Chapter 4: Methods and Encapsulation.

Designing Methods.

Method declaration refers to all the parts of method, the information needed to call it.

```
public final void nap(int minutes) throws InterruptedException ( // take nap }
```

Where "public" is the access modifier, "final" is the optional specifier, "void" is the return type, "nap" is the method name, "minutes" is a parameter and "InterruptedException" is an exception declaration / handling.

Access Modifiers.

Java offers 4 choices:

- *Public*. The method can be called from any class.
- *Private*. The method can be called only from within the same class.
- *Protected*. Method can be called only from any class in the same package or subclass.
- *Default*. Method can be called only from the classes in the same package. There's no keyword for this one, you just don't put any access modifier.

Optional Specifiers.

Unlike access modifiers, you can have multiple optional specifiers in the same method declaration (not all combinations are legal). You can have them in any order, in case of using more than one. Here's the list:

- *Static*. Covered in Chapter 5. Used for class method.
- *Abstract*. Covered in Chapter 5. Used when not providing method body.
- *Final*. Covered in Chapter 5. Used when a method is not allowed to be overridden by a subclass.
- *Synchronized*. Covered in OCP scope.
- *Native*. Covered in OCP scope.
- *Sctrictfp*. Covered in OCP scope.

Return Type.

The return type might be an actual Java type such as String or int. It can be a custom java object created by the programmer. If there is no return type, the *void* keywork is used. The methods that possess a return type, must have a return statement inside of the body. Only void can omit this return statement.

```
public String walk() { return "”; }
```

Method Name.

It follows the same rules previously stated in Chapter 1: an identifier may only contains letters, numbers, $ or _. First characters cannot be a number and reserved words are not allowed. By conventions, methods begin with a lowercase letter.

Parameter List.

Although the parameter list is required, it doesn't have to contain any parameters. If you do have multiple, they are separated by comma. Some rules will be shared shortly when talking about varagrs.

Optional Exception List.
Code can indicate something went wrong by throwing an exception (will be covered in Chapter 6). It is optional to include as many exceptions needed in the method signature, if needed and it goes right after the parameter list, followed by the keyword "throws".

Method body.
It's simply a block of executable code, enclosed by two curly braces (there are exceptions with abstract classes and interfaces, covered in Chapter 5).

Working with Varargs.
A method can use a vararg parameter as if it is an array. It's different than an array. It needs to be strictly the last element in the parameter list, for which it can be used only once.
When calling a method with vargars, you can pass an array, or a list of elements of the array and let Java crate it for you. Also, you can omit it by passing zero parameters.

Applying Access Modifiers.
Private Access.
Only code in the same class can call private method or access private fields. See FatherDuck.java and BadDuckling.java classes in package pond.duck for examples.

Default (Package Private) Access.
Allows classes in the same package to access the members of the class. No access modifier / keyword is used. See MotherDuck.java and GoodDuckling.java classes in package pond.duck for examples.

Protected Access.
Allows everything that default access allows and more. Adds the ability to access members of a parent class (this means by a subclass, using the 'extends' keyword) and by other classes as long as they are under the same package, by creating an instance ('new' keyword). See all classes under "pond" folder as examples.

However, there are some other scenarios, using an instance ('new' keyword): in class under a different package, where inheritance is present ('extends'), using an instance that refers to itself it's valid, but using an instance that refers to the parent class with the protected members, the code won't compile. See pond.swan.Swan.java example. To paraphrase, these are the two scenarios for protected access:

1) A member is used without using a variable. This is done through inheritance, extending the parent class with protected members.

2) A member is used through a variable. This is done by creating an instance, with the 'new' keyword. Only and exclusively works when under the same package as the class with protected members. Works in the subclass when referring to itself.

Public Access.
Anyone can access the members from anywhere.

Designing Static Methods and Fields.
Static methods don't require an instance of the class. You can thing of it as a member of the single class object that exists independently of any instances of that class.

Note: instance and static methods has only one copy of the code. Parameters and local variables (as the reference variable) gets space in the stack. Only data gets its "own copy". No need to duplicate copies of the code itself.

Calling a static variable or method.
You just put the classname before the method or variable and you are done. Example:

```
System.out.println(Koala.count);
Koala.main(new String[0]);
```

Instance variables can be used to access static methods and variables. Be advise that it doesn't matter how many reference variables point to the same object, a static method or variable is not duplicated, only one exists in memory. E.g.:

```
Koala.count = 4;
Koala k1 = new Koala();
Koala k2= new Koala();
k1 = null;
System.out.println(k1); // 4
k2 = 6;
k1 = 5;
System.out.println(Koala.count); // 5
```

Static vs. Instance.
There are static and instance members of a class (methods and fields); for which it's not allowed to try to call an instance member without creating an instance first, from a static method. However, only an instance method can call another instance member on the same class without using a reference variable, because instance methods do require an object. See Static and Gorilla examples.

Type	Calling	Legal?	How?
Static Method	Another static method or variable.	Yes	Using classname.
Static Method	An instance method or variable.	No	Not without instantiating the object.

Instance Method	A static method or variable.	Yes	Using the classname or a reference variable.
Instance Method	Another instance method or variable.	Yes	Using a reference variable.

Static Variables.
Some static variables are meant to change as the program runs. Counters are common examples of this. A static variable can be initialized in the same line it's declared: `private static int counter = 0;`

Other static variables are meant to never change, to which we refer as a constant using the final keyword. A different naming convention is used: all uppercase letters. E.g: `private static final int NUM_BUCKETS = 45;`

Static Initialization.
Similar to the instance initializers from Chapter 1, for static initializers, we add the static word before the curly braces. The statement can hold and assign as many variables as needed.
`static { // body }`

Static Imports.
Static imposts are for importing static members of classes, as regular imports are used for the class. You can use wildcard or import a specific member. The idea is that you shouldn't have to specify where each static method or variable comes from each time you use it. See StaticInJava.java for examples.

By importing the following member: `import static java.util.Arrays.asList;`
Java will allow to replace this: `List<String> list = Arrays.asList("one", "two");`
With this instead: `List<String> list = asList("one", "two");`

Passing Data Among Methods.
Java is a "pass-by-value" language. This means that a copy of the variable is made, and the method receives that copy. Assignments made in the method do not affect the caller.

Overloading Methods.
It occurs when there are different method signatures with the same name but different types of parameters. Also allows different numbers of parameters. Everything other than the method name can vary for overloading methods (access modifiers, specifiers, return types and exception lists).

Overloading and Varargs.
Java treats varargs as if they were an array, but we need to keep in mind that they don't compile exactly the same. For which, the following case is not valid for overloading:

```
public void fly2(int[] lengths) {}
public void fly2(int... lengths) {}       // Does not compile
```

Then, in order to call a method, these are the ways we can pass parameters:

```
fly2(new int[] {1, 2, 3});          // Valid for array and varargs
fly2(1, 2, 3);                      // Valid only for varargs
```

Autoboxing.

```
Public void fly(Integer numMiles) {}
```

When parameters are wrapper classes, Java will do the autoboxing to transform the value to the primitive data type. However, java can recognize that if there's a method overloading in which there is a primitive type available (for this example, int), then it won't do extra work and use the method that receives the primitive type.

When it comes to Reference and Primitives types overloaded methods, java will always try to find the most specific version that it can, when you call the method and pass the data, by evaluating the data type of the parameter passed.

Creating Constructors.

It's a special method that matches the name of the class and has no return type. Constructors are used when creating a new object. This process is called instantiation because it creates a new instance of the class. The constructor gets called when we write 'new' keyword followed by the name of the class. See Bunny.java for examples.

Whenever you don't create a constructor in a class, java provides it automatically. This is called the *default constructor*. Java will provide it, empty, when the code gets compiled. Java won't create it if a constructor is found, no matter if it's empty or if it has parameters, or the access modifier. This code snippet calls a default constructor:

```
public class Rabbit {
     public static void main(String[] args) {
          Rabbit rabbit = new Rabbit();
} }
```

Interesting note: if you code a constructor to be private, the class cannot be instantiated. It's a good practice if all methods within that class are static or the class wants to control all calls to create new instances of itself.

Overloading Constructors.

You can have multiple constructors in the same class as long as they have different method signatures. See Hamster.java class for examples and see BetterHamster.java for examples on how to avoid code duplication.

The *this()* keyword when used in a constructor, it needs to be the first uncommented line of code that is called, otherwise it won't compile.

Constructor chaining: it's when constructors call each other until getting to the constructor that does all the work. For this sample, see Mouse.java.

Final fields. We can also initialize final variables in the constructor, as long as they haven't been initialized anywhere else. See MouseHouse.java for examples.

Order of Initialization.

1. If there's a superclass, initialize it first (this rule will be covered in Chapter 5).
2. Static variable declarations and static initializers in the order the appear in the file.
3. Instance variable declarations and instance initializers in the order they appear in the file.
4. The constructor.

Rule 1 and 2 apply if the class is referred without a 'new' call / instance. The four rules apply when a class instance is created. For examples see: InitializationOrderSimple.java, CallInitializationOrderSimple.java, InitializationOrder.java and YetMoreInitializationOrder.java classes.

Encapsulating Data.

Encapsulation means we set up the class so only methods in the class with the variables can refer t the instance variables. Callers are required to use these methods.
For each private member field, we set up an accessor method or getter and a mutator method or setter. Only these members have access to private fields. The setters and getters are set to public access modifier.

JavaBeans rules for naming conventions.

Rule	Example
Properties are private.	`private int numEggs;`
Getter methods begin with *is* or *get* if the property is a boolean.	`public boolean isHappy() { return happy; }`
Getter methods begin with *get* if the property is not a boolean.	`public int getNumEggs() { return numEggs; }`
Setter methods begin with *set*.	`public void setHappy(boolean happy) { this.happy = happy; }`
The method name must have a prefix of set/get/is, followed by the first letter of the property in uppercase, followed by the rest of the property name.	`public void setNumEggs(int num) { numEggs = num; }`

OCA-Chapter 4 Notes

Creating Immutable Classes.
By making a class immutable, it cannot be changed at all. This makes a class easier to maintain and helps with performance by limiting the number of copies made in memory. One step to make the class immutable is to omit the setters and allow to pass initial values through a constructor. See ImmutableSwan.java class as example. Also see NotImmutable.java class for examples of how a class can still be mutable by the fact is handling an datatype that allows changes (StringBuilder).

Writing Lambdas.
In Java 8, the language added the ability to write code using another style: functional programming. You specify what you want to do rather than dealing with the state of the object. You focus more in expressions than loops. It uses lambda expressions to write code. A lambda expression is a block of code that gets passed around.

It has parameters and a body like a full-fledge method do, but it doesn't have a name. For some traditional code and basic lambda samples to see the difference, see Animal classes in Lambda folder of this repo. You will see this code below in TraditionalSearch.java:
```
print(animals, a -> a.canHop());
```

Java relies on context to figure our what lambda expressions mean. In the TraditionalSearch sample, we are passing a lambda to the "`print`" method. The lambda being passed is: `a -> a.canHop();`

The `print` method expect a CheckTrait as second parameter, since it's a lambda, Java tries to map our lambda to that interface: `boolean test(Animal a)`
Since the interface's method takes an Animal, that mean the lambda parameter has to be an Animal. And since that interface returns boolean, we know the lambda returns a boolean.

Now let's talk about syntax.
```
a -> a.canHop()
```
Where it has 3 parts. The first specify a single parameter with the name **a**. The second is an operator to separate the parameter and the body. The last one it's the body, calls a single method and returns the result (boolean).

```
(Animal a) -> { return a.canHop(); }
```
This one does the same the previous one but it's more verbose. The first part specifies a single parameter with he name **a** and states the type is **Animal**. The second is the operator that separates the parameter and the body. The last one is the body, that includes curly braces, return type and finish the statement with a semicolon.

Parenthesis is needed in the parameter, when it's more than one or the datatype is stated. Curly braces in the body is used for more than one statement, or we want to use the **return** keyword; also keep in mind the semicolons are needed to end each statement.

Predicates.
Lambdas work with interfaces that have only one method, like CheckTrait.java. These are called functional interfaces. But as the code expand, we would need more of these interfaces for each reference datatype we handle in our code. Therefore, Java has solved that problem by providing a package java.util.funtion that allow us using Predicate / Generics:

```
public interface Predicate<T> { boolean test(T t); }
```

Where T is the datatype (and it can be ANY datatype) that is being passed to the interface. Java 8 integrated the Predicate interface into some existing classes. ArrayList declares a removeIf() method that takes a Predicate. Please see PredicateSearch.java for examples.