Chapter 2: Classes in Java

2.1. Classes

Classes in Java are similar to those in Python: they consist of attributes and methods, both private and public. We can inherit from other classes, override methods, and define constructors.

While this section talks about classes at a higher level, the file Monster.java provides a larger example with in-line explanations which you may find helpful. You'll want to look at this file to get a better understanding of the syntax that we use.

2.2. Variables in classes

Unlike Python, we have to declare variables before using them in Java. There are two kinds of variables we can declare for classes:

1. **Instance variables**: These are like the attributes you're familiar with from Python. Every instance of the class will contain its own instance of each of these variables. They come into existence when the instance is constructed (using the `new` keyword).
2. **Class variables**: Also known as **static variables**, all instances of a class share a *single* instance of each class variable. Updating this variable in one instance of a class will reflect across every instance of the class. This is useful, for example, if we want the instances to accumulate a value together.

2.3. Visibility and Access Modifiers

Attributes and methods can be either **public** or **private**. In Python we used a single underscore (`_`) at the start of a variable name to denote whether an attribute was private or not. In Java we use the `public` or `private` keywords to make this distinction: private variables *cannot* be accessed from outside a class. We can also use the `protected` keyword, making the attribute or method accessible to the entire package and any subclasses of a class, but not to anything else. If no access modifier keyword is included, the variable is `package-protected` which is the same as `protected` but excludes subclass access from outside the package.

Additional information about access modifiers in Java can be found at https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html.

2.4. Constructors

Similar to the `__init__` method in Python, we have constructors in Java. These are methods with no return type (not even void), which get called whenever a new instance of a class is created.

In Java, we can define as many constructors as we want so long as the method signatures are different: either taking a different number of arguments, having different argument types, or a combination of these.

As an example, this would be one constructor:

```
public Something(String name, int size) {
    this.name = name;
    this.size = size;
}
```

In Python, we had the convention that all methods had `self` as their first parameter, including for the `__init__` for a class. When we called a method, `self` would be a reference to the object whose method we were calling. In Java, the `this` keyword can be used instead. Inside an instance method or constructor, `this` is a reference to the object whose method or constructor is being executed. Note that, unlike Python, we are not required to include `this` as a parameter in our method signatures!

We could create additional constructors if we wished. For example, if we wanted a constructor that only takes a name:

```
public Something(String name) {
    this.name = name;
    this.size = 1;
}
```

Even better: we could rewrite the above to call on the first constructor we wrote:

```
public Something(String name) {
    this(name, 1);
}
```

Note the syntax above, where `this(name, 1)` calls the `Something(String name, int size)` constructor defined previously.

2.4.1 Object Creation Order

When you create a new object in Java using the `new` keyword, several steps happen in a specific order:

1. Memory is allocated for the object.
2. Instance variables are initialized to their default values.
3. Direct initializations (e.g., `int x = 5;`) are executed in the order they appear in the class.
4. The constructor is called, which may further modify the instance variables.
5. If the class extends another class, the superclass constructor is called *first*; we'll talk more about this in the next chapter about relationships between classes.

This order ensures that all inherited and declared fields are properly initialized before your constructor logic runs.

> This is similar to Python, but differs in that steps 2 and 3 don't take place in Python — the call to `__init__`, which is equivalent to step 4, takes care of initializing all instance attributes.

2.5. Overloading methods

When we define multiple constructors, we are **overloading** the constructor. We can do this for any method, not just our constructors. In Python, we cannot overload methods, but we can provide default parameters. For instance:

```
def my_method(self, something: int, something_else: int = 1) -> int:
    return something + something_else
```

When calling `my_method`, we can optionally pass in a value for `something_else` or the default value of `1` will be used.

In Java, we get similar behaviour by overloading methods:

```
public int my_method(int something, int something_else){
    return something + something_else;
}

public int my_method(int something){
    return my_method(something, 1);
}
```

Note that we don't have to write `this.my_method(something, 1)` in the above example, since Java is able to determine which method we are calling. Including `this` is optional in this case. Previously, when we had the assignment `this.name = name` in our constructor, the `this` was necessary to distinguish between the instance variable `name` and the parameter `name`.

2.6. Overriding methods

In Python, we could re-define a method from a parent class in order to override it. We do a similar thing in Java, but we also include the `@Override` annotation. This lets Java (and any reader of our code) know that we're overriding an inherited method. The annotation will also enforce that we use the correct method signature for the overridden method — this way, we can be certain that we don't have any silly typos in our method or the incorrect argument types!

2.6.1. toString

One common method that we'll want to override is the `toString` method. This is the Java equivalent of the `__str__` special method in Python: a method that gives the string representation of our object.

This method takes no parameters and returns a `String`. For example:

```
@Override
public String toString(){
    return "My name is " + this.name;
}
```

2.6.2. equals

Another common method inherited from `Object` is the `equals` method: this is equivalent to Python's `__eq__` method. Note that this method is **NOT** called implicitly in Java: the `==` checks for identity equality (i.e. that the IDs of the objects are the same). You need to explicitly call the `equals` method (e.g. `object1.equals(object2)`) to check for equality.

The default behaviour of `equals` is to check for identity equality (like `==`), but often we'll want to override this behaviour.

> For primitives, since only the value is stored, two primitives are always checked for value equality.

The designer of a class gets to decide what has to be true in order for two instances to be considered "equals". This sounds trivial to implement, but there are a number of details to be handled. Any implementation of it must obey these properties:

1. **Symmetry**: For non-null references `a` and `b`, `a.equals(b)` if and only if `b.equals(a)`
2. **Reflexivity**: `a.equals(a)` must be true
3. **Transitivity**: If `a.equals(b)` and `b.equals(c)`, then `a.equals(c)`

The `equals` method will often look as follows:

```java
@Override
public boolean equals(Object obj) {
    if (this == obj){
        return true;
    }
    if (obj == null){
        return false;
    }
    if (this.getClass() != obj.getClass()){
        return false;
    }

    MyClass other = (MyClass) obj;  // Here we're casting the type of obj to our class
    // And then we'll want to compare the attributes of this to those of other
    // For example:
    if (this.name != other.name){
        return false;
    } else if (this.something != other.something){
        return false;
    }
    return true;
}
```

You can see another example of this in `Monster.java`. If you change the type of the parameter from `Object` to `Monster`, what happens? You should see some helpful hints from IntelliJ about what problem this will cause.

2.6.3. hashCode

Whenever we override the equals method, we will often want to override another inherited method called "hashCode". The hash code of an object is an integer value that obeys this property:

> If two objects are equal (according to the equals method), they have the same hash code.

If is not an if-and-only-if: it's fine for two objects with the same hash code *not* to be equal.

Why do objects have a hash code and why must it satisfy this property? Some important classes such as HashMap use the `hashCode` method of an object to decide where to store it. This allows them to retrieve the object later by just comparing hash codes and thus avoiding costly calls to the equals method. The `HashMap` class does all this using a data structure called a "hash table". Hash tables have remarkable properties and are thus very important in computer science. You can learn about them in more detail in a course like CSC263. Dictionaries in Python are one example of a class you may have used that relies on a hash map to provide efficient operations.

If we do not override the `hashCode` method in a given class, then the default `hashCode` will be the one from the `Object` class. As far as we're concerned, the hashCode generated by `Object` is random! We have no idea what the value is.

2.7. Class (static) methods

Just as we have class (static) variables, we can have class methods. Again, we use the `static` keyword to declare that a method is a class method. Since a class method is associated with the class and not the instance, we call it by prefixing it with the class name.

For example, suppose we have the following class method:

```
public static int population(){
    return MyClass.count;
}
```

To call it, we would call `MyClass.population()`. If we had an instance of MyClass called `m1`, we could also call `m1.population()`, but this seems a bit strange since the `population` method doesn't depend on `m1` itself.

Although an instance method can reference a class variable (or call a class method), the opposite is not true. A class method **cannot** access an instance variable or call an instance method directly.

So in our example above, the following would not compile:

```
public static int population(){
    return this.some_attribute;
}
```

There is no "this" when you are in a class method!

The only way for a class method to access an instance variable or call an instance method is if a reference to an instance of the class is passed to the method. Through that reference, the instance variables and instance methods of the object are accessible.

2.8 Keyword `final`

The `final` keyword in Java is used to restrict modification. It can be applied to variables, methods, and classes:

Final variables cannot be reassigned after initialization.

```
final int MAX_SIZE = 100;
```

For example, `MAX_SIZE` above cannot be changed. This is often used for constants, especially in combination with `static`.

Note that if a `final` variable refers to an object, the reference itself cannot be changed, but the object it points to **can still be mutated**.

```
final List<String> names = new ArrayList<>();
names.add("Alice"); // allowed — we're modifying the object
names = new ArrayList<>(); // not allowed — reassignment is forbidden
```

Final methods cannot be overridden by subclasses and final classes cannot be subclassed. Using `final` helps enforce immutability and design constraints, potentially making your code safer and easier to reason about.