Java: Methods

Quick summary

- Methods are actions that an object can do.
- Methods accept arguments of specific data types.
- You must specify whether the method returns something.
- You declare methods inside of classes.
- Methods always have a () after them.
- If the method accepts arguments, put them inside the ().
- Even the arguments require you to specify what data types they are – for example, (int x, int y) – include a return type before the method.
- Methods have something called a "method signature." This is simply the declaration part of a method – for example: int childAge(int years);
- Suppose your method returns a string. If so, add return x at the end of the method.
- Methods are written in camel case.

Eclipse

See the getter_and_setter2 example.

Keyboard.java:

```
package getter_and_setter;


public class Keyboard {

	private String functionKeys;


	public String getFunctionKeys() {

		return functionKeys;

	}


	public void setFunctionKeys(String functionKeys) {

		this.functionKeys = functionKeys;

	}
```

```
}
```

App.java:

```
package getter_and_setter;


public class App {


	public static void main(String[] args) {
		Keyboard myKeyboard = new Keyboard();
		myKeyboard.setFunctionKeys("f11, f10, f9");
		myKeyboard.getFunctionKeys();
	}


}
```

Return types

RETURN TYPES	DESCRIPTION
void	returns nothing
String	returns a string

For a description of the access specifiers (e.g., public, private, protected), see Access Specifiers .

Getter and setter methods

Some methods are referred to as “getter” and “setter” methods. With a getter method, you retrieve a specific value. For example:

```
int childAge () {
age = 4;
return 4;
}
```

In your main method, you just call the method to get the value:

```
childAge();
```

Setter methods are a bit different. With setter methods, you set the value using the method. Here’s an example:

void int setChildAge (int years**) {**

this.years **=** years**;**

}

To set the age, you pass arguments to the method:

childAge**(**6**);**

The childAge.age will then be equal to 6.

The this keyword is important. Here's what's happening. When you use this, it refers to the object itself, so you're setting the instance variable. If you don't use this, then you set the class variable value. The this is a really common pattern.

Main method

In one of your Java application's classes, a class must contain the main method as follows:

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

// code to run is here...

}

The JVM will look for this main method and run whatever logic is included here. You normally include the main method just once in your application. Your main method can refer to other classes and methods in the same package. If the classes or methods are in another package, you must import the package into this class file so the main method can access them.

You can actually have more than one main method, but then you'll need to define which main method you want to run in your app. (not sure how you do that…)

Method chaining

As long as an object returns an object, you can chain the methods like this:

.getCat**().**getDog**().**getLizard**()**…**...**

A method that returns void cannot be chained.

Static methods

Static methods are methods that can be run based on the class alone, without the need to instantiate the method in an object.

Static variables are also variables that you can access using the class alone, without the need to instantiate the variables in an object.

Passing parameters to methods

Note the terminology: "passing parameters to the method".

You usually want to pass values into methods to change what it returns, or to customize the data that the method runs.

instead of arguments, the term seems to be “parameters”. you change a parameter to affect how a method runs.

If a method takes a parameter, when you run the method, you have to be sure to pass in a parameter. Otherwise you’ll get an error.

The order of the parameters you pass is important.

You can also create a variable, set it equal to something, and then pass in the variable to the method. You don’t always have to include the parameters in their raw form when you pass parameters to the method.

Void

If you don’t have void set on your method, then you must include a return value at the end of the method. Otherwise, Eclipse flags it as an error. Keep in mind that a lot of methods return information – that’s the point of using the method. So they most likely will return a value.

Instance variable

When a variable is part of the instance of the object, you call it an “instance variable.” When the variable is part of the class, it’s referred to as a … not sure. Maybe “local variable”?

Static methods

Here’s what it means for a method to be static:

Methods can also be static. On a method, the static keyword means that only one such method exists in the system, no matter how many objects of that class might exist. For that reason, a static method is also known as a class method. Also, the method cannot use any of the class’s fields other than fields that have themselves been declared static (usually constants). Static methods can be a handy way to provide utility methods for use throughout your code. For example, a method that takes a date and returns a formatted string might be static, provided it doesn’t need to use any of its containing class’s non-static methods. Also, static methods can be slightly more efficient, because they don’t access anything outside the method body.

Static Members Classes can have static members, including fields, methods, and other classes (a class within a class is called an inner class). A static member of a class is often called a class member. The important thing to know about class members is that only one instance of that member ever exists. If our Cat class has a static method, we might have ten different instances of the Cat class, but there would only ever be one of that static method. That becomes an issue when each of the members of the Cat class want to use that method. The instances end up waiting for each other.

Static members have their uses, though. When you want to be certain that only one of something exists for all the objects that instantiate a particular class, the static keyword is how you do it. One obvious use of the static keyword is on the main method. Imagine if every instance of a program class could start a new program. We would quickly swamp the operating system with programs. A more common and useful use of static members is to implement counters. Suppose we want to keep track of how many Cat objects we create. The Cat class could then include a static field called numberOfCats, and the constructors for the class would increment that field every time we create a Cat object. That code would look something like the Cat class in Listing 6-4.

Listing 6-4. Counting Cat objects

```
class Cat extends Mammal{


  static int numberOfCats;


  Cat() {
    numberOfCats++;
  }
}
```

To get the number of Cat objects, we can either implement a method to return the value or we can reference the field and get its value. To reference a static member, we use the name of the class separated from the static member's name by a period, as shown in Listing 6-5.

```
System.out.println(Cat.numberOfCats);
```

However, we should generally prefer the idiom of making the static field private and creating a get method to return that value.

```
class Cat extends Mammal{


   private static int numberOfCats;


   Cat() {
     numberOfCats++;
   }


   public static final int getNumberOfCats() {
     return numberOfCats;
   }
 }
```

Do you notice that the method is also static? We need only one such method, regardless of how many Cat objects we create, so it makes sense for getNumberOfCats to be static. We also make it final, because there's no reason for child classes to implement their own getNumberOfCats methods. Thus, if we have Tiger and Lion classes extending the Cat class, they could not have getNumberOfCats methods unless those methods have different arguments.