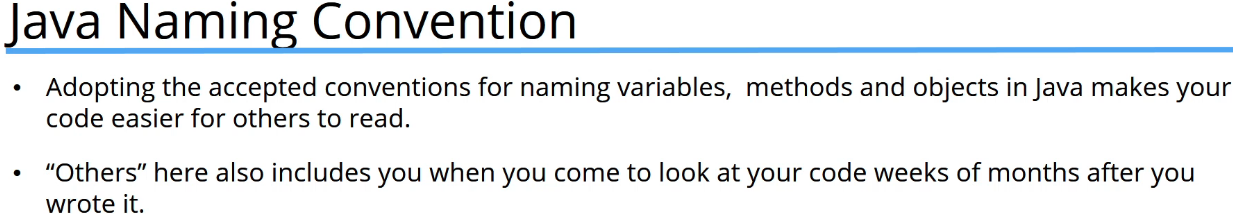
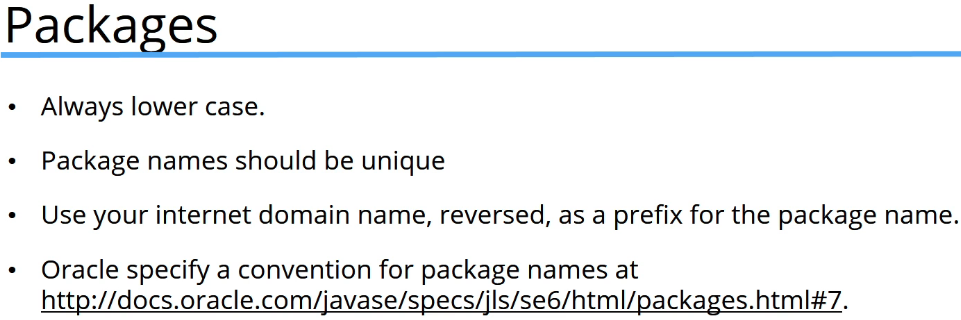
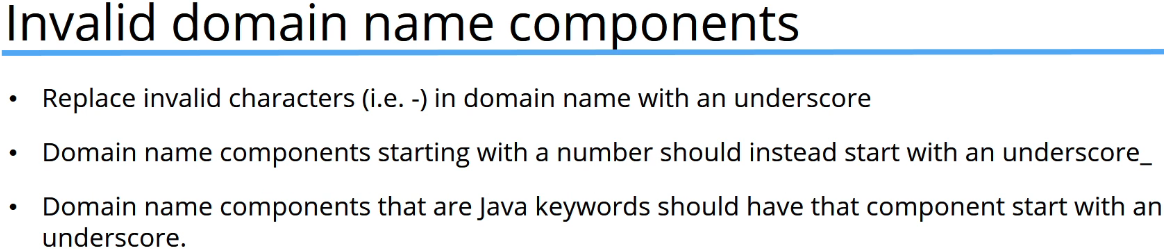
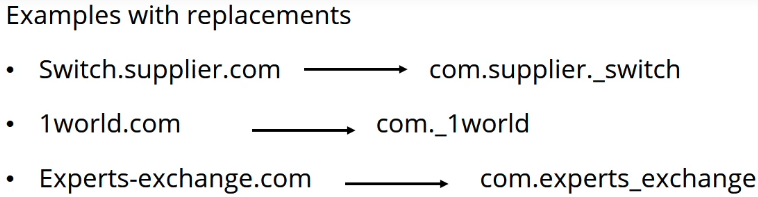
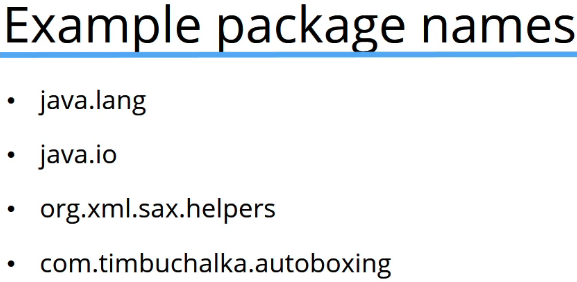
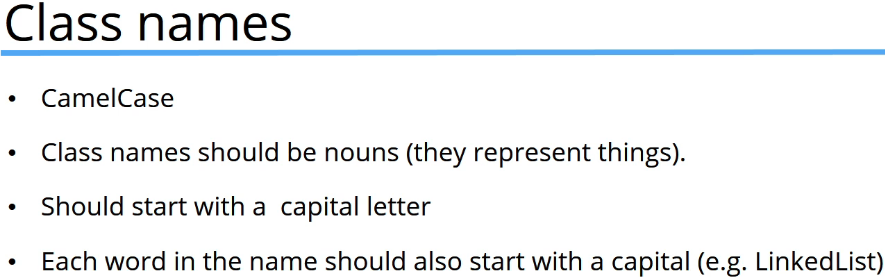
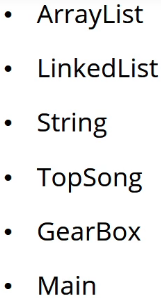
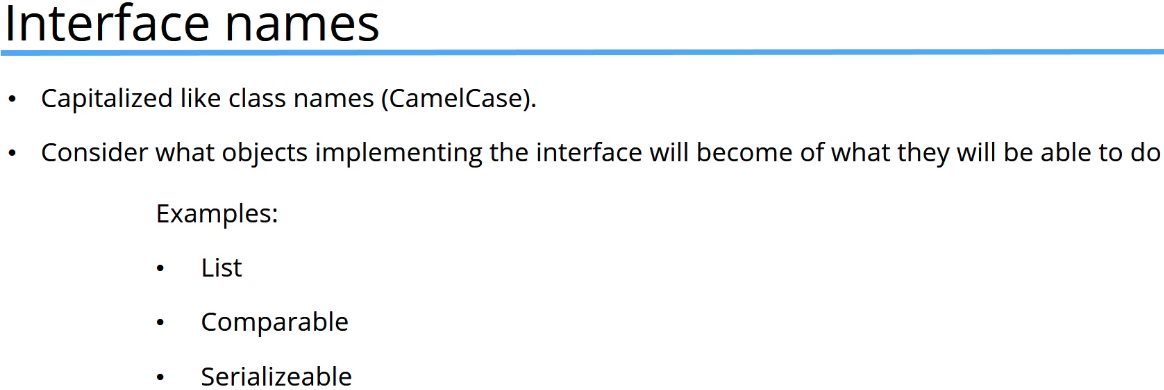
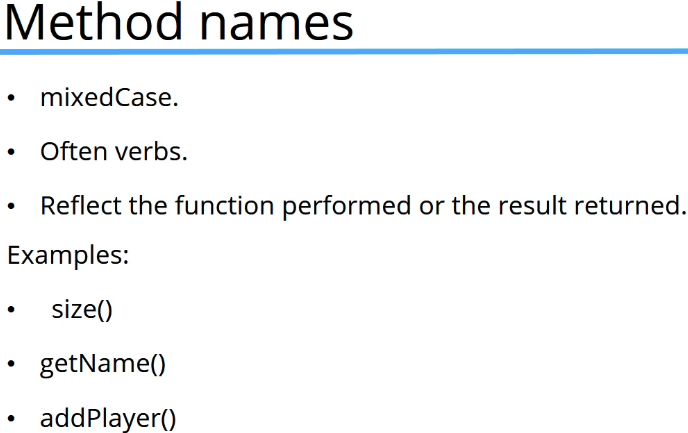
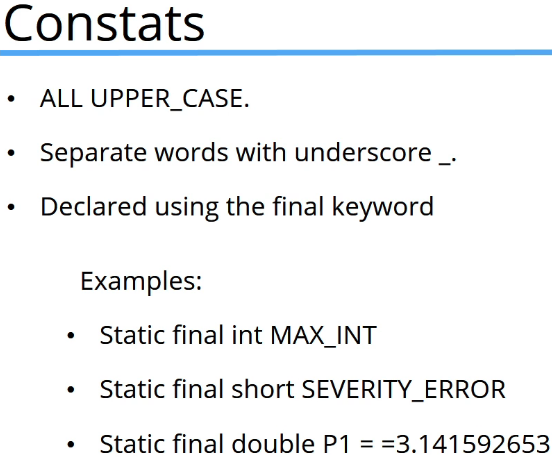
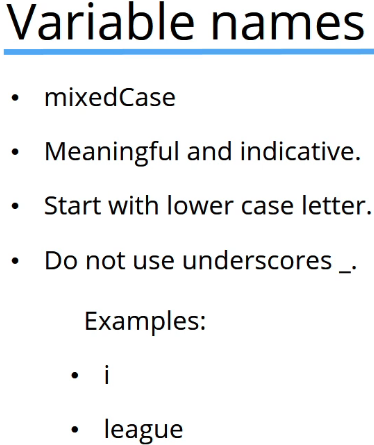
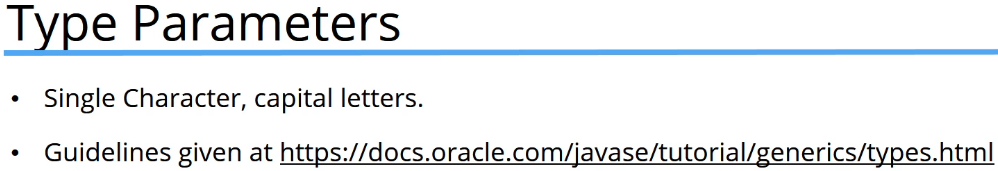
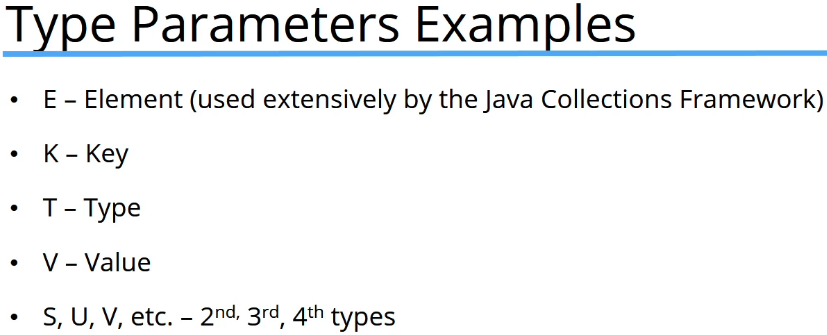
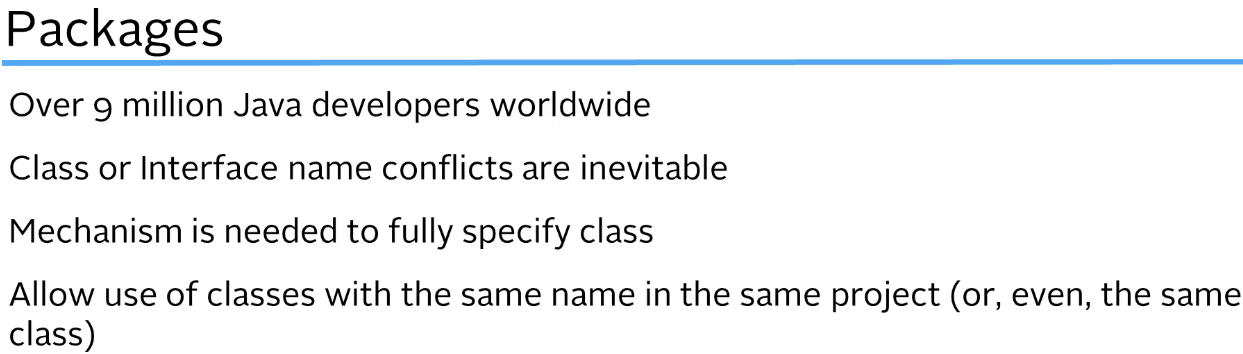
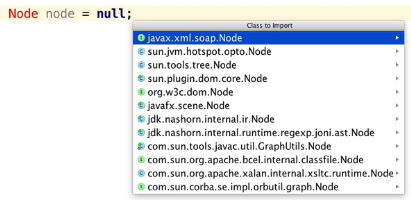
**Naming Conventions**  
\* There’s a good reason why you wanna do this.  
  
  
\* Let’s look at the conventions:  
  
  
  


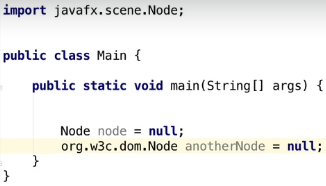
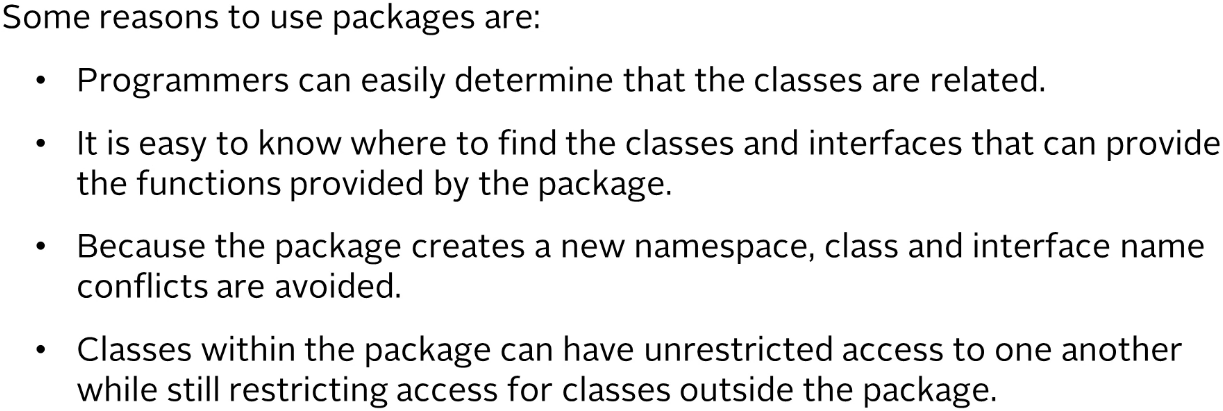
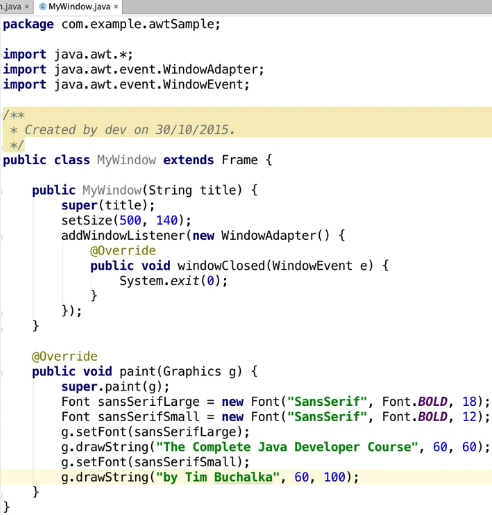
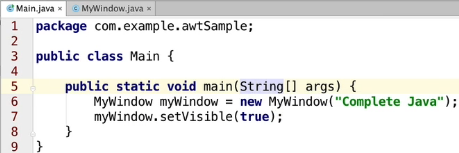


\* Examples:

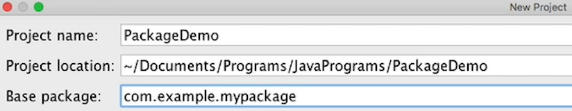
  
  
  


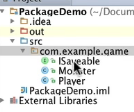
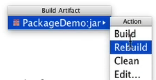
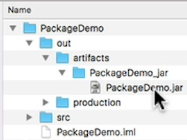
  
  


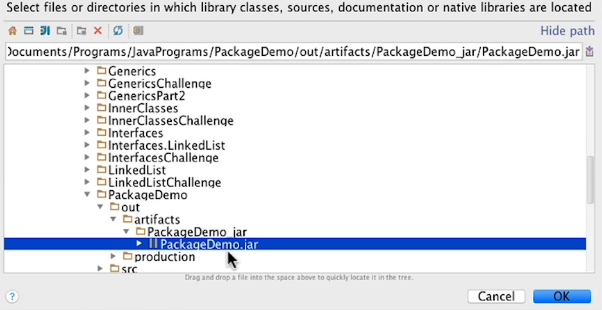
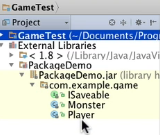
**Packages**  
  
  
\* There’s ton of different options that Java is recommending with the word Node in it.  
\* Think of Package as a way of grouping related classes and interfaces together. The package mechanism provides a way to manage the namespace of object types and it also extends access protection beyond the traditional access modifiers.

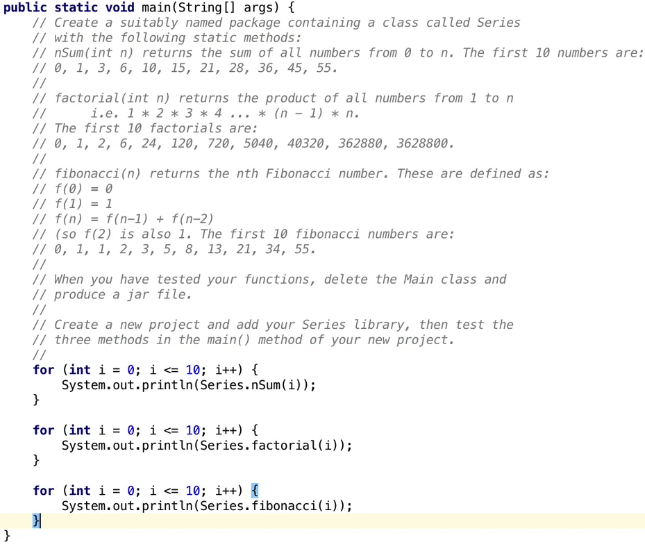
\* There’s another way of using import besides:  
  
  
=> specifying the actual package that contains the Node class that we wanna work with.  
\* The important difference is, unlike the first situation, in this case the package is not imported. That means that every time you wanna reference this Node, you have to type it out.  
\* If you’re using 2 different Node types in the same class, then you have to refer to one of them that way.  
  
\* Java and IntelliJ won’t allow us to do this even if we use the long way if typing it later.  
\* You can do it like this though:  
  
  
\* You don’t see an import for java.lang because it’s automatically imported for us. It contains the class object that is ussed to support our classes as well as Integer and Double, etc. and all the fundamental building blocks of a Java program.  
\* Before we create our own packages, it’s worthwhile to have a look at some of the other ones that are included when you install the Java Software Development Kit.  
=> New > Project… > Java > Java EE > Create project from template > Command Line App > name and Base package name > finish.  
  
\* If you click on the close button in the window, it’s gonna override the standard function, and close down the app which doesn’t happen by default.  
\* We should be using windowClosing instead of windowClosed because otherwise the application will keep running in the background, windowClosed is call when the window has been close while windownClosing is called when a window is in the process of being closed. In an ideal world, it would probably be better to use jframe and swing utilities that invoke later when creating a frame but I haven’t really worried about it too much here because it’s meant to be just a simple example.   
  
  
\* Can you actually imagine having to do this all manually? You can see there’s relatively speaking considering what’s actually happened, there’s actually very little code you have to type. All the infrastructure code to do a lot of this work is automatically in these various classes, these packages that havev been provided by Java. So writing all this yourself, writing a window manager something that actually creates fonts, that actually sets different fonts and draw things on the screen, positioning it up in certain places… that would be huge undertaking to do it yourself and it can be months or even years of work. But by using the awt package, we can take advantage of the facilities that it’s provided for producing these programs.   
\* awt stands for Abstract Window Toolkit.  
\* Now because that awt package provides everything necessary to create the GUI interface, we know that we can actually find all the classes and methods that we need within that package.  
\* And the thing is thaat if awt needs to be upgraded, new packages can be deployed rather than having to update files that are scattered around the Java SDK.  
\* So that’s why packages are really essential. They can actually bundle these changes and from your perspective nothing really changes, you’re still accessing them via the package, all the new code is actually happening and it’s all actually added to the package, making it a fast simple exercise for you to work with.

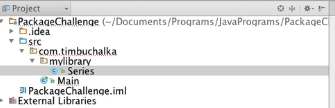
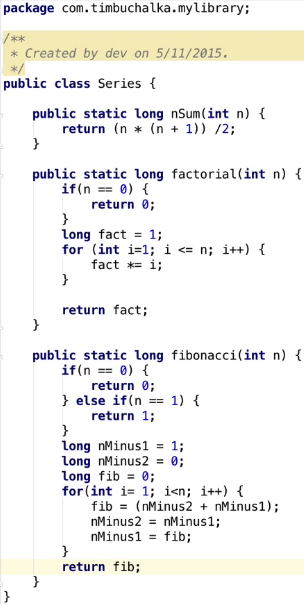
**Packages Part 2**  
\* The **\*** means to import all the classes, interfaces and static objects from the java.awt class.  
\* **But java.awt.\* doesn’t import java.awt.event.WindowAdapter for example. It’s not actually importing everything. There’s 2 different packages here. There’s java.awt and java.awt.event. So java.awt.event is actually a separate package to java.awt**.  
\* You can hold ctrl/alt to see where the class comes from.  
\* You can also check the External Libraries:  
  
\* You can see that the event is sort of like a subpackage but they are different.  
\* So when you’re importing classes from another package, you can choose to import the entire package or choose specific classes. However, if you’re using \*, there’s a possible impact if you’ve got a new classes or interfaces added to one of packages you actually import => if the other package contains a class or an interface with the same name, then your program’s not going to compile.  
\* Let’s now create our own package.

  
\* com.example is the convention that it’s reserved and can be used for packages that you’re never going to distribute and this is very useful if you don’t have your own domain.  
\* The package statement defines which package the class belongs to, but the directory structure has to also match the component parts of the package name.  
\* If you want to rename a package, you need to go to Refactor > Rename…

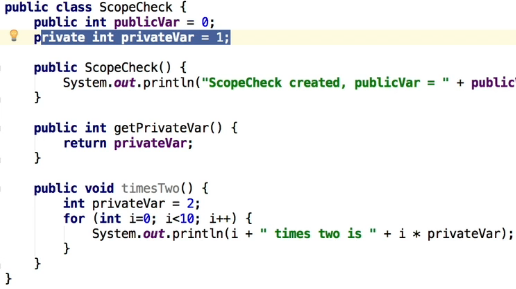
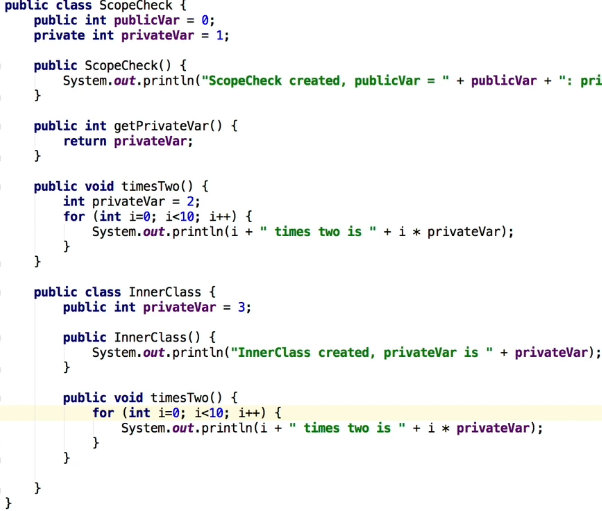
**Packages Part 3**  
\* To copy something from somewhere else, you copy it and than on the package where you want it you right click and select Paste.  
\* Now let’s say we want these 3 classes to be exported:  
  
**=> File > Project Structure… > Artifacts > + > JAR (stands for Java Archive and it’s sort of like a zip file for java code) > From modules with dependencies… > If you wanted to make the Java file executable, you’d specify Main class but we don’t want that now > extract to the target JAR > OK  
  
=> Build > Build Artifacts**  
  
\* We’ll choose Build.  


\* Now let’s create a new project and import it.  
**=> File > Project Structure… > Libraries > + > Java > specify where that is > OK > OK > OK**  
  
\* Now we can see it:  


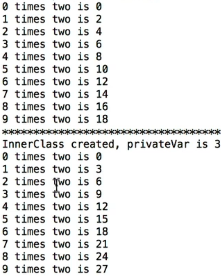
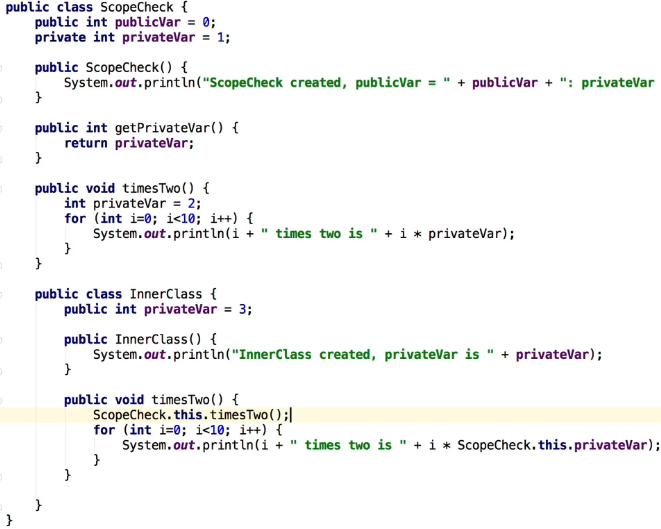
**Packages (Challenge Exercise)**   


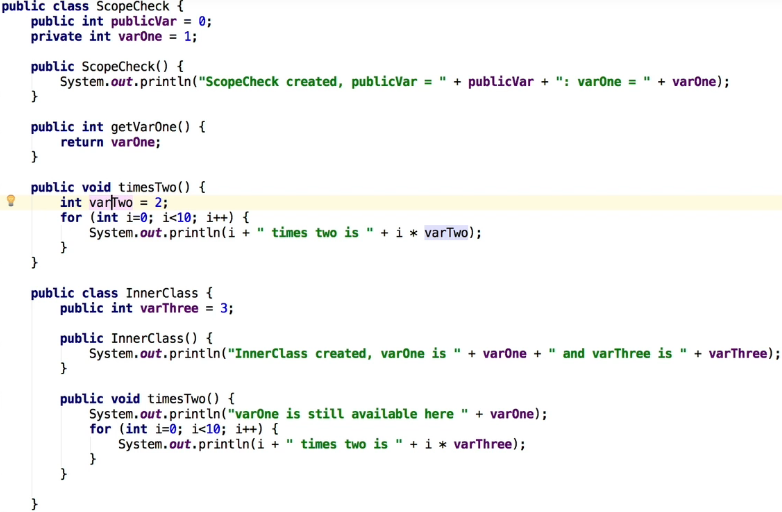
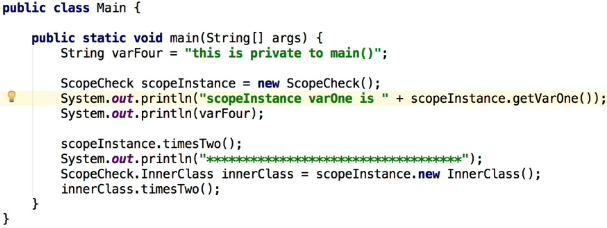
  
  
**!!! there’s a mistake - factorial !0 is 1, not 0.**  
\* To import the packge, go to:  
=> File > Project Structure… > Libraries > + > Java > navigate to the jar file.  
\* Now you can see it in the Eternal Libraries.

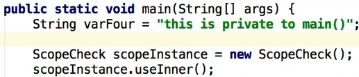
**Scope**  
\* Let’s look at scope and also granting or restricting access to our objects.  
\* Scope is probably one of the easiest things about programming once you understand it, but it’s also one of the most confusing things until you understand it.  
\* **Scope refers to the visibility of a class, member or variable**.  
\* Imagine you’re going to buy a particular computer and you find a local store and it’s got the computer for 20% off. Now the same computer in another store would still be at the full price though. So the scope in other words, or the price reduction is just that one local store. But, contrast that to a situation where you’ve got a large company -Walmart for example-, and they decided to discount a particular computer, so the discount then would apply in about 4 500 stores across America for example. So in that context the scope of the discount is therefore much much wider. If it company applied it world-wide, the scope would be 11 000 stores for example. The important thing here is that Walmart’s discount does not apply to any other stores, the scope is limited to the stored controlled by Walmart. And the same with the local store 20% off example, the scope there is just for that one local store.  
\* So the same computer in Java context would be the same variable.  
\* Our Java objects have scope in much the same way.  

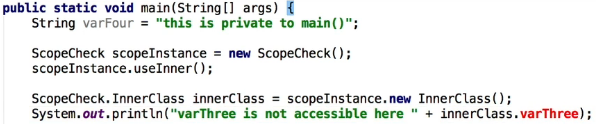

  
  
\* Java knows we’re referring to privateVar in Main because that’s the only one that’s available in the current scope.  
\* So access modifiers such as public, private, and protected, they’re one way to restrict the scope of an object.  
\* As it turns out, scope is even more important within the same class or method.  
  
\* It’s using the privateVar in the method because it’s local to this method, it’s in the scope.   
\* **That variable’s scope is the current method**.   
\* **It’s referring to the variable within the most local/narrow scope**.  
\* Another way to consider this is to use the concept of enclosing blocks. So when Java sees a reference to a variable, what it does is it starts by checking the current block of code to see if the variable was actually declared there. If it is declared there, then that variable is used. But if there’s no such declaration and it’s not there, then Java actually checks any block that encloses the current one to see if there’s a declaration there. So it keeps going backwards until it finds the variable declaration and if it doesn’t find one, then you’ll get an error in your code.  
\* In our example there’s no declaration of privateVar in the FOR block so it check the enclosing block which is the method {} itself in this case and it finds a declaration there.  
\* The i variable is declared in the FOR block.  
\* If we comment out the privateVar inside our method, it’s going to use the privateVar = 1. We could also use both in the method by specifying this.privateVar for the 1.  
\* **The concept of scope doesn’t just apply to variables. Methods also have scope, as do classes**.  
\* Class scope can be a little bit confusing, but let’s start by adding an inner class to our ScopeCheck class and have a look at what it does to the scope of things.  


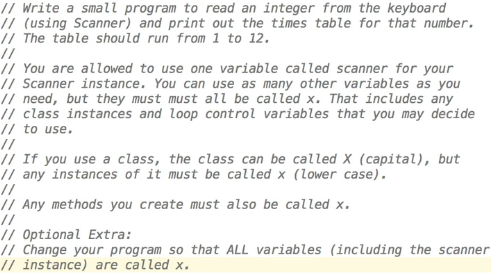


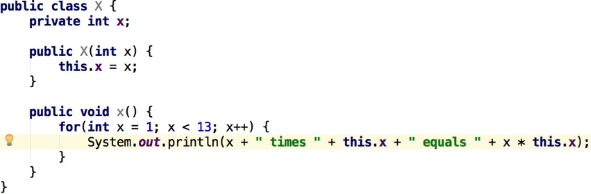
  
\*The rule of scope applies and because privateVar = 3 which is a field of the InnerClass and is the closest one declared, it uses that.  
\* The privateVar = 2 which is inside of timesTwo() method of ScopeCheck can’t be used in the InnerClass because they’re not enclosing each other.  
\* If we comment out the field in InnerClass, it’s going to use the field in ScopeCheck.  
\* **If we use this.privateVar inside of InnerClass’s timesTwo() method while the field in InnerClass is commented out, we get an error because it’s looking for the field in InnerClass.**\***However, we can type ScopeCheck.this.privateVar**.  
\* We can also call ScopeCheck.this.timesTwo() inside of Innerclass’ timesTwo() method.  


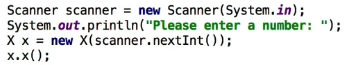
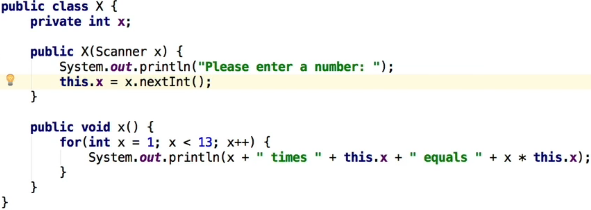
**Scope Part 2 and Visibility**  
\* In Java, things can be a little bit more complicated because objects themselves also have visibility.  
\* Let’s now rename all the privateVar variables and give them their own names varOne-varFour.  
  
  
\* This confirms that the object scope is the block in which it’s declared including any contained blocks.  
\* A variable can only shadow another variable with the same name if it’s declared in an enclosed block.  
\* That brings us to **Visibility**.  
\* In Java, an object’s visibility is governed by the access modifiers. The thing to point out is that visibility is connected to scope and we need to look at that aspect of it first before covering the access modifiers themselves and how they relate to access control.  
\* In the InnerClass we used varOne despite the fact that it was declared as private.  
\* InnerClass has access to all member variables or fields of its containing class. So not only is varOne in scope within the InnerClass, it’s also visible.  
\* The same is true the other way around, so the containing class, can also access all the methods and fields of a contained class even if they’re marked as private.  

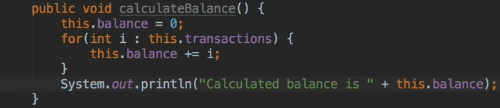
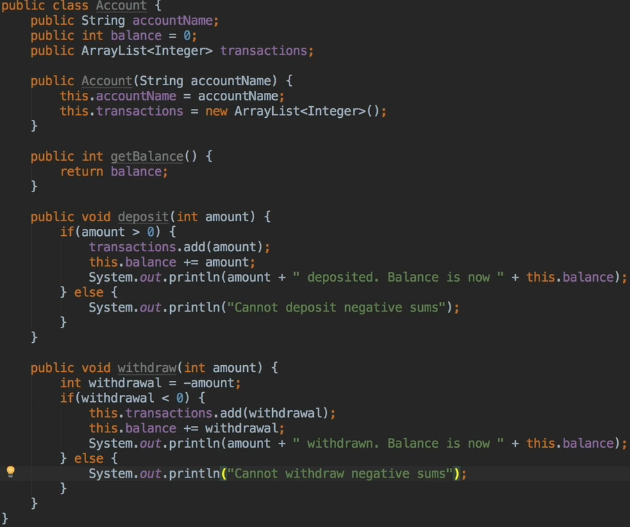

  
\* Now if we change the public int varThree to private and run it again, the code still works.  
  
\* So notice that the scope of our varThree hasn’t extended to the outer class so we actually had to qualify its name with the class instance in order to print that out.

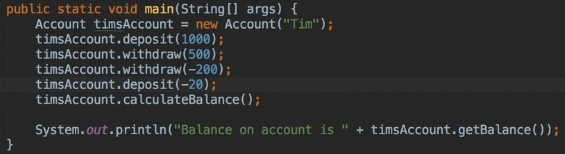
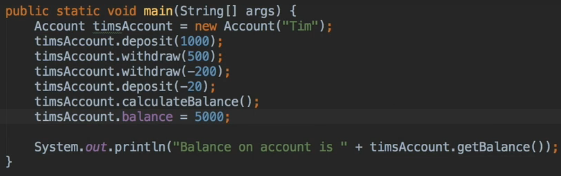
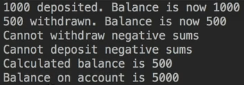
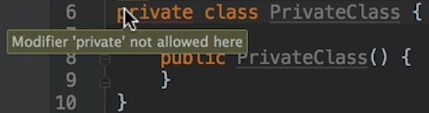
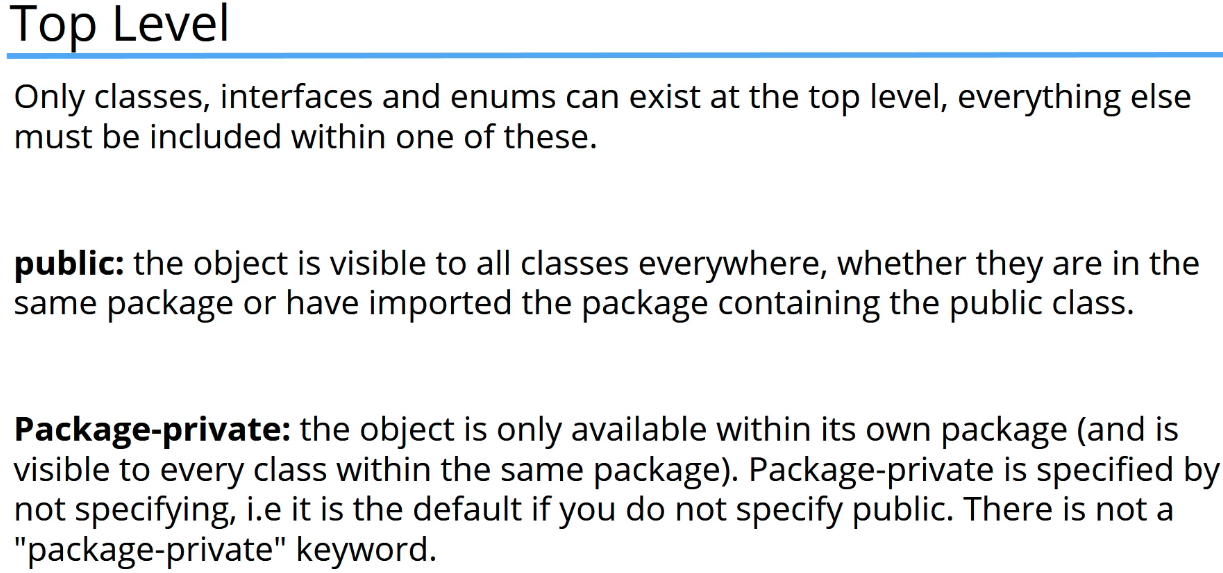
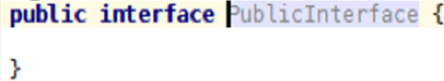
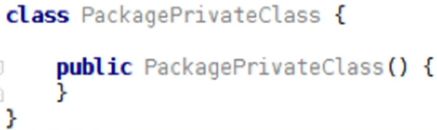
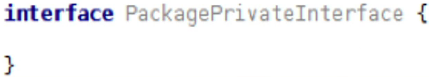
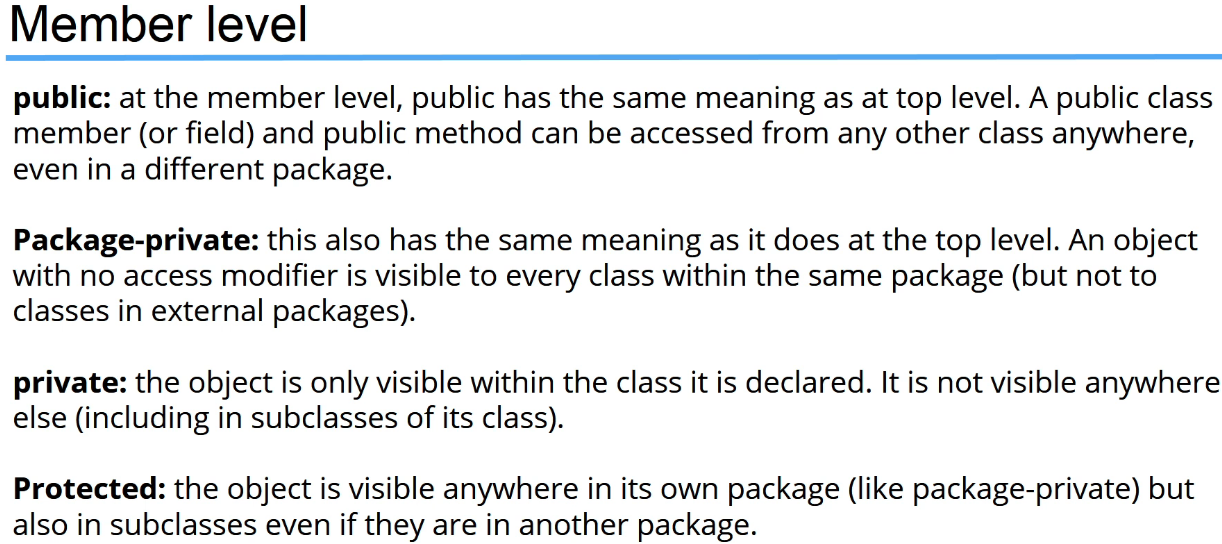
\* But the visibility does extend into the outer class, but with that said, the variable itself can’t be accessed with any class external to ScopeCheck.  
  
\* The reason we can’t do that is because varThree has private access and therefore has no visibility outside its class or a containing class if its class. So in other words: from ScopeCheck or ScopeCheck.InnerClass.  
\* If we change the varThree back to public, the error goes away because we’ve now given visibility for that variable, basically it’s public to any other class. Let’s keep it private.  
\* So this concept now is moving away from scope and into visibility and access but varOne and varThree’s visibility within the ScopeCheck is related to the scope of the variables and I think it’s important to discuss that before we move on to access modifiers sort of properly. In particular, the fact that both varOne and varThree are declared private, as you can see, but both can be accessed from both the outer and inner class, that can be surprising when you’re starting Java.  
\* Finally when we look at inheritance, we often have a subclass override a method from the superclass. So the mechanism for this is very similar to variable scope that we’ve discussed so Java will look for a method in that case in the current class and use that if it finds it, otherwise it’s gonna look for it in the immediate superclass and so on up the chain until it finds a class that has the method. But with that said, it’s not actually correct to say the overriding method shadows the one in the superclass, override is actually the correct term here, not shadow.  
\* If though the method was static, then it can be shadowed or overriden but don’t worry about that now until we get to the lecture on the statick keyword.

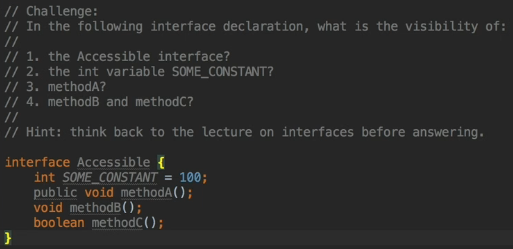
**Scope +(Challenge Exercise)**  


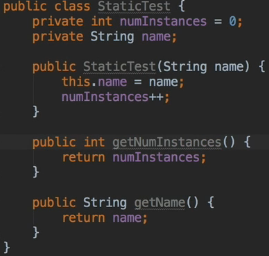
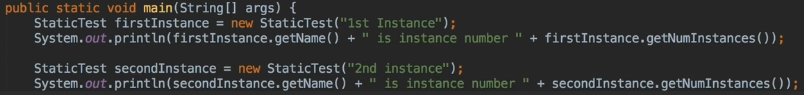


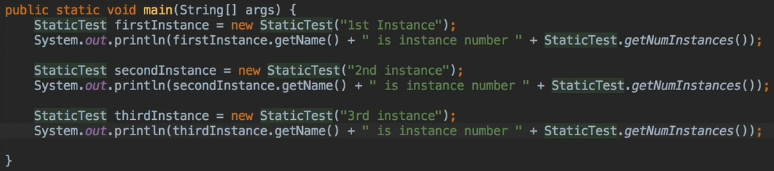
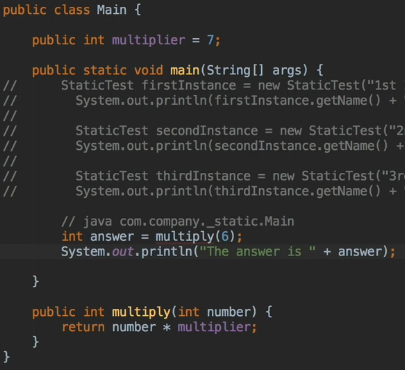
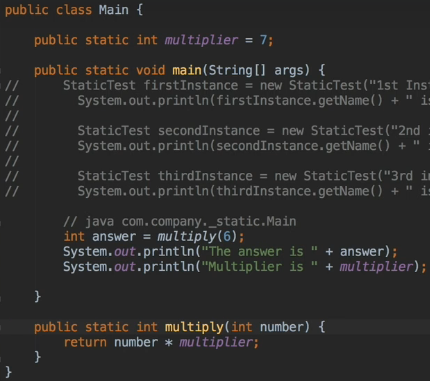
\* In Main.java:  
  
\* We’ve used x everywhere but by using the appropriate scope we’ve managed to get the program to work without it actually coming up with a problem with the x/X being considered a duplicate.  
\* Now let’s change the Scanner’s name to x as well.  
  
\* In Main.java:  


**Access Modifiers**  


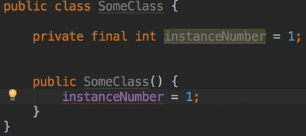
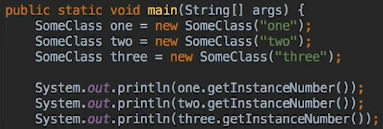
\* Let’s now test it with everything public:  
  
\* If we modify it directly:  
  
  
\* We can also modify the transactions directly:  
  
  
\* Point is if you expose fields of your classes, then anyone using the class will reasonably assume that they can manipulate the values of those fields. And because this bypasses the methods you’ve written to use those fields, your classes may no longer behave as you intended them to. So you may lose control over them.  
\* So we just need to change them to private:  
  
\* **Access control is granted at the top level or at the member level**.  
\* At the top level, what you can do is make your classes and interfaces public or packaged private. You can’t define a private class at the top level.  
  
  
**public**  
**Package-private** (**no modifier specified**)  
**1 a) public class**  
  
**1 b) public interface**  
  
**2 a) Package-private class**  
  
**2 b) Package-private interface**  
  
  
\* Member level is once we’re in the class/interface itself.

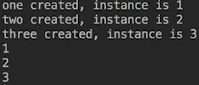
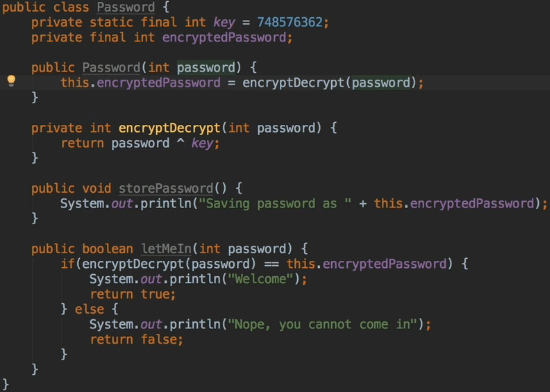
\* There’s 1 case we haven’t mentioned.  
  
\* interface Accessible => the visibility is set to Package-private so it’s accessible to all classes in the com.timbuchalka package  
\* int SOME\_CONSTANT = 100 => the visibility is set to public and that means all interface variables are **public static final**.  
\* public void methodA() => public  
\* void methodB() and boolean methodC() => public because all intereface methods are automatically public, so the lack of an access modifier here doesn’t imply Package-private so that’s a bit tricky.  
\* **It’s not possible to have anything except public methods in an interface**. This makes sense if you think about it because the whole point of declaring interface is to provide methods that have to be implemented. So if you hide those methods in any class implementing the interface it will struggle to implement the hidden methods which sort of defeats the whole purpose of using interface.  
\* **You can make the methods effectively Package-private by ensuring that the interface itself is Package-private as we’ve done here**.  
\* Although all 3 methods are public, if the interface itself is not visible outside the current package, then those methods obviously won’t be visible either.  
\* **So the lack of an access modifier means the default of Package-private, except with interface methods and variables, which arise public**.

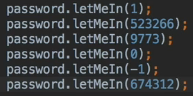
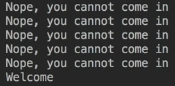
**The static statement**  
  
  

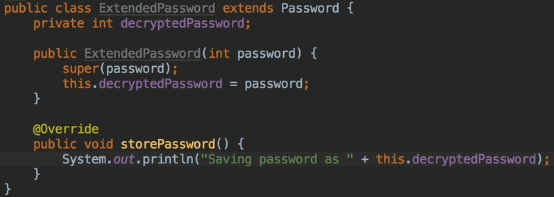
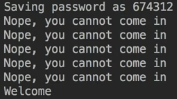

\* **But a static field, which is also known as a class variable, is associated with the class rather than with any particular instance of it**.  
\* **So it’s only ever one copy of the variable in memory. All instances of the class share that one class variable**.  
  
  
\* **If you’re using a method that only works on static fields, it makes far more sense to make that method also static, and that means that we can then access the method without having to use a class instance to do so at all**.  
  
  
\* This explains why the main method we’ve used throughout the course has to be static - because when we want to run a Java program, there has to be an entry point, in other words a method that is executed when the program runs. However, until the program runs there’s no class instances to call methods on, so Java has to use a static method that can be called from the class name rather than from a class instance.  
\* So when we run our sample program in IntelliJ, what IntelliJ does is it invokes a Java executable and passes to it the name of the class that contains the public static void main. So provided you build the project, you can run the program from the command line by changing it into the project production directory and then typing something like `java com.example.\_stataic.Main`. The Java will then expect main to have a static method called main with the exact signature that we have been using throughout the course.  
\* The main method doesn’t have to be a class called Main, it can exist in any of the classes, but the convention is to put it in a class called Main. And if we’re given a project containing hundreds of classes to work on, you will at least know where the program starts by looking for the Main class rather than having to check every other one to find the Main method and that’s why it’s sort of a standard convention.  
\* **Static methods and variables = class methods and variables**.  
\* **Now that we know about them, we can work out why all the methods we’ve created in Main have also been static**.  
\* This is a slightly contrived example but it will help to show what we can and can’t do with static methods.  
  
\* In theory the code looks good but IntelliJ’s showing an error.  
  
  
\* So if multiply is public then why can’t we access it from our main method? Well as we’ve seen, normal class fields require an instance of the class, they don’t actually exist until an instance has been created. Now the main method is static as we can see and it can be called without a class instance so as a result, Java can’t allow a static method to access non-static fields or non-static methods because they don’t exist when the static method’s called.  
\* So in many of our examples in earlier lectures, what we did was we created methods from Main to call and in order for that to work, we had to make the methods static.  
  
\* Note that there’s nothing to prevent a static method from accessing non-static fields and methods in another class because it creates an instance of a class in order to do so.  
\* So the restriction is purely on a static method accessing non-static methods and fields in its own class.

\* There’s one more aspect of static that we need to look at - and that’s static initializers. Now the discussion of these also needs to consider final fields so we’ll come back to static initializers in the next lecture once we’ve discussed the final keyword.

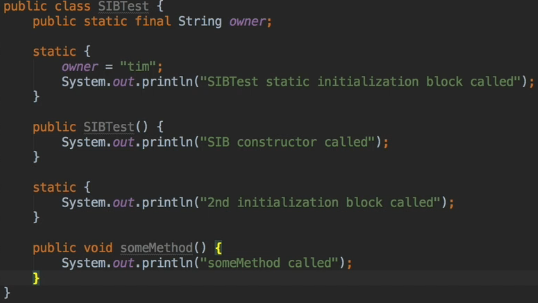
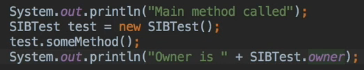
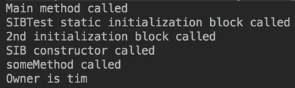
**The final statement**  
\* We use that generally to define constant values.  
\* Strictly speaking, final fields are not actually constants because they can be modified but only once and any modification must be performed before the class constructor finishes.  
\* That means that we can assign a final field its value either when we first declare it, or in the constructor.  
  
\* One reason why you might need to initialize the variable in the constructor would be if its value is a result of some calculation that perhaps relies on some other code such as a method from another class.  
\* As an example, when reading a record from a database, you might want to store the class instance’s unique database key which wouldn’t be available until you read the record from the database and then ready to create the class instance from the database field.  
  


  
\* It’s a good idea to mark variables as final whenever you know that the value shouldn’t be changed once the initial value has been set. And a constant value would be an ideal candidate for that.  
\* That said though, you may be used to seeing constant values in Java declared using **static final**.  
\* So why are constant declared static final? The reason is that if the value really is constant and won’t change, then it doesn’t make sense to store a copy of that in every single class instance. They all hold the same values so it makes sense to store it only once at the class level. So hence, values that are constant are usually declared using static final.  
\* Example: **Math.PI**  
\* It’s usual to assign the value of static field variables at the time they’re declared but again that can be done afterwards in a similar manner to non-static final fields and we’ll see that shortly.  
\* **If you make the constructor private, you prevent anyone from creating instances of that class**. The reason that Math’s constructor is private is because all the Math methods are static and they are all available without requiring instance, so to make that clear and to actually enforce it, the creators of this class made the constructor private and that prevents any instances from being created.  
\* They also marked the Math class a final.  
\* **By marking a class as final, it prevents the class from being sub-classed**, so any attempt to extend Math will fail.   
\* Marking the methods final will prevent them from being overridden which you might want to do sometimes, for example if some methods are crucial and you don’t want anyone who’s extending from it to override it.  
  
  


   
\* The problem comes if someone overrides the storePassword method in the subclass.  
\* Let’s see how to deal with that in the next video.

**Final Part 2 and Static Initializers**  
\* The problem could exist if someone overrides the storePassword method in a subclass.  
  
  
  
\* You can see how this completely changed the security of the original password class and it’s compromised it effectively because we’ve now got a decryptedPassword that’s potentially out in the wild. This is where final would be really really useful.  
  
\* Then the ExtendedPassword would get an error:  
  
\* So you can see how it’s very useful to selectively make some methods final in your classes if you know they’re gonna be overridden or there’s a chance they’re gonna be overridden and that’s also a reminder for you when you’re looking at the code, that you’ve made final for a reason.

\* When talking about static I said that there’s more than one aspect of static that we need to look at, and we needed to understand final to do so.  
\* So now we’re going to look at the static equivalent of constructors.  
\* We’ve obviously got a standarad constructor that you’ve seen throughout the course.  
\* What I’m talking about now is the static equivalent of those constructors and they’re called   
**static initialization blocks**.  
\* Constructors are instance constructors so they’re not static, so they’ll be executed every time we create a new instance of a class.  
\* The static equivalent of that is a static initialization block.  
\* **The difference there is that block is only executed once when the class is first loaded into the project**.  
\* So it’s not often at all that you’d need to create a class constructor which is sort of what a static initialization block is even though that’s not really the correct term for it.  
\* The reason I’m mentioning it is because the static final variable must be initialized by the time all static initialization blocks terminate.  
\* So in the same way that we can set the value of a final field in the constructor, we can also assign the value of a static final variable in the static initialization block.

\* Static initialization blocks are an advanced feature and there’s rarely a case for you to use them.  
\* **static {}**  
\* There can be as many static initialization blocks as you want.  
  
  
  
\* All static initialization blocks are called before the constructor. In fact, they’re called before any non-static metehods including the constructor.