Hyperiondev

TASK

Object-Oriented Programming

Visit our website

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 concept 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 learn the fundamentals of object-oriented programming (OOP). Then we can start to build object-oriented applications.

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 that 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. 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 frequently use on this "desktop." However, 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 the preceding 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 to which we're now accustomed.

PROCEDURAL PROGRAMMING RECAP

As you know, procedural code executes in a waterfall fashion. We proceed from one statement to another in sequence. The data is separated from the code that uses it. Methods are defined as standalone modules. However, they have access to variables we've declared outside the methods. 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, an object-oriented programming system is designed in terms of objects that communicate with each other to accomplish a given task. Instead of separating data and the code that manipulates it, these two are **encapsulated** into a single module. Data is passed from one module to the next using **methods**.

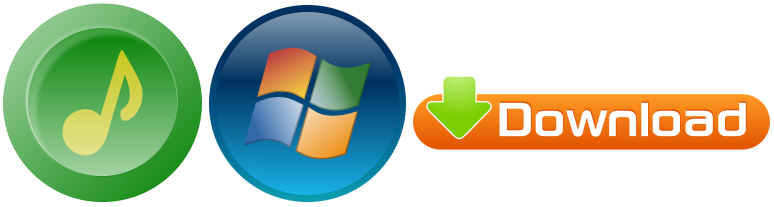
HOW WE DESIGN OO SYSTEMS

We've seen the difference between the way we usually write code and how it's done in an object-oriented way. In Java, as in other languages, an object is created using a **class**. A class is a 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 with which you might interact. All these objects have certain things in common.

1. Certain **attributes** describe them. 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 buttons 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, like a *Button* class. The class defines what attributes and methods each object created using that class contains, like all buttons must have a colour, a size and an action. 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 (e.g. musicButton, windowsButton and downloadButton). 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. 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, let's 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 used 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;
}
```

When we create, or instantiate, a new object of the class, its new attributes are given values thanks to the constructor. You can see how this is done above: each line assigns the value passed into the constructor to the attribute of the new object. The keyword **`this`** indicates that we are referring to the attributes within the new object. Now that we've seen what the constructor does, let's create 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 the data type on the left-hand side of the statement. The `Student` class is 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. `int`, `double`, etc.). These are created by language designers and cannot be changed. An advantage of OOP 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.

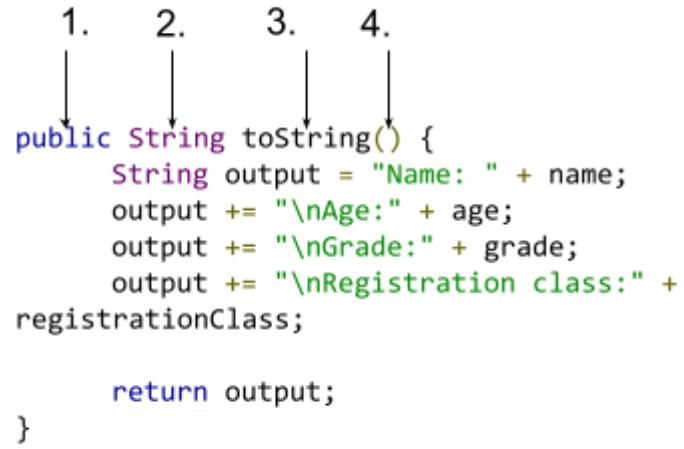
Note that we give these objects meaningful variable names instead of generic names like ObjectOne, ObjectTwo and ObjectThree. This helps with the readability of the code. You can name objects whatever you like; however, it is best to follow best practice at all times.

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. 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 be passed to this method



Our method will return a **`String`** object that we'll use outside the 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 parameter list is 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, another file we are using to test this code, we're going to call this **`toString`** method for each file and see what 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:

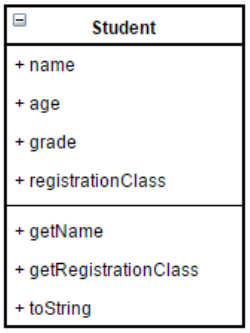
```
Name: Sally
Age: 15
Grade: 8
Registration class: D

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.

Take note:

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. 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 this diagram to communicate them visually.

Compulsory Task

- Create two classes named Dog and Cat. Each class should contain:
  - At least five attributes
  - A constructor
  - A **toString** method to display the values of all the attributes of each class

- Create a class diagram for the following classes:
  - Dog
  - Cat

- Edit **School.java** above to check 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.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that 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.