Hyperion Development

Task: Object Oriented Programming

www.hyperiondev.com

Introduction

Welcome to the Object-Oriented Programming Task!

So far we've discussed a fair amount of essential coding concepts and constructs (like data structures loops and methods): the nuts and bolts of programming. In this section, we momentarily take a step back to study the higher level issue of object-oriented programming. This paradigm has entrenched itself into software engineering as the most used paradigm of system design. In this lesson we will find out why, and also learn the fundamentals of object-oriented programming (OOP) so we can start to build our own object-oriented applications.

You don't have to take our courses alone! This course has been designed to be taken with an online mentor that marks your submitted code by hand and supports you to achieve your career goals on a daily basis.

To access this mentor support simply navigate to www.hyperiondev.com/support.

What is Object-Oriented Programming?

Object-oriented programming (OOP) has become the dominant programming methodology for building new systems. C++, an object-oriented version of C, was the first OO language to be used widely in the programming community. This adoption inspired most contemporary programming languages like Java, C#, and Objective-C to be designed as object-oriented languages. Even languages which were not primarily designed for object-oriented programming, like PHP and Python, have evolved into supporting fully fledged object-oriented programming.

So why is OOP so popular? Firstly, we as humans think in terms of objects and designers of modern technologies have used this knowledge to their advantage. For instance, we refer to the window that appears when we power up our computer as a "desktop". We keep shortcuts and files we use frequently on this "desktop." But we know this is no real desk! We refer to the folder that our files get transferred to when we delete them as a "recycle bin." But is it really a bin? We use these terms in a metaphorical sense because our minds are already familiar with these objects. Software designers leverage this familiarity to avoid unnecessary complexity by playing to the cognitive biases of our minds.

Object-oriented programming was developed for this specific reason. You know from your experience as a programmer, and from the foregoing tasks in this course, that solving computation problems can be complex. It is not just the process of developing an algorithm to solve a problem that is challenging, but the problem domain is often loaded with delicate intricacies that require expert knowledge to understand. These two problems contribute to the overall complexity of designing and implementing software that solves business problems. OOP embraces our mind's natural affinity to think in terms of objects by designing a solution in terms of objects.

This is a paradigm shift away from the procedural paradigm (described in the next subheading) we're accustomed to.

Procedural Programming vs Object Oriented Programming

Procedural code executes in a waterfall fashion. We proceed from one statement to another in sequence. The data are separated from the code uses them. Functions are defined as standalone modules but, they have access to variables we've declared outside the function. These two properties: sequential execution of code and the separation of data and the code that manipulates the data are what distinguish procedural code.

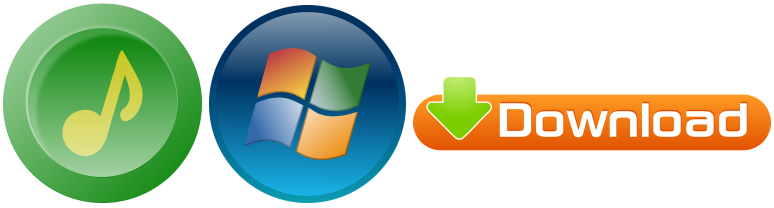
In contrast, with object-oriented programming a system is designed in terms of objects which communicate with each other to accomplish a given task. Instead of separating data and code that manipulate the data, these two are **encapsulated** into a single module. Data is passed from one module to the next using **methods.**

How Do We Design OO Systems

We've seen the difference between the way we usually write code and how it's done in an object-oriented way. But what is an object, and where does it come from? An object is created using a **class**. A class is the blueprint from which objects are made. It consists of both data and the code that manipulates the data.

To illustrate let's consider an object you encounter every time you use software, a button. As shown in the images below, there are many different kinds of buttons you might interact with. All these objects have certain things in common.

1. They are described by certain **attributes.** E.g. every button has a *size*, a *background colour*, a *shape*. It could have a specific *image* on it or it could contain *text*.
2. You expect all button to have **methods** that do something. E.g. when you click on a button you want something to happen - you may want a file to download or an app to launch. The code that tells your computer what to do when you click on it is written in a method.



Although every *object* is different, you can create a single *class* that describes all objects of that kind. The class defines what attributes and methods each object created using that class contain. Each object is called an instance of a class. Each of the buttons shown in the images above is an instance of the class button.

In other words, it encapsulates both code and data. Here is a class that represents a student:

```
public class Student {
	// Attributes
	String name;
	int age;
	int grade;
```

```
        char registrationClass;

        // Methods
        public String getName() {
              return name;
        }

        public char getRegistrationClass() {
              return registrationClass;
        }
}
```

This is the template from which individual objects will be instantiated from. Note that a class is made up of two sections - attributes and methods.

We call the data that a class stores ***attributes***. We identify attributes by thinking of features that an object can take on. We can expect a student in high school to have a *name*, a *grade* they are currently in, and an *age*. The class also contains ***methods***, which contain the code that operates on this data. Methods describe what an object can do.

To create a new object in our program, we need a special method called a **constructor**. This method is used to initialise the attributes to the values that we specify for each object. For example, lets create three objects with the following features:

Object One:

- Name = Sally
- Age = 15
- Grade = 8
- Registration Class = 'D'

Object Two:

- Name = Sipho
- Age = 17
- Grade 11
- Registration Class = 'A'

Object Three:

- Name = Rajesh
- Age = 19
- Grade = 12
- Registration Class = 'B'

Below is an example of a constructor. It contains the code that we will use to create these objects. It is part of the Student class and is listed under the methods section in our class:

```
public Student(String name, int age, int grade,
char registrationClass) {
      this.name = name;
      this.age = age;
      this.grade = grade;
      this.registrationClass = registrationClass;
}
```

The constructor contains a list of all the attributes in our class. These are used to pass values from a client object that will be used to **instantiate,** or create a new instance of, a new object. You can see how this is done inside the constructor. Each line assigns the

value passed to the constructor to the attribute of the class. The keyword this indicates that we are referring to the attribute within the class, and not the variable used to pass data into this class. Now that we've seen what the constructor does, let's create an the three objects above.

```
Student sally = new Student("Sally", 15, 8, 'D');
Student sipho = new Student("Sipho", 17 , 11, 'A');
Student rajesh = new Student("Rajesh", 19, 12, 'B');
```

The first thing to note is data type the left hand side. The student class is considered a legitimate data type! We refer to a class as a **user defined type**. This is in contrast to the **primitive data types** that come with the programming language (e.g string, integer etc.). These are created by the language designers and cannot be changed at all. This is another advantage of OOP, as is that it allows you to define your own data types and thereby gives you control over the kinds of data you want to introduce into your program.

Note that we give these objects meaningful variables names, instead of Object One, Object Two and Object Three as above. This helps with readability of the code. The naming of objects is completely at your own discretion, however, you must follow best practice at all times.

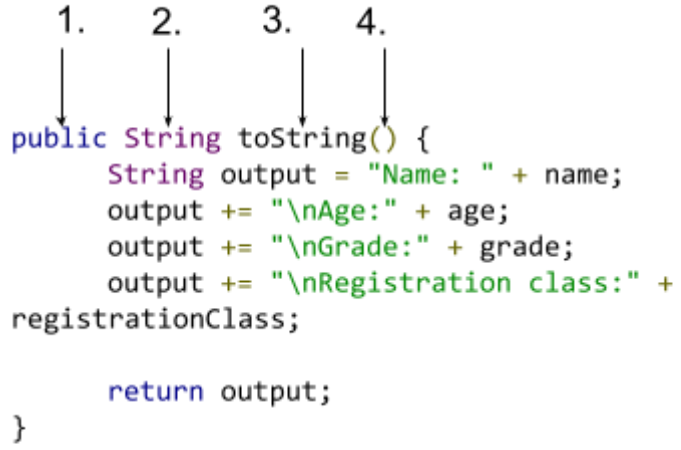
On the immediate right hand side of the assignment operator you will find the Java keyword **new**. This keyword is used to instantiate a new object and must be included immediately before a call to the constructor. Finally, after the **new** keyword we find a call to the **Student** class constructor we created above. This call contains the values we want to assign to the internal attributes for each object. Each value is assigned to its respective constructor variable respectively, therefore values must be listed in the same order in which they are arranged in the constructor. If you fail to do this you'll probably create a compilation or logical error in your code.

Note that we've created these objects in a separate code file named **School.java**. You will find the example code enclosed in this task file and can explore it further to gain a more practical sense of the mechanics of OOP implementation in Java.

Let's return to the **Student** class and define a method to display the contents of each variable. Four things are required to create a method:

1. The keyword public
2. The return type of the method
3. The name of the method
4. A parameter list that declares the values that must passed to this method

Our method will return a String object that we'll use outside to class to display the contents of each object. We will declare our method below the existing three methods:




```
public String toString() {
      String output = "Name: " + name;
      output += "\nAge:" + age;
      output += "\nGrade:" + grade;
      output += "\nRegistration class:" + registrationClass;

      return output;
}
```

This method satisfies our four conditions. We begin with the **public** keyword before declaring the return type for this method - String. The name, **toString**, then follows. You'll notice that we don't have a list of parameters for this method. This is fine because the fourth requirement is actually optional. If your method does not require external data, like our one here, then you don't need to provide a means for it to receive data.

The last statement in the method returns the String (variable named output) we've just constructed. This is done through the **return** keyword. A return statement follows this precise pattern - the keyword followed by a variable or value. The type of the variable or value returned must correspond to the return type declared for this method, as per requirement 2 above.

In our **School.java** file, which we are using as a client to test this code, we're going to call this **toString** method for each file and see that happens. Here is the code to test our method:

```
System.out.println(sally.toString() + "\n");
System.out.println(sipho.toString() + "\n");
System.out.println(rajesh.toString());
```

We call the **toString** method on each object we created. Note the **dot notation** we use to call methods. The variable name is written, then immediately after the name goes a dot, followed by the name of the method and any parameters enclosed by round brackets. Since we didn't declare any parameters for our **toString** method, we don't put anything within the round brackets.

Below is the output from the code above:

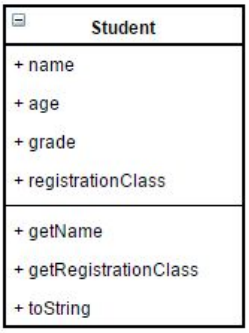
```
Name: Sally
Age: 15
Grade: 8
Registration class: D

Name: Sipho
Age: 17
Grade: 11
Registration class: A

Name: Rajesh
Age: 19
Grade: 12
Registration class: B
```

Note that each object displays the specific values that we assigned to it on instantiation.

That concludes our introduction to object-oriented programming. Ensure that you study the code included in the example project within this task's folder. You will use this project to complete the compulsory task below.

Class Diagrams

Class diagrams are used to illustrate a class graphically. They are part of a modelling language called Unified Modelling Language (UML). This is the most widely used tool for modelling object-oriented systems. Below is a class diagram of the Student class we created above:



*Class diagram of **Student** class*

This diagram has three sections. The first section contains the name of the class. The second section contains a listing of each attribute in the class. And finally, the last section lists the methods in the class. These diagrams communicate much more information than we've included here. As we delve into the nuts and bolts of OOP, we will add more details to in this diagram to communicate them visually.

The Hyperion Team

Compulsory Task

1. Create two classes named Teacher and Secretary. Each class should contain:
    a. At least five attributes
    b. A constructor
    c. A **`toString`** method to display values of all the attributes of each class

2. Create a class diagram for the following classes:
    a. Student
    b. Teacher
    c. Secretary

3. Edit School.java to test the **`toString`** methods you've created for each of the classes above. To test the **`toString`** methods, create three instances of each class (each with different attribute values) and display them as was done for the **Student** class.

SHARE YOUR THOUGHTS WITH US

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes. Think the content of this task, or this course as a whole, can be improved or think we've done a good job? Click here to share your thoughts anonymously.