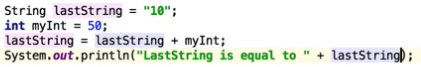
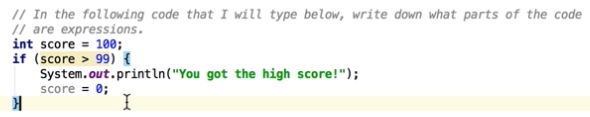
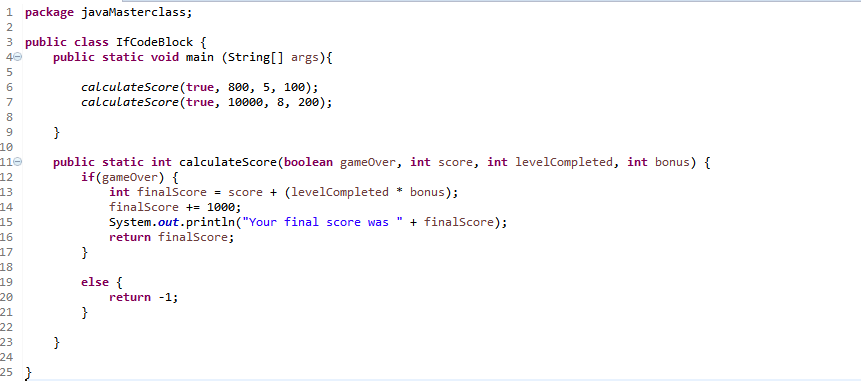
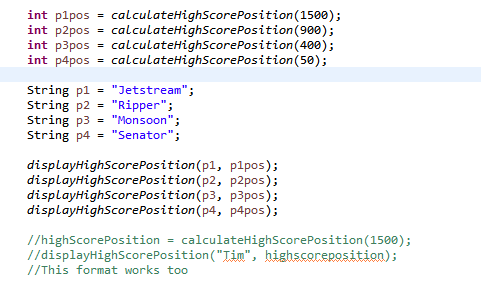
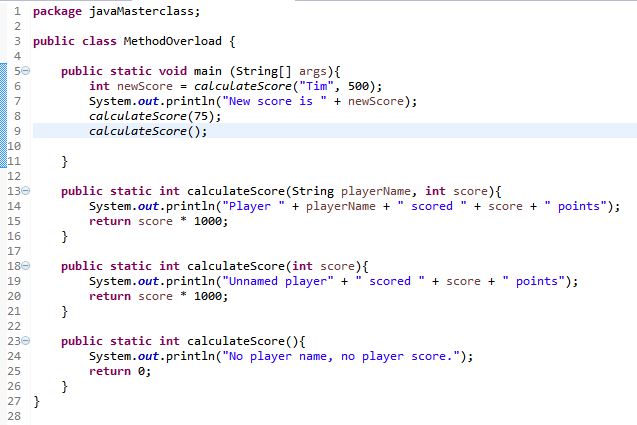
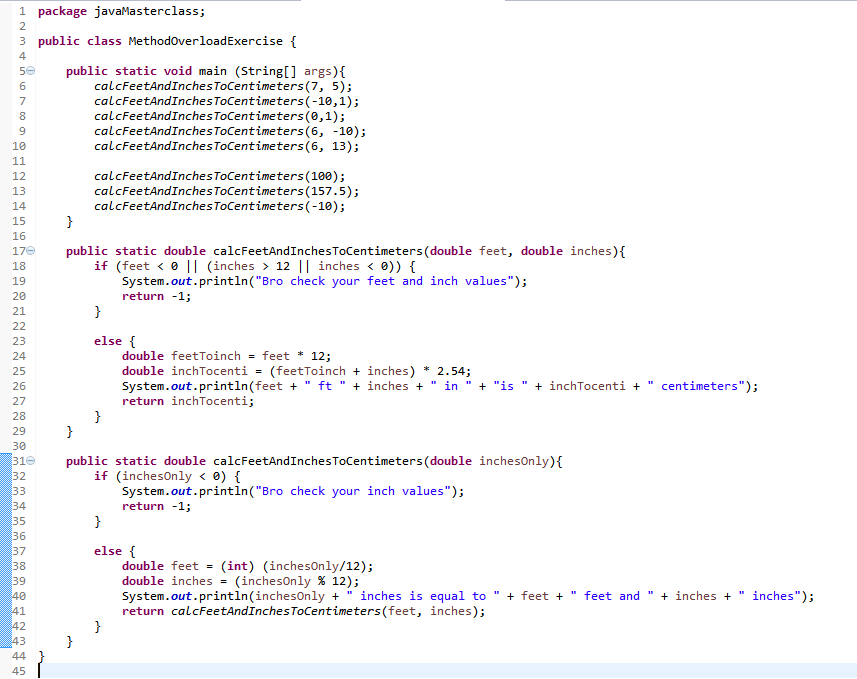
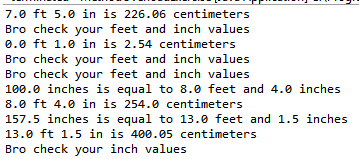
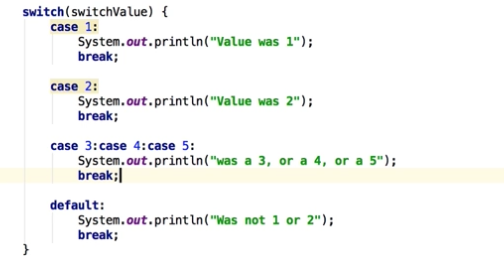
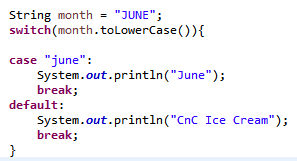
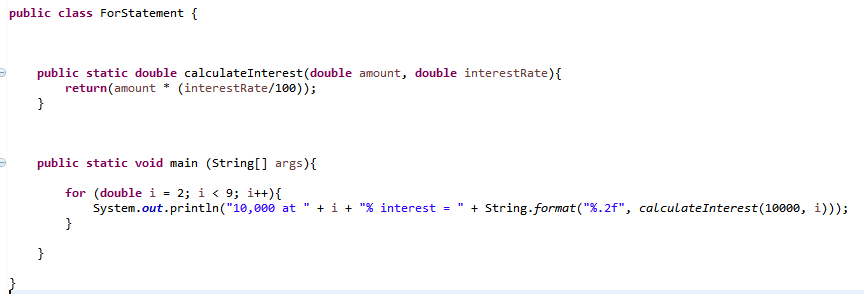
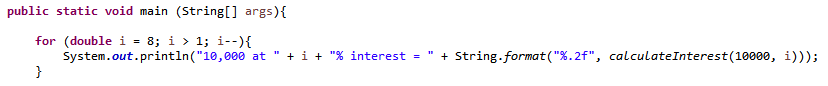
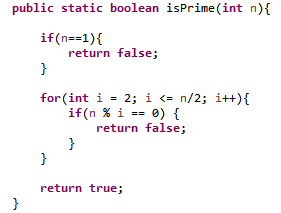
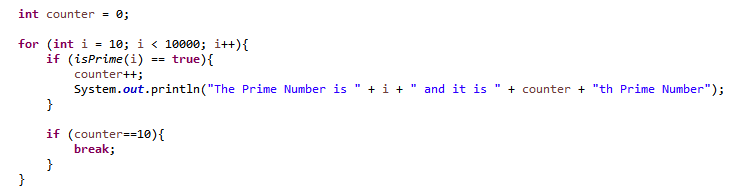
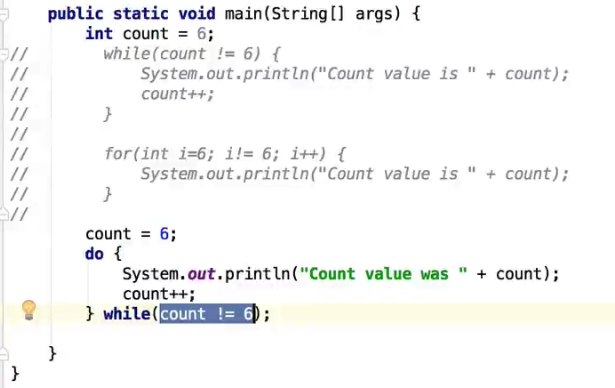
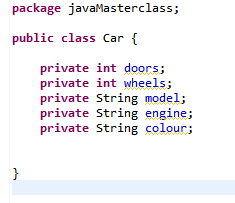
Variable – Place in memory, a box that can be stored with a type of information. Makes program useful.  
To assign variable, declare type, name, and value.  
Creates that variable in memory location.  
**int myFirstNumber =** value, mathematical expression including parenthesis, etc.  
**System.out.println(myFirstNumber vs “myFirstNumber”)** -> Difference between value of int vs. String.  
  
  
**Primitive Data Types**  
  
Every data type has min max boundaries, whenever it is exceeded, it will come out to an error.  
  
Value is literal when it is a direct number.  
Value is expression when it’s something like (a \* b)  
  
Starting from JAVA7, you can use underscore between literal numbers to make it more easy to read.  
  
Byte – 8 bit  
Short – 16 bit  
Int – 32 bit  
Long – 64 bit (put L at the end of the number)  
  
By default, JAVA uses int data type in expression.  
**byte myByte = -128  
byte myNewByte = (myByte / 2); //** will give error, because JAVA assumes answer is integer.  
**byte myNewByte = (byte) (myByte/2);**  // Overrides this issue.  
This trend is same for byte, short. If you don’t override the assumption.  
Since long is 64 bit and Integer is 32 bit, long accepts integer.  
  
Float – 7 digit after decimal point, 32 bit  
Double – 16 digit after decimal point, 64 bit  
Same as long, JAVA assumes double for default decimal point.  
Same as long you can put f or d after number, **5d, 5f** to indicate data type.  
Double is actually faster on modern computer, built in math function use double, more precise.  
Underscore can also be used to make literal values easily readable.  
  
char – 16 bit, one letter or one character in single quote, also accepts UNICODE with backslash.  
**char myChar = ‘\u00A9’;** will print out the copyright symbol.  
Boolean – 1 bit, true or false  
  
**String –** Sequence of characters, not really a primitive data type. It is class.  
String can get appended.  
**String myString = “This is a string”  
myString = myString + “, and this is more.”;**will print out “This is a string, and this is more.”  
Unicodes also work with Strings  
  
  
  
This will work, the JAVA will convert myInt to String, the output will be 1050.  
Same, if you append another random double value to lastString, the output will be string values.  
Strings are flexible, can delete character, append character, insert at specific position, et cetera.  
  
  
  
Typical Operators  
  
Plus, Minus, Multiply, Divide.  
% - remainder operator  
**int result = 4;  
result = 4 % 3;**output is 1  
  
++ - Increment operator (x = x+1)  
-- - Decremental operator (x = x – 1)  
  
result += 2; means, result = result + 2;  
result \*= 10; means, result = result \* 10;  
  
= is assignment operator, sets a value  
== is tester operator  
!= is not equal  
  
&& and operator  
|| or operator  
  
Ternary operator  
result = testCondition ? value1 : value2;  
“If testCondition is true, assign the value1 to result, otherwise assign value2 to result.”  
  
Data type is not included in the expression, everything else on the line typically forms or is part of expression.  
Component of expression – Variable, value, and operators.  
Add data type to expression, valid JAVA statement  
  
  
  
Expressions are  
score = 100; , score > 99, “You got the high score!” , and score = 0. Semicolon not included.  
  
  
If statement, when only one statement after the if, does not really require bracket, but anything after the first statement will be regular line of statement.  
  
Using brackets makes it a code block. Should always use code block for consistency and readability.  
  
Any declaration / Assignment inside if code block cannot be used outside of code block, because if statement is conditional and JAVA does not recognizes it outside of the codeblock.  
  
if(gameOver) is same as if(gameOver == true)  
  
return must be accounted for everything, so if only in a if loop, it will cause an error since it is conditional



Instead of repeating if code blocks, a calculateScore method was created with 4 variables passing through the method. The return type was set to int and is returning finalScore and -1 if gameOver is false.  
  
   
Values returned from a method can be directly assigned to a declared value, see ifCodeBlock.class for more info and the method lecture 30 from section 5.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**Overloading Method** is using the same method, but with different number of parameters.  
Unique method signature is required when overloading method. This means changing the number of parameters, changing the data type of return does not make unique method signature. So change from int -> void will not work, but changing parameter will work.  
  
  
  
  
  
**OUTPUT** **This is method overload exercise for last part in section 5**

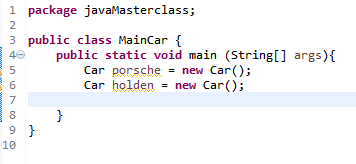
**Switch Statement –** Similar to if statement, but good when testing multiple variables.  
  
  
**Switch** statement, we are only testing for   
different value of same variable.  
**If statement**, variables can be different.  
Switch when testing same variables with   
different values.  
  
Switch can be used with 4 primitive  
date types.  
  
**byte, short, char, int**When using char, case ‘A’ with quotes  
  
  
  
To avoid making case for every single  
combination of upper and lower case  
of a name of a variable, we can  
use toLowerCase() to make everything  
into lower case and make one case  
for the lowercase name of the variable.  
  
  
  
  
**Structure of for loop**for(initialization; termination; increment) { }  
  
**String.format(“%2.f”, arguments)** will limit the decimal point to 2 place after the dot.  
  
  
  
This will start from 8% and end at 2%.  
  
  
  
  
**Pre-defined code method for prime number**  
  
**For statement challenge with prime number method**  
  
  
**If** initial condition not met, while and for loop will not initiate at all.  
Do-while loop will always run at least once up until the condition is met.  
  
  
Example of while, for, and do-while loop in similar fashion.  
continue; is opposite of break, it will go back to the initial loop instead of breaking out.  
**Object – Oriented Programming**Class – template for creating objects  
Objects – has state and behavior  
  
Class – “powerful user defined data type”, you can create something more powerful than primitive type.  
  
Local variables are inside methods, cannot be accessed outside of the method.  
  
Encapsulation – internal workings of object does not allowed by outside access.  
  
Private – does not allow access from outside of the class.  
So, Porsche.model = “Carrera”; from main will not work if the field (instance variable, state) model is defined as private String model. It has to be changed to public in order for this to work.  
  


In the prime number for loop

(long) Math.sqrt(n); can be used instead of n/2 to optimize the looping.  
  
21 loops used when n/2  
6 loops used when using sqrt(n)  
  
Long is casted since Math.sqrt(n) outputs lots of numbers.

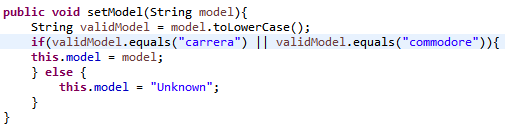
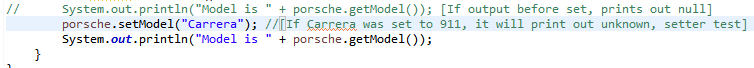
In the main method, two car object using the car class template is created called Porsche and holden. Notice that variables inside the main method is local, while the upper private variables are the instance variables.

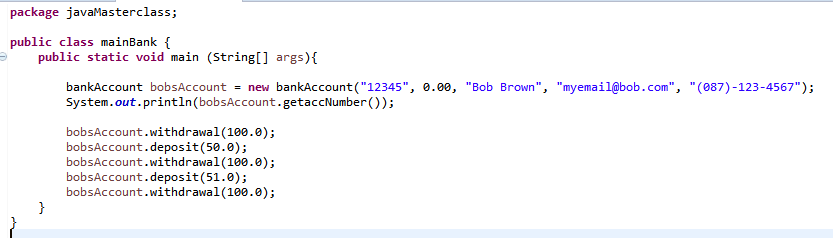
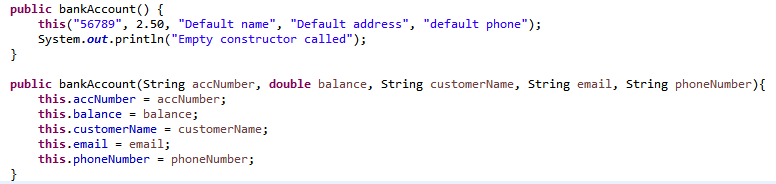
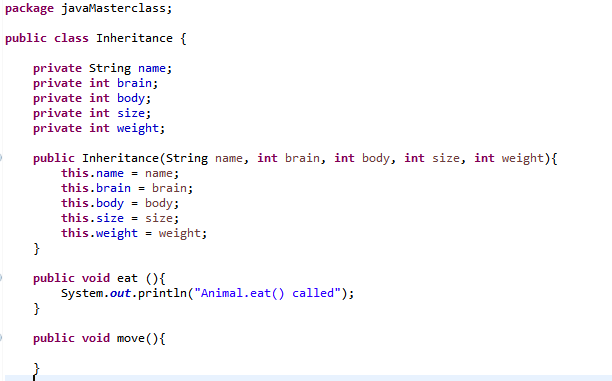
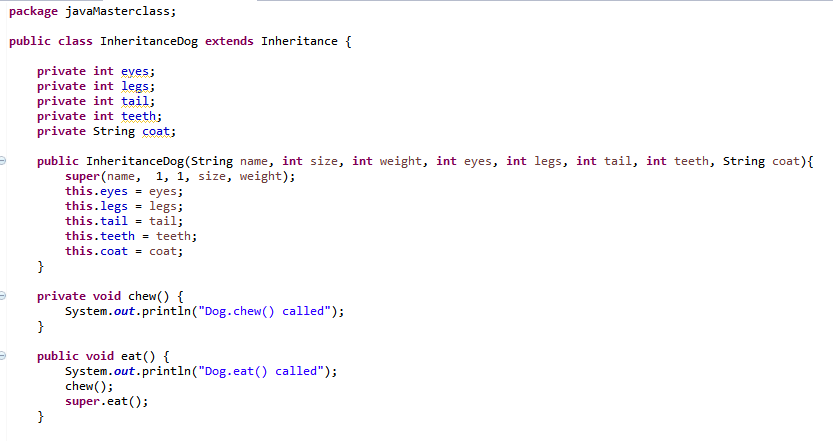
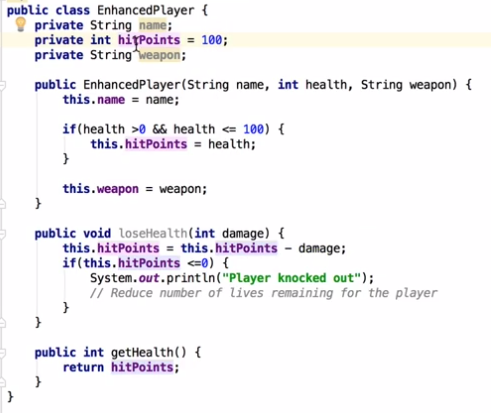
Fields, (Instance variables), the “state” of an object is all declared as private to follow encapsulation.



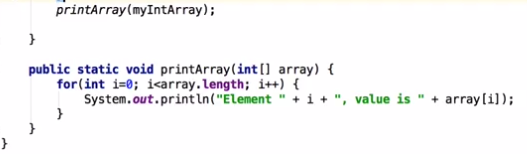
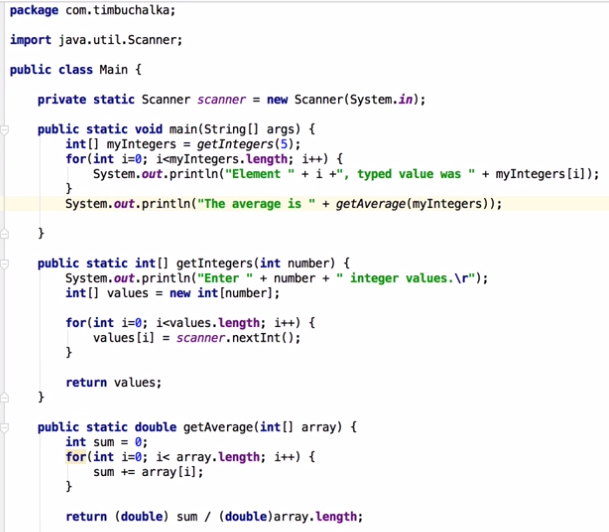
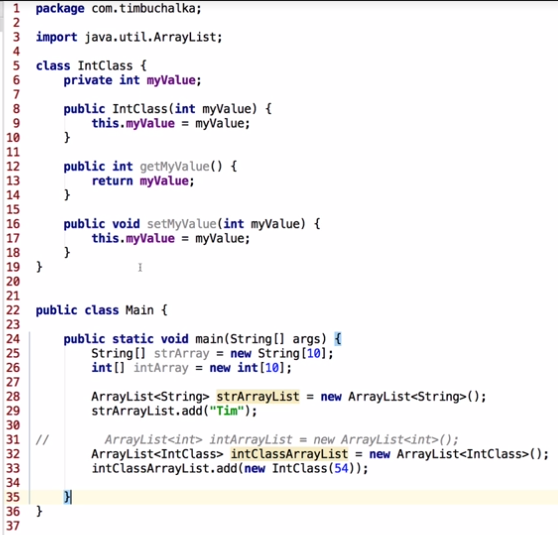
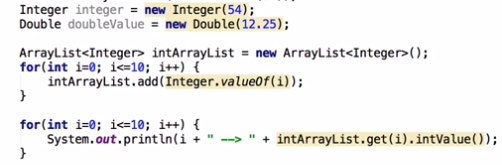
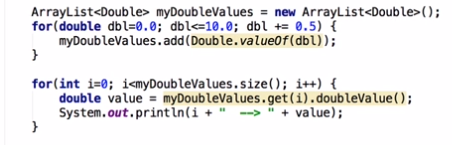
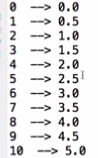
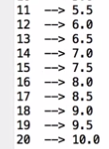
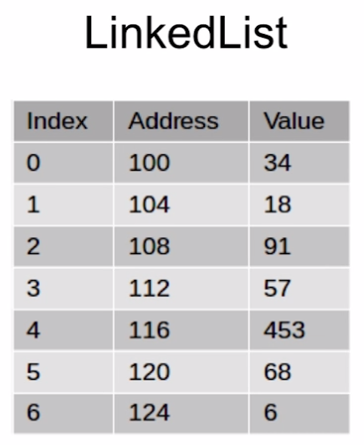
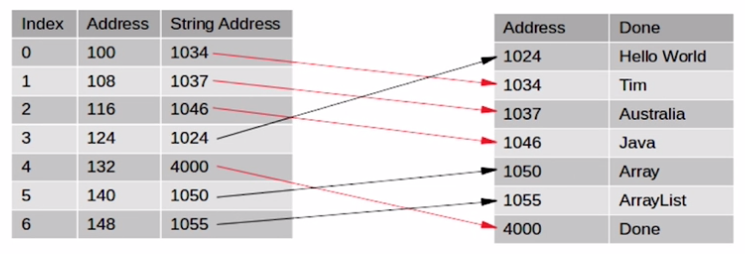
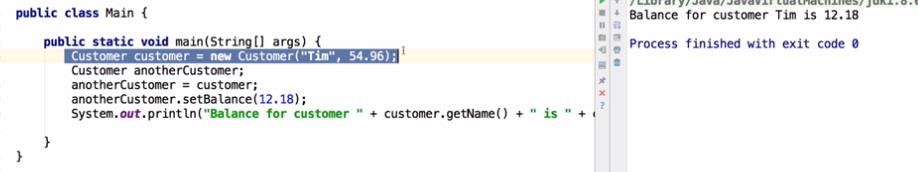
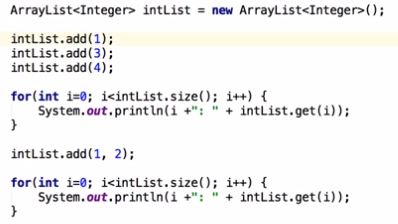
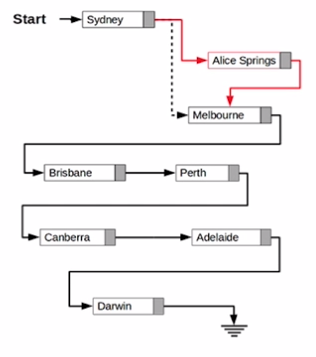
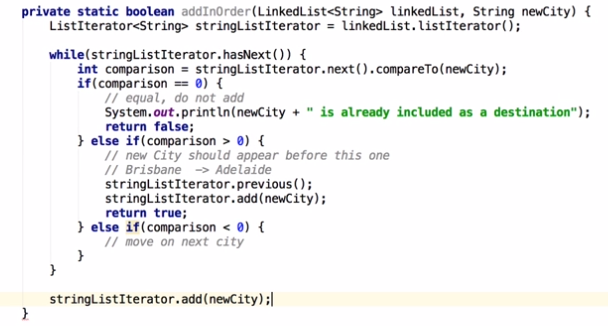
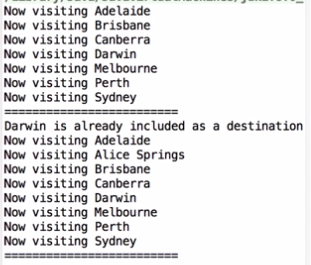
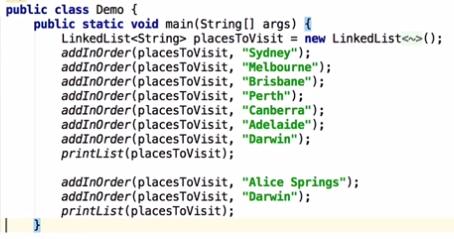


This.model refers to the field model  
model refers to the parameter model in method

Unlike data types, classes doesn’t have default value, so you have to initialize it by doing **new Car ();**  
  
  
  
  
  
  
  
  
  
  
  
**Concept of getter & setter**Can’t assign invalid values when creating models and assigning fields.  
  
 **In the main method,**

**Constructor** allows you to essentially initialize the object you are creating and do whatever else you want to do at the time you are creating the object.  
  
Like methods, constructors can be overloaded.  
Since constructor can be overloaded, the default value can be set like this  
  
  
  
When bankAccount bobsAccount = new bankAccount (); is were to be written, the values will follow the default value in the bankAccount() constructor. If parameters are specified, they will remain specified. If not, they will be default value.  
  
this(….) statement must be the first line in the empty constructor.  
  
In constructor, instead of assigning directly, the previously created setter and getter can be used, but due to inheritance it may not work sometimes, so rule of thumb is to assign directly as shown above.  
  
  
  
  
**Another constructor overloaded with 2 default value and 3 parameter**  
See my javaMasterclass package for 3 class file related to learning constructors.  
  
**Inheritance**This is the super-class.. for “Animal”.  
  
This is inherited class..for “Dog”, dog’s own fields are declared, but it is extended over the original “Animal” class. The constructor includes inherited constructor parameters as well as the new parameters. The eat method is overridden.   
**Super.**------ will always refer back to the method or constructor in the original “Animal” class that inherits.  
JAVA by default, inherits from the java.lang.object class, it is automatically extended.  
Superclass – vehicle, extends to car, “Car is a vehicle”  
 **Composition –** modeling parts, parts of the greater whole.  
in JAVA, you can only inherit from one class at a time.  
In the case of CompositionPC class, it is comprised of Composition case, monitor, and motherboard.  
  
**Composition vs. Inheritance**“Putting together” vs. Animal to Dog.  
Composition – creating objects within objects.  
  
Generally, look at using composition first before inheritance.  
  
**Encapsulation**Mechanism that allows you to restrict access to certain components in the objects you are creating.  
Protect members of the class from outsider (class or code outside of the class you are working on)  
Not using encapsulation opens the application to unwanted access that may change the important part of the program.  
Also, every time any variable field name or etc is changed, since it is directly accessed, the error will shift to another class that was accessing it directly.  
Lastly, using constructer guarantees valid data sets that can be worked, directly, the data may not be valid.  
  
This is without encapsulation, when the String name is changed to fullName, the main class gets an error as well because variable is accessed directly.  
  
This is main class without encapsulation, health can be accessed directly, name is showing error, etc…  


This is player class using encapsulation.  
  
Validation is added in constructor, if validation is failed, it will be set to default value.  
  
Only way to access it is through the getter, this is why the name hitpoint change doesn’t matter in the main class code unlike the one without encap.

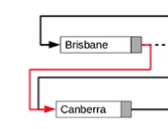
**Polymorphism** – check the source code printouts  
  
**Array –** Data type structure that holds multiple values, to construct, use [ ] square bracket after the data type to indicate.  
All Array starts at position 0, so array with 10 element starts from 0 and ends at 9.  
  
Short cut in JAVA to store values into array.  
**int[] myIntArray = { 1,2,3,4,5,6,7,8,9,10};**need to do this at the very first assignment, the [10] (10 element) is replaced by each values inside the { } bracket.  
**int[] myIntArray = new int[10];** will make 10 element array, but without each values inside the array.  
In this case, you have to assign values **myIntArray[0] = 45;** like this for each element for each index.  
Third way to initialize is using **for loop**, using index for each array and assigning values.  
**ArrayName.length** will tell you the length of the array.. you can use it as parameter in the for loop to avoid hard-coding.  
  
  
Creating a method for printing array and using that method.  
  
  
  
Another common use of array, is to ask user input to make an array.  
Scanner is the input, \r is for using the next line while inputting.  
**getAverage** is created method that takes that 5 input and returns a double average.  
  
**See ArrayChallenge.class** for getIntegers, printArray, and sortArray methods  
  
**See ArrayResize** for resizing array using basic method without using ArrayList.  
  
**List –** very much like arrays, allows you to have items in respective positions, Arraylist inherits from the list.  
  
**ArrayList –** Resizable array (resizes automatically), maintains size of elements and also capacity (amount of memory assigned), don’t have to worry about it, it figures it out automatically.  
  
With ArrayList, you don’t define the type unlike Array, for ArrayList it holds objects so you assign it like  
**private ArrayList <String>** elements in this array list is going to be String data types.  
ArrayList is class and can have constructor between ( ).  
  
  
  
  
  
  
  
  
**ArrayList** - <String> does work, but <int> does not work because primitive type is not a class and ArrayList needs class reference.   
  
One way to go about this is to create a simple int class to mimic the primitive type.  
  
  
  
Simple intclass is created with constructor, getter and setter, ArrayList is created using that intclass and addes element to the ArrayList using the int class.  
  
**Autoboxing –** “Wrapper”, converting a base primitive type into a class (**int to Integer)**  
**Unboxing –** Taking from the class to primitive type (**Integer to int)**  
**int** is primitive type, **Integer** is a defined class.  
**double** is primitive type, **Double** is defined class  
  
  
**When** using class, you have to declare and assign it using new keyword.  
  
**Integer** integer = new **Integer (54);** // This is declaring new integer class and passing 54.  
**JAVA** has a shortcut, **Integer myIntValue = 54;** // will work, because this is equal to **Integer.valueOf(54);** when compiling.  
Consquently, **int** myInt = **myIntValue** // also works, this is same as **myIntValue.intValue();**  
**Example of using Double without the JAVA Shortcut**    
**The Shortcut is making  
myDoubleValues.add(dbl);  
double Value = myDoubleValues.get(i);**  
Since JAVA is doing the Long way when compiling, for the user.  
  
  
  
  
  
  
  
**For String,** 8 bytes are allocated for each String. However, it points to another location in memory where String is. This allows JAVA to keep a track of actual String in the array, and the strings can each be variable lengths.   
  
  
  
  
  
  
**Customer** class simply has a name and double field.  
This creates one customer class called **customer** and **anotherCustomer** just points to the **customer** class internally. No second class is created. So when anotherCustomer’s setter was used, the pointed **customer** class changed and this is why the output is displaying 12.18 instead of 54.96.  
  
  
  
  
**Linked List** **Sample Code Using LinkedList fundamentals**  
 **List Iterator is** special type of iterator that enables you to go to previous listiterator and modify the element.  
  
  
This is new method created to take in the string newCity, compare it and organize it alphabetically.  
Using While loop, we are going to go through all entry in the linkedList.  
  
**JAVA** implemented linked list as double reference, its able to go forward and backward.  
hasNext() and hasPrevious()….. check source code for more info.  
  
**In ListIterator** there is no current element, this is to avoid infinite looping in LinkedList  
To avoid this, we need new variable for tracking the direction, check source code for more info.  
  
  
  
  
  
  
  
  
  
  
  
  
  
**Interface** – specifies methods that that a particular class that implements the interface must interface.  
Interface itself is abstract, no actual code exists.  
  
Provide common behavior that can be used by several classes, standardize the way particular set of classes are being used.  
  
Interface is a “Commitment” that the method’s signature and variables will not change. If it changes, any other thing that uses that method will cause an error because things changed.  
  
JAVA Libraries make extensive use of interfaces, because of that, we can change entire implementation of code (EXAMPLE is LinkedListChallenge), we can simply change from LinkedList to List and the code will work.  
  
Because of signatures being specified in the interface, as long as we create a class that implements these methods and create the methods, things will work. Meaning, since ArrayList, LinkedList, Vector implements the List interface, we can exchange freely between those that implements the List interface and this will still work without breaking.  
  
So when using any type of list, it is generic to use List, use generic list as a type and then specify the specific type  
**(EXAMPLE: List<Song> playlist = new Vector<>(); )  
  
Deciding between Implementing and Extending… look at the final method.**  
Both DeskPhone and MobilePhone is telephone, but MobilePhone is more like a computer that uses phone interface. This case, extending phone to both classes doesn’t make sense compared to using implementation.  
  
**Also,** Inheriting only works from one superclass to another class. However, implementation can be from many interfaces.  
  
**Using Animal Class from Inheritance lecture,** since not all animal can walk, we cannot put a walk method in the animal base class. But we can define walk interface. In this case, a dog will extend animal and implement walk. A bird will extend animal and implement both walk and fly.   
  
Dog and Bird is both animal, so they inherit, but their implementations differ because dogs walk and birds walk and fly.  
  
**Consider relationship of final class** to the object you are extending or implementing.  
  
**4 Types of classes in JAVA**  
1. Static nested classes

**intArrayList.add(Integer.valueOf(i));**  
This is autoboxing, converting i (int) to value in Integer  
  
**intArrayList.get(i).intValue())**  
This is unboxing, converting value in Integer to **int** value.

JAVA allocates 4 bytes of memory for each integer, since the highest value takes up to 4 byte of memory.  
  
There is a formula for JAVA allocating physical address.  
  
Looking at the base address, we can access the list using the base address formula  
(100 + 4 \* index). This is how JAVA handles it internally. Double uses up to 8 byte.

OUTPUT   
0: 1 First time, it displays 1,3,4  
1: 3 Second time, value 2 is  
2: 4 added to index 1  
0: 1 So other 2 are moved down  
1: 2 Same happens for remove  
2: 3 Others will move up  
3: 4 This will take up lots of resources

When the ArrayList contains lots of info.

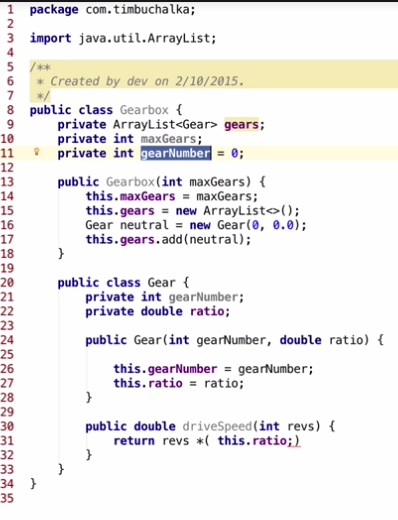
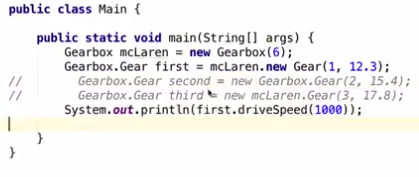
Originally, this linked list, **Sydney** points to **Melbourne** points to **Brisbane..etc.**  
  
If we want to add Alice Springs after Sydney, we just have to edit the list as shown on the left. There is no need to move every element one step down like the ArrayList requires to do. This is easier and why **LinkedList** is useful.  
  
We can also remove elements like this, in this case Brisbane will point to Canberra and Perth will automatically be trashed during garbage collection in JAVA

2. Non Static nested class – inner class

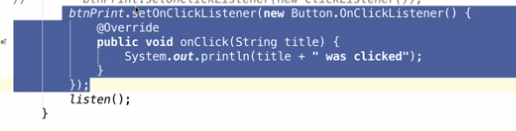
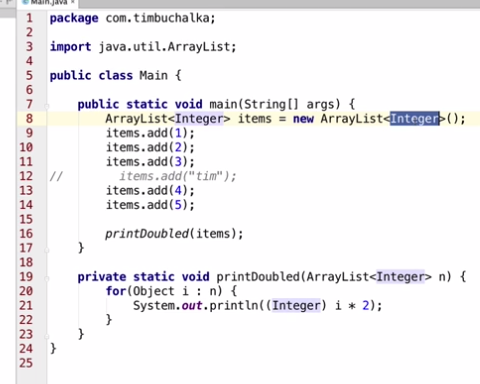
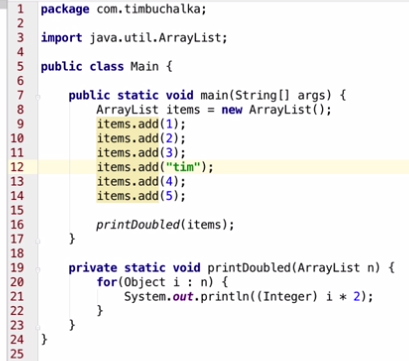
3. local class –inner class defined inside scope block (method)

4. Anonymous class – nested class without name.

**Static Nested Class**

Mainly used to associate class with its outer class. Identical to a top level class, but it is packaged in outer class. It cannot access non-static methods in other classes.  
  
  
  
  
  
  
  
  
  
  
  
  


Example of non static nested inner  
  
The inner class “Gear” is only useful when paired up with Gearbox. On its own, it is useless, this is why we can use it as inner class.  
  
When two variables are the same (gearNumber), using this.gearNumber in the inner class refers to the gearNumber in the inner class, not the outer class.  
  
We can use Gearbox.this.gearNumber, but the best way is to name the variables differently so that the variable does not get shadowed by the inner class.  
  
When using inner class in main, **.new** has to be used to declare the inner class, see example below.  
  
Conventionally, inner classes are private and cannot really be accessed from outside, so the commented out ways cannot be used to declare the element in the inner class.  
  
OUTPUT: 12300 (drive speed method)

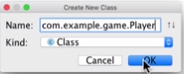
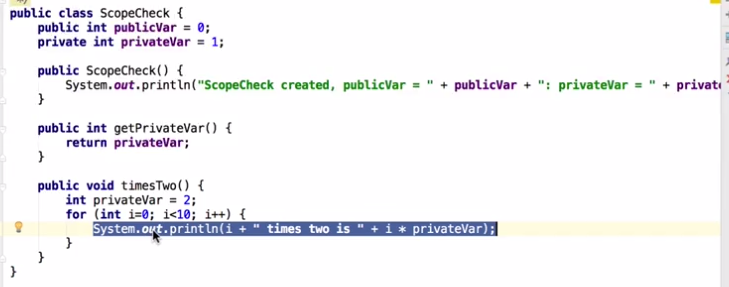
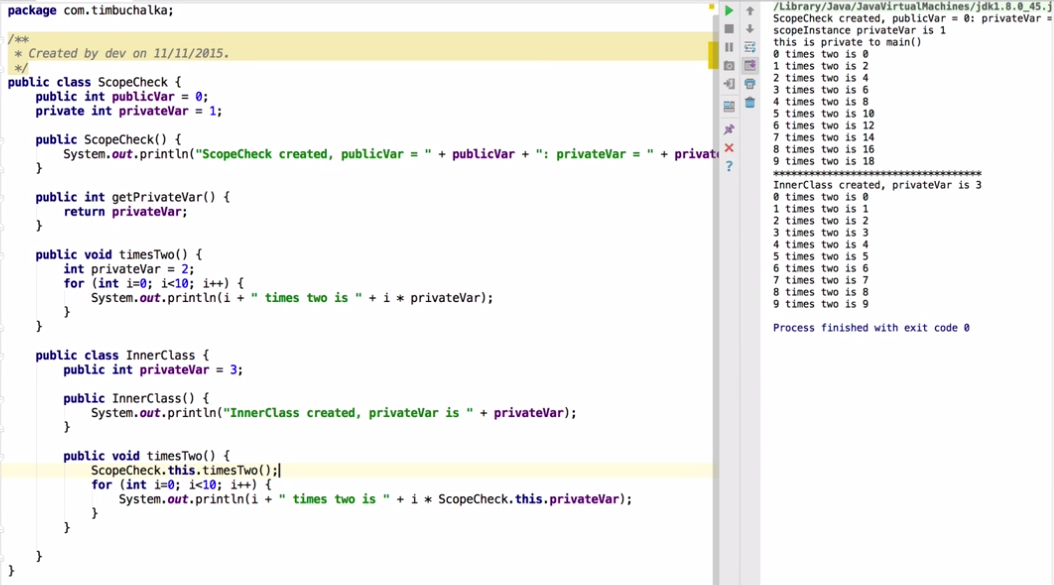
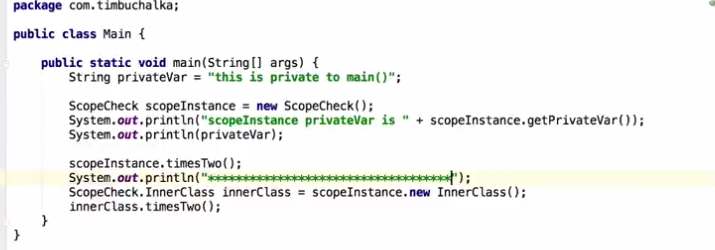
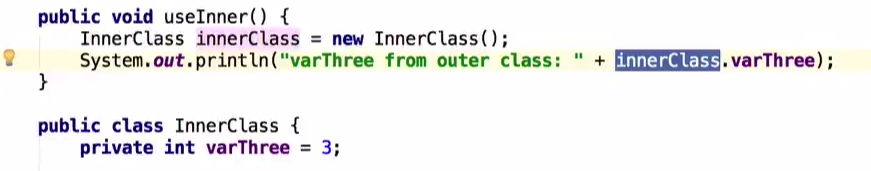
See Inner class package for more code from future lectures.  
**Local class** (Inside scope block, method) – is practically not used…. See lecture 77 for more details  
**Anonymous class** – used when local class is required just once since there is no name. Anonymous class is only implemented by a single line and between brackets..  
  
**For Inner Class Challenge, We modified the Linked ListChallenge, so check it and compare it, ONLY ALBUM CLASS was changed but the working is the SAME.  
Abstraction**When you define the required functionality for something without defining the data house.. focusing on what needs to be done not how it has to be done.  
  
Using abstraction for flexibility and leave implementation to individual classes (List interface..classes can use every list”  
  
Using Abstract class forces the sub-classes to specifically use the methods that are specified in the abstract classes.  
  
Abstract methods have to be used (implemented).  
  
With abstract classes, we can use regular fields and methods along with abstract methods, so we are not restricted to only using abstract methods that has to be implemented.  
  
Abstract class can extend from another abstract class (Bird class in Abstract package), but it cannot be instantiated using **new**, because it is abstract. In this case, you have to create a new class extending from the abstract class.  
  
Also, it can override the methods, but if the abstract class that it extends from did override, it is not necessary.  
  
  
**Difference between Abstract class vs. Interface**Consider relationship,  
  
Abstract class can have member variables that can be inherited, can have constructors, can be private, can have defined methods, methods with implementation.  
  
Interface can have variables, they are all public static final, constant variables that cannot be changed, cannot have constructors, are only public, all method in interface are abstract, all the code has to be written in the class that implements the interface.  
  
**Detail on BINARY SEARCH TREE – See Final Lecture of Abstract Challenge** **We can create out own generic class… check the code for full detail  
Check Lecture 86 for explanation on class & multiple interface…if needed**

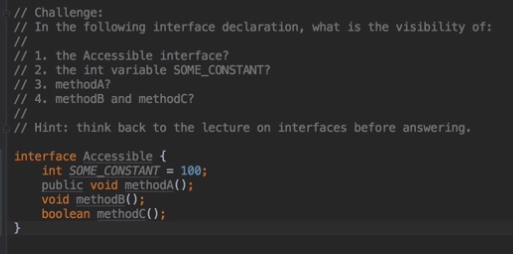
Code without using generics (Type Parameter)   
  
This does not show any error and shows up as valid, but when it is compiled it will throw exception and crash, because String cannot be casted into integer.

This is example of using generics, using <Integer> to pass a type parameter.  
  
**new ArrayList <> () ;**   
**SHOULD** be used, it is the correct way because it is explicit that the ArrayList is Integer.  
  
**< >** This is called “Diamond”

**JAVA Naming Convention  
  
Packages**Internet domain name first, followed by the package name -> com.timbuchaka  
Replace invalid character (Java keyword, etc) with underscore.  
Class & Interface name conflicts are inevitable, mechanism to allow use of classes with same name in the same project.  
  
We can import package, or for one type use, we can use something like…..  
  
There cannot be two import for same named class….  
  
switch.supplier.com -> com.supplier.\_switch  
1world.com -> com.\_1world  
 **Classes**Should be nouns, each word should start with a capital letter (LinkedList) **Interfaces**Consider what objects implementing the interface would do (Comparable)

**Methods**Often verbs, reflect the purpose of the methods

**Constants**Separate words with underscore, all uppercase, declared using final keyword  
 **Variables**start with lower case letter, do not use underscores. **Type parameters**Single, capital letters.  
E – Element  
K – Key  
T – Type  
S, U, V – 2nd, 3rd, 4th types  
  
**Import java.awt.\***Asterisk will import all static class and interface from the package.  
Asterisk does not contain another package or sub-package  
Look out for same class name when using star as it auto imports.. it will create error in compile.  
  
 You can create new class in a new package  
  
**To export classes into .jar file and importing it, see lecture 92 to see how Interface Challenge project was exported out as .jar file and used by a single main class after importing that .jar file  
  
  
  
  
  
  
  
How to export out to .JAR file and import it to use it as a library (Package Challenge)**File -> Project Structure -> Artifacts -> Add -> JAR -> From modules with dependencies   
Build -> Build Artifacts -> Select JAR -> Build (Now its built, check the folder directory for created JAR file)  
  
Now, create new project that is going to import this series class.  
  
File -> New -> New Project -> so in new project, we can make a new separate main  
  
Project Structure -> Libraries -> Add created JAR file as library (in artifacts folder)  
  
Go to external library, check if the JAR is added as library.  
  
When using the imported method (Sum, Fib, Fac), it will automatically import that for you.  
  
**Scope – Granting & Restricting Access to objects**Scope – refers to visibility of a class, member, or variable.  
Rule of scope -> Variable with the narrow scope is the one that is used… the most local one  
So, if there is a **private int private =1;** in the class and **int private = 2;** in a method inside the class,  
The rule of scope ensures that **2** will be used for that method, because **int private = 2;** is the most local  
  
JAVA starts by checking current block of code for declaration, if there is none, it checks the outside block and this repeats until it reaches the most outer block.  
  
  
  
This case, it starts inside the **for loop**, nothing there, so it goes out to the **timesTwo** block, sees the privateVar and uses that variable.  
  
Similarly, **int i** is only declared inside the for loop and the scope is limited within the for loop. So if you try to use int i outside of the for loop, it will cause an error.  
  
If **int privateVar = 2;** does not exist, JAVA will search outside the method block and see the initial declaration of 1.  
  
**this.privateVar** will use the variable with value of 1.  
  
  
  
  
**Example code from first scope lecture  
  
Incorporates Inner Class.**if **privateVar = 3;** is commented out, JAVA will use **privateVar = 1;**  
  
To use the **privateVar = 1;** we can use **ScopeCheck.this.privateVar.**  
  
Commenting out **privateVar = 3;** and using **this.privateVar** in InnerClass will show error, because privateVar is not declared. This is why **ScopeCheck.this.privateVar** has to be used..  
  
Everything else is self-explanatory.  
  
scopeInstance.timesTwo(); -> calls method in the ScopeCheck class in the inner class.  
  
  
  
  
  
Since **InnerClass** is contained inside the **ScopeCheck** class, **varThree** is visible and accessible in the upper **ScopeCheck** Method by using example like that.  
  
**See scope package** for full detail on lecture code.  
**Review qualifiers and dot operator usage (See scope lecture code)  
  
Access Modifiers**

**Top Level –** Classes, Interfaces, Enums can exist, everything else must be included within one of these  
  
**Public –** visible to all classes everywhere, whether they are in same package or imported the package.  
**Package-Private –** object available within its own package, visible to every class within same package, it is specified by not specifying, (Default)  
  
**Member Level –** Inside the top level  
**Public –** same definition as the one in the top level (visible everywhere)  
**Package-Private –** same definition  
**Private –** only visible in the class it is declared, it is not visible anywhere else (including in the subclass of its class)  
**Protected –** object is visible anywhere in its own package, but also in subclasses even if they are in another package.  
  
  
  
**Mini Challenge**  
  
  
1. Package Private  
2. Public static final (interface are automatically public)  
3. Public  
4. Public static final (interface are automatically public)  
  
  
  
  
**Static –** only one version of that exists in memory at any given time, it belongs to the **class,** not to the **instances** of the class, as a result can be called by referencing the class name rather than the instance name.  
(Instance name is like.. **SampleClass firstSample = new SampleClass(); ..** in this case, **firstSample**)  
  
**Restriction –** Purely on a static method access non-static methods and fields in its own class, because technically the instances does not exist. The reverse or different class does not matter, because it does exist.  
  
  
  
**Final –** The variable’s value can be only changed once, hence at the initialization. Constants are usually used as static final, because it doesn’t make sense to make final variable store in every single level of the instance.  
  
**Constants –** are usually static final & Uppercase  
  
Declaring a class as **final** prevents that class from being used as subclass.  
  
Marking methods as **final** prevents that specific method from being overridden.  
  
Example of this is shown on lecture 99 & 100 on encrypt & decrypt example.  
  
**Constructors -**  are instance constructors, instances are created every time we run.  
  
**Static Initialization block –** ran only once when the project is loaded, declared using Static keyword, these static initialization blocks are called before any non-static methods including the constructor