Download

Day 11: OOP 2

Download

Agenda

- OOP Quick Review
- Getters and Setters
- Static Methods in Instance Methods
- toString
- Array of Objects
- null
- Project 2
- Lab 6

Download

OOP Quick Review

Download

Class Structure

Download

A class follows the structure to the right.

Remember, the name of the file must match the name of the class.

`ClassName.java`

```
public class ClassName {
  // field variables/members
  public ClassName(params) {
    // Constructor Code
  }
  // Methods
}
```

```
public class Stationary {
  private String name;
  private String description;
  public Stationary(String name, String description) {
    this.name = name;
    this.description = description;
  }
  public String getName() {
    return this.name;
  }
}
```

Class Name

Field Variables

Constructor

Methods

Download

Getters and Setters

Download

In order to access and modify private variables in a class, we need to have public methods to get and set values. These are called getters and setters.

Download

Getters

This will get the value and return it. Nothing else needs to happen and it simply returns the value outward so other scopes can see the hidden, encapsulated, values.

```
private String name;

public String getName() {

  return this.name;

}
```

Download

Setters

This will set and replace the value in a field variable with a given new value. This method is often a void method as it doesn't return anything and simply updates a value.

```
public void setName(String name) {
  this.name = name;
}
```

Download

Accessing static methods in instance methods

Download

Static functions can be used to write more generic code or to write utility functions that can have specialized functionality within the class itself.

```
public static String getInfo(String f, String l) {
  StringBuilder s = new StringBuilder();
  s.append(f).append(" ").append(l);
  return s.toString();
}

public String getInfo() {
  StringBuilder s = new StringBuilder();
  s.append(getInfo(this.firstName, this.lastName));
  s.append(" ").append(this.age);
  return s.toString();
}
```

Download

We can call static functions in instance methods.

```
public static String getInfo(String f, String l) {
  StringBuilder s = new StringBuilder();
  s.append(f).append(" ").append(l);
  return s.toString();
}

public String getInfo() {
  StringBuilder s = new StringBuilder();
  s.append(getInfo(this.firstName, this.lastName));
  s.append(" ").append(this.age);
  return s.toString();
}
```

Download

toString

Download

When we print out an Object itself, we often get some information printed out to the console.

When we print out an object, we actually call a method called `toString()` behind the scenes.

```
Student s = new Student("Edward",
"Rees", "erees", "20250101", 25);

System.out.println(s);

// Could give us something like:
"Student@2a138b55"
```

Download

This `toString` default behavior returns the class name, an @ sign, and the hex string of what we call the hashcode of the object.

```
Student s = new Student("Edward",
"Rees", "erees", "20250101", 25);

System.out.println(s);

// Could give us something like:
"Student@2a138b55"
```

Download

This Hex String is a hexadecimal (base 16) version of a number.

A hashcode is an identifier for an object that allows for object comparison and equality. This implementation depends on the JVM, but often is the memory address for the object converted into an integer.

```
Student s = new Student("Edward",
"Rees", "erees", "20250101", 25);

System.out.println(s);

// Could give us something like:
"Student@2a138b55"
```

Download

While the default behavior is there, we can actually override the `toString` method, to provide more information about the object itself.

```
Student s = new Student("Edward",
"Rees", "erees", "20250101", 25);

System.out.println(s);

// After overriding toString, we
could get something like:

// "[erees/20250101] Edward Rees:
25"
```

Download

What does this overriding look like?

```
@Override
public String toString() {
  StringBuilder s = new StringBuilder();
  s.append("[");
  s.append(this.username);
  s.append("/");
  s.append(this.id);
  s.append("] ");
  s.append(this.firstName);
  s.append(" ");
  s.append(this.lastName);
  s.append(": ");
  s.append(this.age);
  return s.toString();
}
```

Download

When we add the **@Override** annotation above the method, the JVM sees that we are overriding the base implementation and uses our version instead.

```
@Override
public String toString() {
  StringBuilder s = new StringBuilder();
  s.append("[");
  s.append(this.username);
  s.append("/");
  s.append(this.id);
  s.append("] ");
  s.append(this.firstName);
  s.append(" ");
  s.append(this.lastName);
  s.append(": ");
  s.append(this.age);
  return s.toString();
}
```

This is important as we can then simply print out the object variable itself!

Since this method return a string, we not have a String representation of the object. Anytime we want to print out the object information, we can call this `toString()` method itself.

```
@Override
public String toString() {
  StringBuilder s = new StringBuilder();
  s.append("[");
  s.append(this.username);
  s.append("/");
  s.append(this.id);
  s.append("] ");
  s.append(this.firstName);
  s.append(" ");
  s.append(this.lastName);
  s.append(": ");
  s.append(this.age);
  return s.toString();
}
```

Download

With the `toString` method overridden, we can also do:

```
Student s = new Student("Edward",
"Rees", "erees", "20250101", 25);

String sInfo = s.toString();

System.out.println(sInfo);

// or even simpler:

System.out.println(s);
```

Download

Array of Objects

Download

Since objects are another data type, we can create an array of objects.

The logic is the same as an array of any other data type, but instead of primitive types, we have objects.This means we can use the object methods too.

```
Student[] students = new
Student[10];
```

Download

However, we must set values in the array to new instances of the data type.

```
Student[] students = new
Student[5];

students[0] = new Student( ... );

students[1] = new Student( ... );

...

for(Student s : students) {

  System.out.println(s);

}
```

Download

Alternatively, if we know the student objects already, we could also set them initially:

```
Student[] students = { new
Student(...), new Student(...), ...
new Student(...) }

for(Student s : students) {

  System.out.println(s);

}
```

Download

If we wanted to print out only the names of the students, we could also do this:

```
Student[] students = { new
Student(...), new Student(...), ...
new Student(...) }

for(Student s : students) {

  System.out.println(s.getName());

}
```

Download

null

Download

The default value for a primitive type is `0`, but for an object, it's `null`.

`null` is another primitive value, like how `None` is a value in Python.

`null` is used when an object doesn't exist or a value is not found.

```
if (obj == null) {
  return false;
}
return true;
```

Download

Project 2

Download

Reminder of Spec Grading

Download

Do the work for a C **first**, then implement the features for a B **next**, then work on the features for an A **last**. If you tackle the A features but your C and B features aren't working, you won't get an A.

Work incrementally, not tackling the A+ features when your base functionality doesn't work. Taking that approach will lead to less of the project functionally complete at the end, leading you towards getting a D instead. I do'nt want that when I know you're all capable of doing well.

Go slow and incrementally!

Download

Lab 6