A quick overview of OOP

Josh Felmeden

May 16, 2019

Contents

1 An object's Birth

An object in Java is probably the most crucial piece of information (because it's called **object** oriented programming duh). An object has behaviour, attributes, and identity. An object combines both *data* and *methods* (which manipulate the data) in a single unique entity.

A **class** is like the blueprint of an object. Classes act like a module or unit of description. Here is an example class:

```java
class Robot {
    String name;
    int numLegs;
    float powerLevel;

    void talk(String phrase) {
        if (powerLevel >= 1.0f) {
            System.out.println(name + " says " + phrase);
            powerLevel -= 1.0f;
        } else {
            System.out.println(name +  " is too weak to talk.");
        }
    }

    void charge(float amount) {
        System.out.println(name + " charges.");
        powerLevel += amount;
    }
}
```

A class is different to an object. Creating an object is called *instantiating* an object.

2 Java basics

Remember, a class is an immutable blueprint for an object. It has both methods and fields. An object is a stateful and unique instantiation of a class. Remember, a class is an immutable blueprint for an object. It has both methods and fields. An object is a stateful and unique instantiation of a class.

Some people don't like Java because they say that 'nouns' (objects) shouldn't dictate program structure. It potentially overemphasises data over algorithms. Additionally, classes and their relations put limitations and rigidity on reusability and modularity. It's often difficult to reform the code for parallelisation and so on

People also say that Java is stupidly verbose (wordy) and this doesn't need to be the case.

Java is kind of like C, except that lots of things are automated like:

- Exception handling
- Garbage collection

- Default values

It also doesn't use pointers which is really nice.

2.1 Static elements

Static methods can access static data and can change the value of it. Static methods are unable to use non-static data members or call non-static methods directly. Static code blocks can be used to initialise the static data members, and more importantly, they are executed before the main method at the time of class loading.

3 An object's life

Attributes capture what objects can be. Each object has its own copy of attributes, and they can be a few things:

- Plain data types (such as bool, char, int, ...)
- References to other objects
- ...

Methods, on the other hand, capture what objects actually do. Methods take parameters and return values. They are able to be *overloaded* (same name, different parameters).

3.1 References

When we write something like:

```
Adder adder = new Adder();
```

Here, 'adder' is NOT an object. It's just a reference to an object. In fact, we can create many more references to the same object. We're not allowed to create a reference to nothing.

3.2 the Heap

The heap is memory that's set aside for dynamic allocation. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap, so you can allocate a block at any time and free it at any time. This means that it's a lot more complicated to keep track of which parts of the heap are allowed to be freed or allocated at any given time. We look at one of the methods in computer architecture actually.

3.3 Reference equality

The '==' operator checks for equality just as you might expect. Unfortunately, it refers to the reference equality, not the equality of object attributes. If we want to check whether two different objects have the same value, we need to use the `equals()` method.

3.4 This keyword

The 'this' keyword provides a reference to the current object whose method is being executed. It can also be used inside a method or constructor, but not in a static element. Within a constructor, you can also use this in order to refer to another constructor method of the current object under construction.

3.5 Object scope

The cope of argument variables and local variables are the same as in C. However, objects do not just vanish when the creation reference goes out of scope. As long as there's a single reference somewhere in the program, the object is kept on the heap.

4 Inheritance

Sub-classes and polymorphism are parts of OOP that are used quite a lot. In order to fight code duplication with inheritance, we can define a parent class that defines attributes and methods that are common to the subclasses you plan to create. After this, we just extend the parent class with subclasses that add and/or override the parent class's characteristics.

An **abstract class** is one that prevents us from making instances of a class that we apply the abstract keyword to. They are often ones that are conceptual and don't need instances (such as a generic shape). The benefits of inheriting means that we don't have to repeat ourselves. The code can also be maintained way easier, and it also facilitates **polymorphism** (which is a reference that can be made to any object of a sub-class of the references class).

4.1 Dispatch

When an overridden method is called via a reference, the actual method to execute is selected based on the *type* of the object referenced, rather than the reference type. This is known as dynamic method dispatch. Since this dispatch decision cannot be made at compile time, dynamic dispatch refers to the choice of code execution (so the method call) resolved at runtime.

Dispatch is based on two or more types:

- **Single dispatch**, which could be something like `mammal.makeNoise();`. This resolves mammal to its underlying type and calls its make function rather than the mammals.
- **Double dispatch**, which looks like `(mammal1 & mammal2).makeNoise();`. You might do this for collision detection in games (for example).
- **Multiple dispatch** which we'll look at now

4.1.1 Multiple Dispatch

Big switch In the case of rock paper scissors, we can just write out each possible outcome in a big switch statement. This can be good because it means that the code is relatively easy to step through. Unfortunately, it makes adding or removing types a big job that won't be verified by a compiler. We also need a type field for all types and we have to keep track of all the types. Switch statements also tend to grow and show up as code replication throughout the project.

Function table