Programming Fundamentals 2

Pierre Talbot

23 February 2021

University of Luxembourg

Chapter III. Object-Oriented Programming

Object type

Object = record + functions.

The functions are bundled together with the record, in a same structure.
In this context:

- the fields of the record are called *attributes*.
- the functions are called *methods*.

Characteristics of object-oriented programming

We can find 5 main characteristics of the object-oriented paradigm[1]:

- **Encapsulation**: attributes are not accessible from outside of the object.
- **Inheritance** (§4): An object I reuses the implementation of the methods of an object J, through class and *subclassing*.
- **Subtyping** (§6): If an object I contains, at least, all the attributes and method of J, then we can pass I as argument everywhere J is expected.
- **Multiple representations** (§8): a same set of methods (an *interface*) can be implemented by two different objects.
- **Open recursion**: A method on I can invokes any another method on the current object I.

[1] *Types and Programming Languages*, Benjamin C. Pierce.

Encapsulation

Concert app: from records to objects

Imagine that a newcomer in your team is asked to implement a function to cancel concerts:

```
public class ConcertPlanning {
  // ...

  // Start and end time of cancelled concerts are set to 0 and -1.
  public static void cancelConcert(ConcertPlanning planning, int concertIndex) {
    planning.concerts[concertIndex].startTime = 0;
    planning.concerts[concertIndex].endTime = -1;
  }
}
```

In this case, he decides that the time interval `[0..-1]` represents empty (cancelled) concert.

What could be the problem with this code?

The problem

The invariant `startTime < endTime` of the record `Concert` is violated. When computing `totalTimeConcert`, the total time will not be correct since the duration of each cancelled concert will be $-1 - 0 = -1$.

The problem is that anybody can access and modify the attributes of `Concert`. Imagine in large projects (hundreds of files): it is almost impossible to enforce the invariants everywhere, and bugs are easily introduced.

The solution: Encapsulation

A trait of object, namely *encapsulation*, helps in solving this problem. The previous code can be rewritten:

```
public class Concert {
  private int startTime;
  private int endTime;
  // ...
}
```

We set the attributes to `private`, we forbid any function external to this object to read or modify them. Any change to these attributes are located in the file `Concert.java`, so it is easier to enforce invariants.

```
public class ConcertPlanning {
  public static void cancelConcert(ConcertPlanning planning, int concertIndex) {
    // Won't work anymore! The assert will be triggered.
    planning.concerts[concertIndex] = Concert.makeConcert(0, -1);
  }
}
```

A paradigm shift

Due to encapsulation, there is a shift from data-centric programming to behavior-based programming:

- We do not reason on data anymore: these are hidden in classes that we cannot necessarily modify or even look at!
- We are only interested by **the services a class can give us**.
- These services are the methods of the class annotated with `public`.

Imperative / Procedural

```java
public class Concert {
  public int startTime;
  public int endTime;

  public static Concert makeConcert(int startTime, int endTime)
  {
    assert startTime < endTime;
    Concert c = new Concert();
    c.startTime = startTime;
    c.endTime = endTime;
  }

  public static int duration(Concert concert) {
    return concert.endTime - concert.startTime;
  }
}
```

Object-oriented

```java
public class Concert {
  private int startTime;
  private int endTime;

  public Concert(int startTime, int endTime)
  {
    assert startTime < endTime;
    this.startTime = startTime;
    this.endTime = endTime;
  }

  public int duration() {
    return this.endTime - this.startTime;
  }
}
```

From records to objects

- `makeConcert` becomes the constructor `Concert` (**must have the same name as the class!**).
- Static methods become method (removes the `static` qualifier).
- The argument `Concert concert` passed to static methods, is now referred as **this** and is passed implicitly to the methods.

Using objects

Imperative / Procedural

```
public class ConcertApp {
  public static void main(String[] args) {
    Concert c1 = Concert.makeConcert(18, 19);
    Concert c2 = Concert.makeConcert(20, 22);
    ConcertPlanning planning =
      ConcertPlanning.makeConcertPlanning();
    ConcertPlanning.addConcert(planning, c1);
    ConcertPlanning.addConcert(planning, c2);
    System.out.println("Total duration of the concerts: " +
      ConcertPlanning.totalTimeConcert(planning));
  }
}
```

Object-oriented

```
public class ConcertApp {
  public static void main(String[] args) {
    Concert c1 = new Concert(18, 19);
    Concert c2 = new Concert(20, 22);
    ConcertPlanning planning =
      new ConcertPlanning();
    planning.addConcert(c1);
    planning.addConcert(c2);
    System.out.println("Total duration of the concerts: " +
      planning.totalTimeConcert());
  }
}
```

From records to objects

- The constructor is called with `new Concert(18, 19)`.
- `new Concert()` actually calls the *constructor by default*. It initializes the attributes to some default values (0 for integers, `null` for objects, ...).

Summary of objects

```
public class Concert {
  private int startTime;
  private int endTime;

  public Concert(int startTime, int endTime) {
    assert startTime < endTime;
    this.startTime = startTime;
    this.endTime = endTime;
  }

  public int duration() {
    return this.endTime - this.startTime;
  }
}
```

Attributes of the class (`startTime`, `endTime`)

Constructor (`Concert(int startTime, int endTime)`)

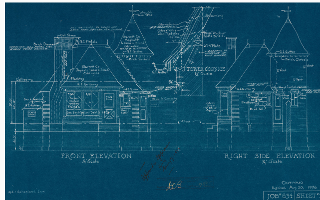
Methods of the class (`duration()`)

```
public class ConcertApp {
  public static void main(String[] args) {
    Concert c1 = new Concert(1, 2);
    Concert c2 = new Concert(3, 4);
    System.out.println("Duration c1: " + c1.duration());
    System.out.println("Duration c2: " + c2.duration());
  }
}
```

Constructor calls (`new Concert(1, 2)`, `new Concert(3, 4)`)

Method calls (`c1.duration()`, `c2.duration()`)

Vocabulary: Class vs Instance vs Object



- A class is a “blueprint”, a description of something,
  e.g., the plan of a house.
- An object is the concrete realization of a class,
  e.g., a house with a customized colour.
- An instance is a relationship between object and class,
  e.g., the object `new Band("Daft Punk")` is an instance of the class `Band`.

A note on encapsulation

Encapsulation is not proper to the object-oriented paradigm. It exists in other paradigms, such as *functional programming* which often relies on modules to enforce encapsulation.
Moreover, some (non-mainstream) object-oriented languages do not enforce encapsulation (*e.g.*, CLOS).

Static methods and attributes

Static methods

Functions that do not compute on an object in particular.

```
public class Math {
  public static int max(int a, int b) {
    return a > b ? a : b;
  }
}
```

Consequently, in `static` methods, `this` is not passed as an argument to the function, thus you do not have access to the class attributes.

Advice

For now, you should stay away from static methods. They encourage an imperative/procedural style, instead of object-oriented style (see the examples above).

Static attributes

You can annotate an attribute with the keyword `static`. It creates a variable that is shared and global to all objects of this class.

We present two main use cases for static attributes:

1. Create unique identifiers.
2. Declare constant values.

Use case 1: unique identifier (UID)

Consider the following class:

```
public class Laptop {
  int product_uid;
  int ram;
  int hard_drive_memory;
  // ...
}
```

The attribute `product_uid` should be unique, each laptop should have a different one. How to ensure that?

Use case 1: unique identifier

Add a static attribute to count the number of objects created.

```
public class Laptop {
  static int uids = 0;
  int product_uid;
  int ram;
  int hard_drive_memory;
  // ...
  public Laptop() {
    product_uid = uids;
    uids = uids + 1;
    //...
  }
}
```

Each time we will construct an object with `new Laptop()`, the static variable `uids` will be incremented, so the next invocation of the constructor will initialize `product_uid` to a different UID.

Use case 1: unique identifier

```
public class Laptop {
  static int uids = 0;
  int product_uid;
  int ram;
  public Laptop(int ram) {
    this.product_uid = uids;
    this.ram = ram;
    uids = uids + 1;
  }
}

public class LaptopCreator {
  public static void main(String[] args) {
    Laptop l1 = new Laptop(500);
    Laptop l2 = new Laptop(1000);
  }
}
```

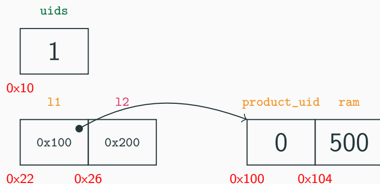
uids

0

0x10

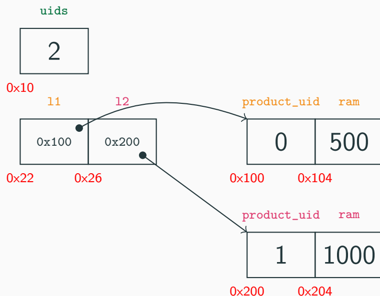
Use case 1: unique identifier

```
public class Laptop {
  static int uids = 0;
  int product_uid;
  int ram;
  public Laptop(int ram) {
    this.product_uid = uids;
    this.ram = ram;
    uids = uids + 1;
  }
}

public class LaptopCreator {
  public static void main(String[] args) {
    Laptop l1 = new Laptop(500);
    Laptop l2 = new Laptop(1000);
  }
}
```



Use case 1: unique identifier

```
public class Laptop {
  static int uids = 0;
  int product_uid;
  int ram;
  public Laptop(int ram) {
    this.product_uid = uids;
    this.ram = ram;
    uids = uids + 1;
  }
}

public class LaptopCreator {
  public static void main(String[] args) {
    Laptop l1 = new Laptop(500);
    Laptop l2 = new Laptop(1000);
  }
}
```



Use case 2: constant values

```java
public class Checkers {
  static final int WHITE = 0;
  static final int BLACK = 1;
  int[][] grid;
  //...
  public printGrid() {
    for(int i = 0; i < grid.length; ++i) {
      for(int j = 0; j < grid[i].length; ++j) {
        if(grid[i][j] == WHITE) {
          System.out.print("\u25CB");
        }
        else {
          System.out.print("\u2B24");
        }
      }
      System.out.print("\n"); }}}
```

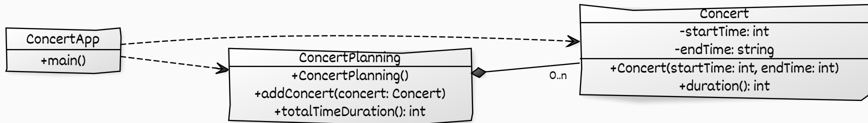
Final keyword

- `final int x` makes the content of the cell immutable (we cannot change it anymore).
- `final Person p` only means that we cannot assign *p* two times with `p = new Person()`. But we can still change what's inside of `p` with `p.name = "Alfred"`, as many times as we want (unless `name` is final as well).

Has-a relationships

Drawing class diagrams

When thinking about the design of an object-oriented application, it is often useful to draw a diagram showing how the classes are interconnected.



Unified Modeling Language (UML)

UML is a formalism to draw diagrams representing your application.

Class diagram

- A class is a box with three parts: the name, the attributes and the methods.
- *Visibility*: The symbols `-`, `+` or `#` in front of an attribute or method, respectively for `private`, `public`, `protected`.
- `public float m(int x)` is written `+m(x:int):float` in UML. The return type is written after the method.

Relationships between classes

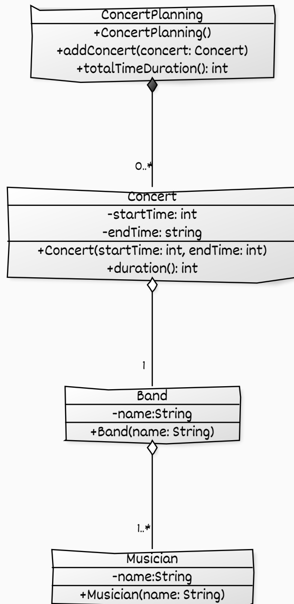
Relationships (simplified view)

- **Dependency** (dashed arrow): Class A uses class B in a method, but do not store object of type B as attributes.
- **Association** (plain line): Class A has an attribute of class B.
- **Aggregation** (arrow with empty diamond): Association + class B does not depend on A to live.
- **Composition** (arrow with plain diamond): Association + class B depends on A to live.

Each of these relationships can be annotated with *multiplicities*:

- `1`: Exactly one instance.
- `0..1`: Zero or one instance.
- `0..*`: Zero or more instances.
- `1..*`: One or more instances.

Aggregation vs Composition



More on UML relationships

Association, aggregation and composition all represent a connection between A and B such that B is an attribute of A.

- You don't need to (and probably shouldn't) specify all associations.
- You can use attributes instead of associations, if the link between the two classes is not important for what you want to show.
- An aggregation implies that an object is a part of another, *e.g.*, the band and the musician.
- An association would be, for instance, between the band and the musical style of the band.

UML should help you to explain the architecture of your project in a compact and informative manner.

Inheritance (is-a relationship)

The manga example

Suppose the following class:

```
public class Book {
  private String title;
  private Person writer;
  private int pages;
  public Book(String title, Person writer, int pages) {
    this.title = title;
    this.writer = writer;
    this.pages = pages;
  }
  public int numberOfPages() { return pages; }
  ...
}
```

Imagine we want to represent a *manga* (or comics). It is a book, but with at least one additional characteristic: the artist who drew it.

Let's pause a second, and think how we could do it with what we learnt.

Solution #1: Total rewrite

Perhaps the easiest solution is to create a totally new class `Manga`, not related to `Book`:

```
public class Manga {
  private String title;
  private Person writer;
  private Person artist;
  private int pages;
  public Manga(String title, Person writer,
   Person artist, int pages)
  {
    this.title = title;
    this.writer = writer;
    this.artist = artist;
    this.pages = pages;
  }
  public int numberOfPages() { return pages; }
  ...
}
```

The problem is quite obvious: we have two classes (`Book` and `Manga`) with attributes and methods in common. *Don't repeat yourself (DRY)!*

Solution #2: Forwarding

A better solution is to store Book as an attribute of Manga:

```
public class Manga {
  private Book book;
  private Person artist;
  public Manga(String title, Person writer,
   Person artist, int pages)
  {
    this.book = new Book(title, writer, pages);
    this.artist = artist;
  }
  public int numberOfPages() { return book.numberOfPages(); }
  ...
}
```

We create *forwarding methods* in Manga to call the methods we need from Book. At least, if a bug is corrected in Book, we automatically benefit from the correction.

Solution #3: Inheritance

The concept of inheritance allows to “import” the attributes and methods from a *base class* `Book` into a *subclass* `Manga`.

```
public class Manga extends Book {
  private Person artist;
  public Manga(String title, Person writer,
   Person artist, int pages)
  {
    super(title, writer, pages);
    this.artist = artist;
  }
}

public class BookApp {
  public static void main(String[] args) {
    Manga snk = new Manga("Shingeki no kyogin", ..);
    System.out.println("Number of pages of SNK: "
      + snk.numberOfPages());
  }
}
```

Actually, it is not always simple to choose between solution #2 or #3 (cf. `https://en.wikipedia.org/wiki/Composition_over_inheritance`).

Inheritance subtlety #1: `super` keyword

The `super` keyword refers to the base class we inherited from. It can be used in three ways:

1. Call the constructor of the base class.
2. Refer to the public or protected attributes of the base class.
3. Refer to the public or protected methods of the base class.

```java
public class Manga extends Book {
  private Person artist;
  public Manga(String title, Person writer,
   Person artist, int pages)
  {
    super(title, writer, pages);
    this.artist = artist;
  }
  public int blackAndWhitePages() {
    // We don't count the cover, two middle pages and last page.
    // (let's suppose it's really like that...)
    return super.numberOfPages() - 4;
  }
}
```

Inheritance subtlety #2: `protected` keyword

You can only use in the subclass `Manga` the attributes and methods with a `public` or `protected` visibility qualifier.

```
public class Manga extends Book {
  private Person artist;
  public Manga(String title, Person writer,
   Person artist, int pages)
  {
    super(title, writer, pages);
    this.artist = artist;
  }
  public void addOnePage() {
    // Will not work because ‘pages’ is private.
    super.pages = super.pages + 1;
  }
}
```

Just change `private` to `protected` if it makes sense:

```
public class Book {
  protected int pages;
  //...
}
```

Review and more

Preliminary

Here a example of inheritance we will reuse later:

```
class Weapon {
  protected double damage;
  public Weapon(double damage) {
    this.damage = damage;
  }
}

class Axe extends Weapon {
  private static final double DAMAGE = 10;
  public Axe() {
    super(DAMAGE);
  }
}

class Hammer extends Weapon {
  private static final double DAMAGE = 20;
  public Hammer() {
    super(DAMAGE);
  }
}
```

Quiz

- What are the attributes proper to an object and proper to a class?
- What is a constructor?
- What is `super` and `this`?
- Why does a constructor call must be the first instruction?
- What happens if we do not call `super` in the constructor of a subclass?

Solutions

Object vs Class

- Object attribute: `Weapon.damage`.
- An object attribute is different each time we create a new object, *e.g.*, `new Book()`.
- Class attributes: `Axe.DAMAGE` and `Hammer.DAMAGE`.
- A class attribute is shared by all objects, it is a kind of global variable.

Solutions

Constructor

- A constructor is not a method of the object. It is a function that construct the object.
- Therefore, constructors are not inherited (think: how could the constructor `Book` initialize the attribute `artist` in `Manga`?)

Solutions

`super` and `this` : 2 usages

- **During object construction**: call a constructor of the parent's class (`super(arg1, arg2)`) or of the current class (`this(arg1, arg2)`).
- The last case implies that a class can have more than 1 constructor (more on that in Chapter 4).
- **During method call**: `super` gives us a reference to the parent's object, and `this` to the current object, in order to explicitly access an attribute or method (`this.damage`).

Pitfall: shadowing

```
protected double damage;
public Weapon(double damage) {
  this.damage = damage;
}
```

Note: *Shadowing* of the attribute `damage` by the local variable, this is why we need to add `this.damage`.

Solutions

Call constructor of the base class first!

- We first build the base class, before building the subclass.
- Sometimes a bit cumbersome if you want to "prepare" the argument of the base class constructor.
  $\Rightarrow$ In that case, use a static method to prepare the arguments (see also *Builder pattern* in Chapter 1x).

Solutions

`super` not explicitly called

- The line `super();` is automatically inserted as the first line of the constructor body of the subclass.
- There is a compile-time error if there is no default constructor (constructor without arguments) in the base class.