Object Oriented Programming - Day 1

- What do you remember learning before?
- What do you remember about objects and classes?

Java is an object-oriented language. Everything we do involves objects and classes. Important topic!

Words ¬ please ask meaning! Stop me when things don’t make sense anymore. Earlier than you think. If you have trouble following, and it doesn’t improve within 1 minute, say something!

Most difficult thing for me to do!

Module: encapsulation

Private properties

We’ve seen how to define a class before. You may recognize the following class from the section about objects:

```
class Purchase {
    String product;
    int quantity;
    double price;

    Purchase(String initialProduct, int initialQuantity, double initialPrice)
{
        this.product = initialProduct;
        this.quantity = initialQuantity;
        this.price = initialPrice;
    }

    double getSubTotal() {
        return this.quantity * this.price;
    }

    void increaseQuantity() {
        this.quantity = this.quantity + 1;
    }
}
```

In one of the exercises, we added some checks to the constructor to ensure that both the `quantity` and the `price` are always positive. A `Purchase` wouldn’t make sense otherwise. Someone might even walk away from the supermarket with a basket full or groceries without paying a cent.

```
// Construct an invalid purchase, with a negative quantity:
Purchase purchase = new Purchase("Invalid purchase", -10, 1.00);
```

```
// We will RECEIVE money (EUR 10.00) by buying this!
```

```
System.out.println("Pay " + purchase.getSubTotal());
```

Checks like that would look like somewhat like the following:

```
class Purchase {
    String product;
    int quantity;
    double price;

    Purchase(String initialProduct, int initialQuantity, double initialPrice)
{

        if (initialQuantity <= 0) {
            throw new IllegalArgumentException("Quantity must be positive!");
        }

        if (initialPrice <= 0) {
            throw new IllegalArgumentException("Price must be positive!");
        }

        this.product = initialProduct;
        this.quantity = initialQuantity;
        this.price = initialPrice;
    }

    // Like before...
}
```

Whenever `initialQuantity` or `initialPrice` is zero or less, we *throw an exception*. Throwing an exception prevents an object being constructed:

```
jshell> Purchase purchase = new Purchase("Invalid purchase", -10, 1.00);
|  Exception java.lang.IllegalArgumentException: Quantity must be positive!
|        at Purchase.<init> (#1:9)
|        at (#3:1)

jshell>
```

This is similar to the output we have received when trying to access the property of a `null`-object, or try to access an element of an array beyond its length. In both of these cases, an exception is thrown (respectively a `NullPointerException` and an `ArrayIndexOutOfBoundsException`). In this case, the `IllegalArgumentException` communicates that something was wrong with one of the inputs we received ("argument" is a fancy word for the input we pass into a function). More importantly, throwing an exception in the constructor prevents invalid objects from being constructed.

Although the validation in the constructor works, it is still possible to construct an invalid `Purchase` by changing a valid, positive quantity to an invalid, negative quantity after the object is constructed:

```
// Construct an invalid purchase, with a negative quantity:
```

```
Purchase purchase = new Purchase("Invalid purchase", 1, 1.00);
purchase.quantity = -10;

// We will RECEIVE money (EUR 10.00) by buying this!
System.out.println("Pay " + purchase.getSubTotal());
```

Is there any way to prevent this? Certainly!

We can add an *access modifier* to each property to indicate that these properties can’t be accessed from "outside" the class:

```
class Purchase {
    private String product;
    private int quantity;
    private double price;

    Purchase(String initialProduct, int initialQuantity, double initialPrice)
{

        if (initialQuantity <= 0) {
            throw new IllegalArgumentException("Quantity must be positive!");
        }

        if (initialPrice <= 0) {
            throw new IllegalArgumentException("Price must be positive!");
        }

        this.product = initialProduct;
        this.quantity = initialQuantity;
        this.price = initialPrice;
    }

    // Like before...
}
```

By adding the access modifier `private` to the properties of `Purchase` these properties can’t be accessed from "outside" the class anymore:

```
// Construct a valid purchase, with a positive quantity:
Purchase purchase = new Purchase("Invalid purchase", 1, 1.00);

// Try to change the quantity to negative:
purchase.quantity = -10;
```

Because we’re trying to access a `private` property, we will receive an error message (a compilation error this time) in return:

```
Error:
quantity has private access in Purchase
purchase.quantity = -10;
```

```
^---------------^
```

Properties with a `private` modifier can’t be changed from "outside" the class. Even retrieving their value is impossible. On the other hand, access from the "inside" the class (constructors and method of `Purchase`) is still possible. The method `getSubTotal` has no trouble accessing the properties:

```
// Construct a valid purchase, with a positive quantity:
Purchase purchase = new Purchase("Invalid purchase", 1, 1.00);

// Print the sub-total:
System.out.print("Sub-total: " + purchase.getSubTotal());
```

Likewise, the method `increaseQuantity` still works. We can use it to increase the quantity by one:

```
// Construct a valid purchase, with a positive quantity:
Purchase purchase = new Purchase("Invalid purchase", 1, 1.00);

// Add one more to shopping basket:
purchase.increaseQuantity();
```

Because we didn’t provide a method `decreaseQuantity` it is impossible to change the quantity of the purchase to something negative.

By making the properties `private` we prevented the `Purchase` class from moving from a valid state (quantity > 0) to an invalid state (quantity ¬ 0). All changes to `quantity` happen through a well-defined interface with the outside world that preserves the validity of a `Purchase`-object. This is one of the core principles of *encapsulation*.

Getters and setters

You may wonder if we haven’t thrown out baby with the bathwater. Nobody can change the `quantity` or the `price` to an invalid value, but we seem to have lost the ability to retreive those fields as well. The only thing we can do with a `purchase` is ask it for its sub-total…

To maintain encapsulation, it is common to provide dedicated methods to retrieve the properties you may want to expose:

```
class Purchase {
    private String product;
    private int quantity;
    private double price;

    String getProduct() {
        return this.product;
    }

    int getQuantity() {
        return this.quantity;
    }

    double getPrice() {
        return this.price;
    }

    // Like before...
}
```

Methods that expose the properties of a class (for reading) are collectively called *getters*. These methods typically start with the verb "get".

Although the class is starting to become very long, the safety that these getters provide is worth it to many (all?) Java developers.

If you want to preserve the ability to change some properties too, we could introduce some dedicated methods for that as well:

```
class Purchase {
    private String product;
    private int quantity;
    private double price;

    void setQuantity(int newQuantity) {
        if (newQuantity <= 0) {
            throw new IllegalArgumentException("Quantity must be positive!");
        }

        this.quantity = newQuantity;
    }

    void setPrice(double newPrice) {
        if (newPrice <= 0) {
            throw new IllegalArgumentException("Price must be positive!");
        }

        this.price = newPrice;
    }

    // Like before...
}
```

Like the constructor, the method `setQuantity` ensures that the quantity never becomes negative. Likewise, `setPrice` ensures that the price is always positive.

The methods to change (private) properties are collectively called *setters*. These methods typically start with the verb "set". Note that it is not necessary (or even expected) to provide setters for all properties. It makes little sense to change the name of the product of a `Purchase`. If we would like to buy four bananas instead of four cucumbers, many people would consider this a different purchase (that happens to have the same quantity).

Public, protected an default access

We’ve learned how to restrict access to properties by using the `private` access modifier. There are other access modifiers that you should know about. Although it is less common, you may sometimes encounter a property marked with the access modifier `public` (the `length` property on an array is one such example). The access modifier `public` is the opposite of `private`. Where `private` makes the property invisible from outside the class, the `public` modifier ensures that everybody can see and change the property. Because you often don’t want to allow people to change these properties at whim (encapsulation!) you won’t often see properties marked `public`.

You must be wondering what the difference is between a `public` property and a property with no access modifier at all (which we say has "default access"). From what we have observed so far, it seems like all properties without an access modifier (neither `private` nor `public`) are accessible from outside the class too. Although they seem to be similar, properties without an access modifier are only accessible from classes in the same package (files in the same folder on your hard disk) whereas `public` properties are accessible from everywhere.

Besides `private` and `public` there is one last access modifier that we haven’t discussed yet: `protected`. Protected properties behave a lot like properties with no access modifier at all. The difference between public access, protected access and default access (no access modifier) will be discussed later in this course, when we talk about inheritance.

If you are confused, don’t worry! In the vast majority of cases, either `private` (access from "inside" the class) or `public` (access from both "inside" and "outside" the class) is the correct choice. From now on, we will be adding either `private` or `public` to all of our properties, and we suggest you do the same!

Besides properties, we can also mark methods and even constructors as `private`, `public` or `protected`. For the `Person` from before, it seems like a good idea to allow everyone to use the getters `getName`, `getFirstName`, `getLastName` and `getAge`, so we mark them as `public`:

```
class Person {
    private String firstName;
    private String lastName;
    private int age;

    public String getName() {
        return this.firstName + " " + this.lastName;
    }

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }
    public int getAge() {
        return this.age;
    }
}
```

JavaBeans standard

There are some conventions on how to name the public getters and setters that you might like to add to your classes. For a property named `p` with type `T` the getters and setters should look like

```
// Getter
public T getP() { ... }

// Setter
public void setP() { ... }
```

When `T` is `boolean` you are also free to use

```
public boolean isP() { ... }
```

Especially `if`-statements can benefit from switching from prefix `get` to `is` (for a `boolean`):

```
if (cat.isHappy()) {
    cat.purr();
}
```

Module: Software development

Programming and software development

In this training, we are not teaching you to become programmers. We are teaching you to become Software Developers. What is the difference? The difference is the scale. A programmer might make a single application by themselves. A Software Developer works together with hundreds of people on a code base that is too large for any single person to fit in their heads. It is much like the difference between you building a dog house for your family's Beagle on a Sunday afternoon and a construction worker working on a multi-story apartment building over the course of many years. The latter can't know all the details about every aspect of the building (foundation, plumbing, wiring, escape routes) and likely does not know all the people involved on the project.

This difference has consequences for the way you write your code. Although a programmer might know exactly how each piece of their code is used (and has direct control over that!), a Software Developer has no idea who might be calling their functions, and how.

Let's see how our little `Purchase` class might be called, as more and more people become aware of it and start using it.

- Of course we have the code for the shopping system, which is concerned with placing products in your shopping basket and increasing (or decreasing) their quantity.
- We also must consider the checkout system, which deals with payment of the products in the shopping basket. It does not need to know exactly what you bought, and how many, but it does need to know what it is going to cost you!
- People from the inventory system started using the class to keep track of the products that are still in stock. The products that you bought need to be replenished!
- The marketing system calls out to your code to be able to send you the customer promotional e-mails based on their recent purchases.
- Your code is in use in the accounting system, to comply with tax law.
- Believe it or not, but the programmers responsible for store design need your class as well, to determine which products are often bought together (so they can be placed closer together in the store).

On the one hand, this is great. By writing a small class, you were able to help a lot of people to achieve their goals. Software is almost unique in that regard. With absolutely no extra effort on your part, your code be used in hundreds of different circumstances and applications.

On the other hand, you have taken on quite some responsibility. Probably without even realizing it! The more people use your class, the harder it is for you to change it.

Imagine that one day you decided you did not like the name of the method `getQuantity` anymore. After all "quantity" sounds so dry and mathematical. Perhaps `getNumberOfProductsPurchased` is more to-the-point? Ten minutes after changing the name of the method (and feeling darn proud of yourself too!) you start to get a lot of angry e-mails.

Dave, who is working on the inventory system, wrote a particularly long and nasty e-mail. It turns out that Dave was calling the method `getQuantity()` about fifty times. He now has to change all those fifty calls from `getQuantity` to `getNumberOfProducts` for no apparent benefit to him! In fact, he liked `getQuantity` better!

Tanya from the marketing system also sent you an e-mail. She just started last week, and she can’t figure out why she can’t get the system to compile anymore.

From the corner of your eye you see an e-mail from your boss, asking you if you realize how much time and money your silly little change is costing the company! Ouch!

We’ve seen one of the benefits of encapsulation. By making properties `private` and by restricting state transitions through well-defined setters, we can avoid ending up with an invalid `Purchase`. The next section is about another benefit: to make it easy to change a class without having to change code all over the place.

Designing classes

Recap on encapsulation

Remember our friend *encapsulation*? It ensures that it’s possible to do this:

```
double price = purchase.getPrice();
```

But not this:

```
purchase.price = 0.00; // Now it's free!
```

It also allows you to change the internals of your class without the caller noticing. So you can keep doing this:

```
double price = purchase.getPrice();
```

And it doesn’t matter if your `Purchase` class looks like this:

```
class Purchase {
    private double price;

    public double getPrice() {
        return price;
    }
}
```

Or this:

```
class Purchase {
    private int priceInCents;

    public double getPrice() {
        return priceInCents / 100;
    }
}
```

Or even this:

```
class Purchase {
    // No field at all!

    public double getPrice() {
        return readPriceFromFileOnDisk();
    }
}
```

Nice! It gives us the freedom to change our code, without giving anyone else the freedom to stick their nose where it doesn’t belong.

What can a client use?

How do we decide what a client can use and see, and what not? Unfortunately that’s more of an art than a science: there are no strict rules. There are, however, rules of thumb. The first one is: try to expose methods, not fields. Methods are more flexible, and allow for more changes. Compare these two classes:

```
class Purchase {
    public int priceInCents;
}
```

and

```
class Purchase {
    private int priceInCents;

    public int getPriceInCents() {
        return priceInCents;
    }
}
```

If you decide you want to change your mind and store the price in euros instead, the following code is much easier to modify:

```
class Purchase {
    private double price;

    public double getPrice() {
        return price;
    }

    public int getPriceInCents() {
        return Math.round(price * 100);
    }
}
```

This way, you can keep the original `getPriceInCents()` around for compatibility, but move forward with the price in euros as well. In the first example, this would have been much more difficult.

Only expose what you need

Another rule of thumb: expose only what you can get away with, and no more.

If later it turns out that a client needs something that you didn’t expose, it’s easy to add a getter. But if later it turns out that you exposed something that you now regret, it can be quite hard to fix that.

For instance, let’s say that your `Purchase` class looks like this:

```
class Purchase {
    private int priceInEuros;
    private int priceRemainderInCents;

    public double getPrice() {
        return priceInEuros + (priceRemainderInCents / 100);
    }

    // Getters included without thinking too hard about
    // it in the early stage of the project:

    public int getPriceInEuros() {
        return priceInEuros;
    }

    public int getPriceRemainderInCents() {
        return priceRemainderInCents;
    }
}
```

Obviously, you would like to move away from this nonsense, refactor to a `double price` field, and remove those methods.

But first, you need to make sure that no clients use those `getPriceInEuros()` and `getPriceRemainderInCents()` methods anymore. However, it turns out that `getPriceInEuros()` is used 4562 times, mostly by the tax-related code from the Accountancy team on the 6th floor, because as it turns out, the Dutch tax authorities round everything down to the nearest whole euro and they thought, hey this is a convenient method for us to use!

So, basically, now you have to support this annoying `getPriceInEuros()` method forever. Even worse: when the tax authority changes its rules and says that now the rounding must be done to the nearest whole euro, instead of always rounding down, the Accountancy team will come a-knockin' on your door.

Immutability

Another rule of thumb is to design your classes for immutability, which is fancy programmer-speak for classes whose internal values you can’t change.

So, instead of this:

```
class Purchase {
    private double price;

    public double getPrice() {
        return price;
    }
    public double setPrice(double newValue) {
        this.price = newValue;
    }
}
```

Do this:

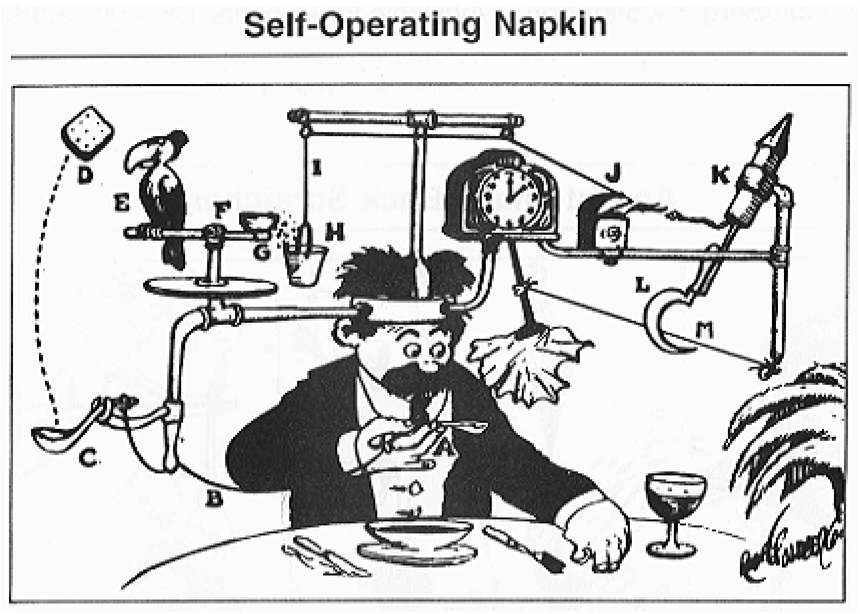
```
class Purchase {
    private final double price;

    public double getPrice() {
        return price;
    }
}
```

As you can see, the second example doesn't have a `setPrice()` method, so a client can't change the price of a value. And because there's no setter method, you can now also make the property `final`, which also tells the compiler that the value can't change so it will give you errors when you accidentally try to assign a new value.

Why would we want that? First of all, there's rule of thumb number 1: don't expose what you don't need to expose. If you don't need a setter, don't write one, or else you'll have to support it forever.

The second reason goes deeper. It turns out that code is often easier to understand if object's can't change their values. That doesn't just apply to you, the reader of the code, but also to the compiler. This has to do with *concurrency*, an advanced programming topic outside the scope of this training. But it also has to do with programming in general: the less moving parts in your code, the easier it is to understand. This is an insight that can't really be taught; you have to experience it for yourself over time. However, you can compare it with this picture, which also has a lot of moving parts:



Note also, that some of the moving parts will only work once: as soon as their state changes, you can't use them again. Can you identify those parts?

Reference escapism

We got a good grasp on what to keep `public` and what to make `private`, but there are still plenty of escape hatches through which creative-thinkers can see and modify what we *think* is now hidden from the outside world! In what follows, we'll investigate how we can close down all the holes through which malicious users can change what should not be changed!

Let's say we have a class with a field of type array and a corresponding getter method:

```
class Receipt {
    private final double[] amounts;

    public Receipt(double[] amounts) {
        this.amounts = amounts;
    }

    public double[] getAmounts() {
        return amounts;
    }
}
```

Let's say we initialize it like this:

```
double[] initial = { 1.0, 2.0, 3.0 };
Receipt receipt = new Receipt(initial);
```

When we call `getAmounts()`, we get the expected value:

```
receipt.getAmounts();
// double[3] { 1.0, 2.0, 3.0 }
```

Let's say we make a modification:

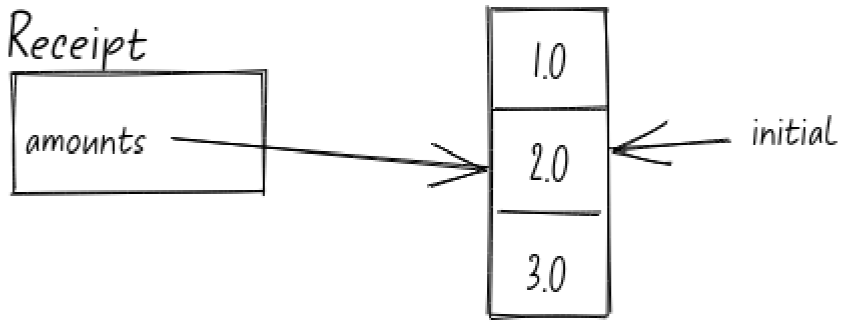
```
double[] initial = { 1.0, 2.0, 3.0 };
Receipt receipt = new Receipt(initial);
initial[1] = 42.0;
```

What will `getAmounts()` return? The answer might surprise you:

```
receipt.getAmounts();
// double[3] { 1.0, 42.0, 3.0 }
```

What happened? We made the `amounts` field private and final, we neatly encapsulated everything with a constructor and a getter! What gives?

In memory, this is what happened:



It turns out that both `initial` and `receipt.amounts` point to the same array in the JVM's memory, which means that if you change one of them, you change them both. In more formal programmer-speak: the reference to the array has escaped the boundaries of the Product class. We don't like that!

And it gets worse:

```
double[] initial = { 1.0, 2.0, 3.0 };
Receipt receipt = new Receipt(initial);
double[] observed = receipt.getAmounts();
observed[1] = 42.0;
```

Again, the surprising answer:

```
receipt.getAmounts();
// double[3] { 1.0, 42.0, 3.0 }
```

This happens because the value returned by `getAmounts()` is still a reference to the same array in memory.

How do we prevent this? One way is to use a primitive value instead of an object, since primitives get copied instead of referred to. But that isn't always an option, of course.

The other way is to make sure that your class is immutable. If you can't change the content of a class, it doesn't matter if a reference to it escapes.

What if you do require a mutable object, and don't want a reference to escape? Then you have to program defensively.

Defensive programming

Defensive programming means that you assume that your clients will attack you, and you try to prevent these attacks. The best way to do that, is to make a copy of every reference that comes in and goes out. For arrays, we can use `Arrays.copyOf()`:

```
import java.util.Arrays;
```

```
class Receipt {
    private final double[] amounts;

    public Receipt(double[] amounts) {
        this.amounts = Arrays.copyOf(amounts, amounts.length);
    }

    public double[] getAmounts() {
        return Arrays.copyOf(amounts, amounts.length);
    }
}
```

You can verify for yourself that this works for constructors:

```
double[] initial = { 1.0, 2.0, 3.0 };
Receipt receipt = new Receipt(initial);
initial[1] = 42.0;

receipt.getAmounts();
// double[3] { 1.0, 2.0, 3.0 }
```

And for getters:

```
double[] initial = { 1.0, 2.0, 3.0 };
Receipt receipt = new Receipt(initial);
double[] observed = receipt.getAmounts();
observed[1] = 42.0;

receipt.getAmounts();
// double[3] { 1.0, 2.0, 3.0 }
```

This works for arrays. How about other reference types? Well…it depends. Many mutable reference types in the Java APIs implement the `Cloneable` interface:

```
GregorianCalendar time = new GregorianCalendar();
GregorianCalendar copy = time.clone();
```

So…isn't it inefficient to copy objects all the time? Well…yeah. Yes it is. It's the price we have to pay in order to use these types.

Fortunately, instead of copying, some types can be *wrapped*. For instance, Lists can be wrapped:

```
List<String> strings = new ArrayList<>();
List<String> copy = Collections.umodifiableList(copy);
```

While `strings` is an instance of the `ArrayList` class, `copy` is not. But both still implement the `List` interface. The unmodifiable list class implements all the methods in `List`, and for most of them, it just delegates to the underlying `ArrayList`. But for the methods that change things, it will throw an exception. This is what that looks like:

```
// Real implementation is different
public class UnmodifiableList<T> implements List<T> {
    private final List<T> underlying;

    public UnmodifiableList(List<T> underlying) {
        this.underlying = underlying;
    }

    public T get(int index) {
        return underlying.get(index);
    }

    public void add(T element) {
        throws new UnsupportedOperationException();
    }
}
```

This doesn't cost much extra memory, barely costs extra time to execute, and solves the problem nicely.

Exercises

- Fill in access modifiers in the following classes, and explain your choices.

```
class Person {
    final String name;
    final int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    String getName() {
        return name;
    }

    int getAge() {
        return age;
    }
}
```

- Prove by writing a Java program that reference escapism really happens.
- Prove by writing a Java program that with primitive values, reference escapism doesn't happen.
- Can we have reference escapism with `java.util.GregorianCalendar`? And with `java.time.LocalDateTime`? Write programs to prove it.
- Fix the vulnerable programs above using defensive programming techniques.