");
```

```

// The following statement should fail since addAll expects
// Collection<? extends String>

```

```
list.addAll(new ArrayList<>());
```

Note that the diamond often works in method calls; however, it is suggested that you use the diamond primarily for variable declarations.

Java NIO.2 –File Navigation Helpers

```
//Make a reference to a
File
Path src = Paths.get("/home/fred/readme.txt");
Path dst = Paths.get("/home/fred/copy_readme.txt");
//Make a reference to a
path
Path src = Paths.get("/home/fredSRC/");
Path dst = Paths.get("/home/fredDST/");
//Navigation /home/fredSRC -> /home/fredSRC/tmp
Path tmpPath = src.resolve("tmp");
//Create a relative path from src -> ..
Path relativePath = tmpPath.relativeize(src);
// Convert to old File Format for your legacy apps
File file = aPathPath.toFile();
```

Java NIO.2 Features –Files Helper Class

Class `java.nio.file.Files` exclusively static methods to operate on files, directories and other types of files

- Files helper class is feature rich:Copy
- Create Directories
- Create Files
- Create Links
- Use of system “temp” directory
- Delete
- Attributes –Modified/Owner/Permissions/Size, etc.
- Read/Write

```
Files.move(src, dst, StandardCopyOption.ATOMIC_MOVE);
Files.copy(src, dst, StandardCopyOption.COPY_ATTRIBUTES,
StandardCopyOption.REPLACE_EXISTING);
```

Java NIO.2 Directories

DirectoryStream iterate over entries Scales to large directories

Uses less resources

Smooth out response time for remote file systems

Implements **Iterable** and **Closeable** for productivity

Filtering support Build-in support for glob, regex and custom filters

```
Path srcPath = Paths.get("/home/fred/src");
try (DirectoryStream<Path> dir =
    srcPath.newDirectoryStream("*.java")) {
    for (Path file
        : dir)
        System.out.println(file.getName());
}
```

Concurrency APIs JSR 166y -Phasers

PhaserBarrier similar to **CyclicBarrier** and **CountDownLatch**

Used for many threads to wait at common barrier point For example, use this to create N threads that you want to do something simultaneously –“start gun” metaphore

How is Phaser an improvement? Dynamic add/remove “parties” to be sync’d

Better deadlock avoidance

Arrival “counting” and phase advance options, etc

Termination api’s

Tiering (tree structure) Rather than sync 100 threads, sync 2x50 then 2x.

TransferQueue interface Extension to **BlockingQueue**

Implemented by **LinkedTransferQueue**

Additional Benefits: Adds methods: `transfer(E e)`, `tryTransfer(E e)`, `tryTransfer(E e, long timeout)`, `hasWaitingConsumer()`, `getWaitingConsumerCount()`

Allows for smarter queues to be built –sidestep the data structure if it’s known there are consumers waiting.

Fork Join Framework -JSR 166y

ForkJoinPoolService for running **ForkJoinTasks**

aFjp.execute(aTask); // async

aFjp.invoke(aTask); // wait

aFjp.submit(aTask); // async + future

ForkJoinPool(); // default to platform

**ForkJoinPool(int n); // # concurrent
threads**

**ForJoinPool(n,aThreadFactory,exHandler,FIFO
tasks); // Create your own thread handler,
exception handler, and boolean on task
ordering (default LIFO)**

```
ForkJoinPool p = new ForkJoinPool();  
  
MyTask mt = new MyTask(n); // implements compute  
  
p.submit(mt);  
  
while (!mt.isDone()) { /*THUMPER!*/ }  
  
System.out.println(mt.get());
```


<? extends T> Vs <? super T> - ProducerExtends ConsumerSuper

Basic object structure

```
package com.ddlab.rnd.pecs;
class Animal {
}

class Carnivores extends Animal {
}

class Herbivores extends Animal {
}

class Tiger extends Carnivores {
}

class Lion extends Animal {
}

class Cow extends Herbivores {
}

class Zebra extends Herbivores {
}
```

Case- I – Define list as read-only, make it as producer

```
List<? extends Animal> animalList = new ArrayList<>();
```

```
Animal animal = new Animal();
Herbivores herbivores = new Herbivores();
Carnivores carnivores = new Carnivores();
Lion lion = new Lion();
Tiger tiger = new Tiger();
Cow cow = new Cow();
//The following lines will not be compiled
animalList.add(animal); // Compilation Issue
animalList.add(herbivores); // Compilation Issue
animalList.add(lion); // Compilation Issue
animalList.add(tiger); // Compilation Issue
animalList.add(cow); // Compilation Issue
```

Case- II – Add the data to the list and then make it read-only.

```
List<Animal> list = new ArrayList<>();

Animal animal = new Animal();
Herbivores herbivores = new Herbivores();
Carnivores carnivores = new Carnivores();
Lion lion = new Lion();
Tiger tiger = new Tiger();
Cow cow = new Cow();

list.add(animal);
list.add(herbivores);
list.add(lion);
list.add(tiger);
list.add(cow);

List<? extends Animal> animalList = new ArrayList<>();
animalList = list; //Now it becomes read-only, You can add anything

animalList.add(new Zebra()); //Compilation Issue
animalList.add(herbivores); //Compilation Issue
```

```
/**
 * Here the list is the considered as readonly,
 * there is no need to define as unModifiable list.
 * This is the best approach because it provides
 * information at compile time but where as
 * Collections.unModifiableList() provides information
 * only at the Runtime
 *
 * @param list
 */
public static void onlyTraverse(List<? extends Animal> list) {
    for (Animal animal : list)
        System.out.println(animal);

//      list.add( new Zebra()); //Compilation Issue as you cannot add
}
```

You can think why can't I use Collections.unModifiableList(). Yes you can use it, but it will throw exceptions at runtime only.

```
List newList = Collections.unmodifiableList(list);
tryToTraverse(newList);
public static void tryToTraverse(List<Animal> list) {

    list.add( new Zebra() ); //It will throw Exception only at runtime.
}
```

? extends T is producer, you can only iterate the list of elements, but you cannot add the elements.

<http://stackoverflow.com/questions/2723397/java-generics-what-is-pecs>

Suppose you have a method that takes as its parameter a collection of things, but you want it to be more flexible than just accepting a `Collection<Thing>`.

Case 1: You want to go through the collection and do things with each item.

Then the list is a **producer**, so you should use a `Collection<? extends Thing>`.

The reasoning is that a `Collection<? extends Thing>` could hold any subtype of `Thing`, and thus each element will behave as a `Thing` when you perform your operation. (You actually cannot add anything to a `Collection<? extends Thing>`, because you cannot know at runtime which *specific* subtype of `Thing` the collection holds.)

Case 2: You want to add things to the collection.

Then the list is a **consumer**, so you should use a `Collection<? super Thing>`.

The reasoning here is that unlike `Collection<? extends Thing>`, `Collection<? super Thing>` can always hold a `Thing` no matter what the actual parameterized type is. Here you don't care what is already in the list as long as it will allow a `Thing` to be added; this is what `? super Thing` guarantees.

The principles behind this in Computer Science is named after

- Covariance - `? extends MyClass`,
- Contravariance - `? super MyClass` and
- Invariance/non-Variance - `MyClass`

PECS (short for "**P**roducer `extends` and **C**onsumer `super`") can be explained by : **Get and Put Principle**
Get And Put Principle (From Java Generics and Collections)

It states,

1. use an **extends wildcard** when you only **get** values out of a structure
2. use a **super wildcard** when you only **put** values into a structure
3. and **don't use a wildcard** when you **both get and put**.

<? Super T> as consumer

```
List<? super Animal> list = new ArrayList<>();  
list.add(animal);  
list.add(carnivores);  
list.add(herbivores);  
list.add(tiger);  
list.add(lion);  
list.add(cow);
```

The above works fine and there is no compilation issue.

```
List<? super Carnivores> carList = new ArrayList<>();  
carList.add(animal); //Compilation Issue  
carList.add(carnivores);  
carList.add(herbivores); //Compilation Issue  
carList.add(tiger);  
carList.add(lion); //Compilation Issue, Lion extends Animal  
carList.add(cow); //Compilation Issue
```

```

public static void consume(List<? super Animal> list ) {
    for(Animal animal : list) { //Compilation Issue

    }
    //But the following is fine
    for (int i = 0; i < list.size(); i++) {
        System.out.println(list.get(i));
    }
    list.add(new Zebra()); //Ok
}

```

<? Super T> as Producer

```

public static void asProducer(List<? extends Animal> list) {
    //You can only iterate the list, you cannot add the elements
    for(Animal animal : list)
        System.out.println(animal);
}

```

Practical usage of PECS

```

import java.util.ArrayList;
import java.util.List;
public class Util {

    /**
     * Iterate from the source list and
     * copy to the destination list
     */
    public static <T> void copy(List<? extends T> srcList, List<? super T> destnList) {
        for(T t : srcList) {
            destnList.add(t);
        }
    }

    public static void main(String[] args) {
        Animal animal = new Animal();
        Carnivores carnivores = new Carnivores();
        Herbivores herbivores = new Herbivores();
        Tiger tiger = new Tiger();
        Lion lion = new Lion();
        Cow cow = new Cow();

        List<Animal> srcList = new ArrayList<>();
        srcList.add(animal);
        srcList.add(carnivores);
        srcList.add(herbivores);
        srcList.add(tiger);
        srcList.add(lion);
        srcList.add(cow);
        List<Animal> destnList = new ArrayList<>();
        copy(srcList, destnList);
        for(Animal an : destnList)
            System.out.println("Animal Obj ::: "+an);
    }
}

```

PECS—Producer extends, Consumer super

- use `Foo<? extends T>` for a T producer
- use `Foo<? super T>` for a T consumer
- **Only applies to input parameters**
- **Don't use wildcard types as return types**

Complete Example is given below.

```
import java.util.ArrayList;
import java.util.List;

public class TestPECS {

    public static <T> void copy(List<? extends T> srcList, List<? super T> destnList) {
        for (T t : srcList)
            destnList.add(t);
    }

    public static void asProducer(List<? extends Animal> list) {
        //You can only iterate the list, you cannot add the elements
        for(Animal animal : list)
            System.out.println(animal);
    }

    public static void asConsumer(List<? super Animal> list) {
        //You can not iterate the list using advanced for loop
        //You can use normal for(i) loop to get the elements
        //You can add the elements to the list
        Animal animal = new Animal();
        Carnivores carnivores = new Carnivores();
        Herbivores herbivores = new Herbivores();
        Tiger tiger = new Tiger();
        Lion lion = new Lion();
        Cow cow = new Cow();

        list.add(animal);
        list.add(carnivores);
        list.add(herbivores);
        list.add(tiger);
        list.add(lion);
        list.add(cow);
    }

    public static void main(String[] args) {
        Animal animal = new Animal();
        List<Animal> list = new ArrayList<>();
        asProducer(list);
        asConsumer(list);
    }
}
```

Why Java uses two kinds of Sort Merge Sort as well as Quick Sort

Highly likely from Josh Bloch [§](#):

I did write these methods, so I suppose I'm qualified to answer. It is true that there is no single best sorting algorithm. QuickSort has two major deficiencies when compared to mergesort:

1. It's not stable (as parsifal noted).
2. It doesn't *guarantee* $n \log n$ performance; it can degrade to quadratic performance on pathological inputs. Stability is a non-issue for primitive types, as there is no notion of identity as distinct from (value) equality. And the possibility of quadratic behavior was deemed not to be a problem in practice for Bentley and McIlroy's implementation (or subsequently for [Dual Pivot Quicksort](#)), which is why these QuickSort variants were used for the primitive sorts.

Stability is a big deal when sorting arbitrary objects. For example, suppose you have objects representing email messages, and you sort them first by date, then by sender. You expect them to be sorted by date within each sender, but that will only be true if the sort is stable. That's why we elected to provide a stable sort (Merge Sort) to sort object references. (Technically speaking, multiple sequential stable sorts result in a lexicographic ordering on the keys in the reverse order of the sorts: the final sort determines the most significant subkey.)

It's a nice side benefit that Merge Sort *guarantees* $n \log n$ (time) performance no matter what the input. Of course there is a down side: quick sort is an "in place" sort: it requires only $\log n$ external space (to maintain the call stack). Merge, sort, on the other hand, requires $O(n)$ external space. The TimSort variant (introduced in Java SE 6) requires substantially less space ($O(k)$) if the input array is nearly sorted.

We know that quick sort is the best sorting algorithm.

Also, the [following](#) is relevant:

The algorithm used by `java.util.Arrays.sort` and (indirectly) by `java.util.Collections.sort` to sort object references is a "modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist)." It is a reasonably fast stable sort that guarantees $O(n \log n)$ performance and requires $O(n)$ extra space. In its day (it was written in 1997 by Joshua Bloch), it was a fine choice, but today but we can do much better.

Since 2003, Python's list sort has used an algorithm known as timsort (after Tim Peters, who wrote it). It is a stable, adaptive, iterative mergesort that requires far fewer than $n \log(n)$ comparisons when running on partially sorted arrays, while offering performance comparable to a traditional mergesort when run on random arrays. Like all proper mergesorts timsort is stable and runs in $O(n \log n)$ time (worst case). In the worst case, timsort requires temporary storage space for $n/2$ object references; in the best case, it requires only a small constant amount of space. Contrast this with the current implementation, which always requires extra space for n object references, and beats $n \log n$ only on nearly sorted lists.

Timsort is described in detail here: <http://svn.python.org/projects/python/trunk/Objects/listsort.txt>.

Tim Peters's original implementation is written in C. Joshua Bloch ported it from C to Java and end tested, benchmarked, and tuned the resulting code extensively. The resulting code is a drop-in replacement for `java.util.Arrays.sort`. On highly ordered data, this code can run up to 25 times as fast as the current implementation (on the HotSpot server VM). On random data, the speeds of the old and new implementations are comparable. For very short lists, the new implementation is substantially faster than the old even on random data (because it avoids unnecessary data copying).

Also, see [Is Java 7 using Tim Sort for the Method Arrays.Sort?](#).

There isn't a single "best" choice. As with many other things, it's about tradeoffs.

Answer given by Joshua Bloch

We know that quick sort is the best sorting algorithm.

There isn't a single "best" choice. As with many other things, it's about tradeoffs.

As to the "why" question, only the original author can answer it with any certainty. However, the following is relevant:

The algorithm used by `java.util.Arrays.sort` and (indirectly) by `java.util.Collections.sort` to sort object references is a "modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist)." It is a reasonably fast stable sort that guarantees $O(n \log n)$ performance and requires $O(n)$ extra space. In its day (it was written in 1997 by Joshua Bloch), it was a fine choice, but today but we can do much better.

Since 2003, Python's list sort has used an algorithm known as timsort (after Tim Peters, who wrote it). It is a stable, adaptive, iterative mergesort that requires far fewer than $n \log(n)$ comparisons when running on partially sorted arrays, while offering performance comparable to a traditional mergesort when run on random arrays. Like all proper mergesorts timsort is stable and runs in $O(n \log n)$ time (worst case). In the worst case, timsort requires temporary storage space for $n/2$ object references; in the best case, it requires only a small constant amount of space. Contrast this with the current implementation, which always requires extra space for n object references, and beats $n \log n$ only on nearly sorted lists.

Timsort is described in detail here: <http://svn.python.org/projects/python/trunk/Objects/listsort.txt>.

Tim Peters's original implementation is written in C. Joshua Bloch ported it from C to Java and end tested, benchmarked, and tuned the resulting code extensively. The resulting code is a drop-in replacement for `java.util.Arrays.sort`. On highly ordered data, this code can run up to 25 times as fast as the current implementation (on the HotSpot server VM). On random data, the speeds of the old and new implementations are comparable. For very short lists, the new implementation is substantially faster than the old even on random data (because it avoids unnecessary data copying).

Also, see [Is Java 7 using Tim Sort for the Method Arrays.Sort?](#)

I did write these methods, so I suppose I'm qualified to answer. It is true that there is no single best sorting algorithm. QuickSort has two major deficiencies when compared to mergesort (1) It's not stable (as parsifal noted), and (2) It doesn't *guarantee* $n \log n$ performance; it can degrade to quadratic performance on pathological inputs. Stability is a non-issue for primitive types, as there is no notion of identity as distinct from (value) equality. And the possibility of quadratic behavior was deemed not to be a problem in practice for Bentley and McIlroy's implementation (or subsequently for Dual Pivot Quicksort), which is why these QuickSort variants were used for the primitive sorts.

Stability is a big deal when sorting arbitrary objects. For example, suppose you have objects representing email messages, and you sort them first by date, then by sender. You expect them to be sorted by date within each sender, but that will only be true if the sort is stable. That's why we elected to provide a stable sort (Merge Sort) to sort object references. (Technically speaking, multiple sequential stable sorts result in a lexicographic ordering on the keys in the reverse order of the sorts: the final sort determines the most significant subkey.)

It's a nice side benefit that Merge Sort *guarantees* $n \log n$ (time) performance no matter what the input. Of course there is a down side: quick sort is an "in place" sort: it requires only $\log n$ external space (to maintain the call stack). Merge, sort, on the other hand, requires $O(n)$ external space. The TimSort variant (introduced in Java SE 6) requires substantially less space ($O(k)$) if the input array is nearly sorted.

--Josh