Lecture 2: Object Oriented Programming

Pablo Oliveira <pablo@siffez.org>

Outline

1 References
2 Inheritance
3 Encapsulation
4 Polymorphism
5 Interfaces
6 Summary
Introduction: Object Oriented Programming

- In the last lecture we learned that we can structure programs using objects of many classes.
- In this lecture we will examine OOP concepts in more detail:
  - **constructors**: creating new objects.
  - **references**: designating objects.
  - **inheritance**: creating families of classes.
  - **encapsulation**: hiding implementation.
  - **polymorphism**: factorizing common behaviours.
  - **interfaces**: behavioral contracts.

Constructors: creating a new object

**Definition**
Constructors are special methods that are called to create a new instance of their class.

```java
class BankAccount {
    int balance;
    BankAccount () {
        balance = 0;
    }
    BankAccount (int initialDeposit) {
        balance = initialDeposit;
    }
}

account1 = new BankAccount();
account2 = new BankAccount(100);
```
When a variable is assigned a primitive type it contains a value.
When assigned an object, array or string, it contains a reference to the data.
If a is copied or passed, old and new references point to the same original object.

```java
static void changeValues (int anArray[], int value){
anArray[0] = 42;
value = 42;
}
public static void main (String args[]){
    int v = 0; int [] a = {0,0};
    System.out.println(v + " " + a[0] + " " + a[1]);
    changeValues(a,v);
    System.out.println(v + " " + a[0] + " " + a[1]);
}
output :
0 0 0
0 42 0
```
Immutability

- String are a special case, because they are immutable (cannot be changed).
- When you change a String a new different String is created and the characters of the original one are copied.
- For performance: do not build a string with concatenation, use StringBuilder.

```java
public static void main(String args[]) {
    String s1 = "hello";
    String s2 = s1;
    s1 = s1 + "!";
    System.out.println(s1 + " " + s2);
}
```

output:
hello! hello

Outline

1 References
2 Inheritance
3 Encapsulation
4 Polymorphism
5 Interfaces
6 Summary
Q: Remember our turtle? It could turn and advance. But we want a new class Crab that advances sideways ...

- We could write a new class Crab, but there would a lot of code in common with Turtle (which makes the code base difficult to maintain).
- We are going to use inheritance.
- Inheritance makes it possible to create a subclass that inherits the properties of its ancestor or superclass.

```
class Animal {
    Color color;
    Position position;
    double rotation;

    void turn(double angle) {};
    void advance() {};
}

class Crab extends Animal{
    void advance() {
        /* code for moving sideways */
    }
}
```

Crab crab = new Crab();
crab.color = Color.BLUE;
crab.advance();
overriding and hiding

What we just did with method advance is called **overriding**.

- When we call `crab.advance()` the crab’s advance is called!
- The animal’s advance has been overrided.
- If a method is not overriden, the superclass’ is used (here `crab.turn(10);` would call Animal’s turn implementation.
- the **final** keyword in a method declaration indicates that the method cannot be overridden.

overriding a static method or a variable is called **hiding**, because the new static implementation or variable *hides* the old one, doing this is usually a bad idea.

---

this and super

- for a given class **this** represents the current class and **super** the superclass.
- **super** is used to call overriden superclass’ methods.

```java
class Animal {
    void advance();
}

class Crab extends Animal{
    String name;
    advance() {
        this.turn(90);
        super.advance();
        this.turn(-90);
    }
}
```
In java all the classes are subclasses of the Object class. 
- A subclass constructor will always call a superclass constructor. 
- If a class possess no constructor, an empty one with no parameters is implicit. 
- Every constructor of a subclass call the no-parameters superclass constructor. 
- But we can control this with super and this keywords.

```java
abstract class Animal {
    Position position;
    double rotation;

    Animal(Position position, double rotation) {
        this.position = position;
        this.rotation = rotation;
    }
}

class Crab extends Animal{
    String name;
    Crab(Position position) {
        super(position, 90);
    }
    Crab(Position position, String name) {
        this(position);
        this.name = name;
    }
}
```
**abstract methods**

Suppose we add birds to our class hierarchy.

- birds and crabs do not move the same way... there is no common implementation for advance that we can put in Animals.
- we could create an empty `advance()` in the Animal class and override it in Bird and Crab.
- Yet, another programer could add a new subclass and forget to implement the `advance()` method.
- Thus, we use **abstract methods**.

**Definition**

- An abstract method is a method which has no implementation.
- An abstract class is a class with abstract methods.
- It is mandatory for all the non-abstract subclasses to override all the abstract methods.
- An abstract class cannot be instantiated.

```java
abstract class Animal {
    Position position;
    double rotation;

    abstract void advance();
}

class Crab extends Animal{
    String name;
    void advance() {
        /* crab moves */
    }
}

Animal a = new Animal(); // COMPILATION ERROR
Crab c = new Crab(); // Works!
```
Encapsulation

Definition

Encapsulation is the act of hiding properties and methods inside a class.

- This allows to protect classes from unexpected side-effects from the outside.
- It also enforces implementation agnostic programming, which is a good idea.
Packages

Definition

- A package is a group of classes.
- Packages define a namespace.
- Classes in the same package share the same namespace.

```java
package Animals;
class Animal{}
class Crab{}
import Animals.Crab;
import Animals.*;
class MyProgram{}
```

Acces modifiers

In java encapsultation is obtained through acces/visibility modifiers.

- Classes can be public, visible by everyone or without modifier in which case they are only visible inside their package (a group of classes).
- Class members (variables and methods) can have 4 modifiers with different degrees of visibility.

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Class</th>
<th>Package</th>
<th>Subclass</th>
<th>World</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>protected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>no modifier</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>private</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
```java
package animals;

class Animal {
    private double rotation;
    public void turn(double angle)
    {
        position += angle;
    }
}

class Crab extends Animal {
    public void turnBack() {
        turn(180);       // legal
        rotation += 180; // illegal
    }
}
```

Outline

1. References
2. Inheritance
3. Encapsulation
4. Polymorphism
5. Interfaces
6. Summary
**Q**: How to make a group of animals advance?

- We want to make a group of animals (crabs and turtles) advance at the same time.
- We need a container for all of them, what is the container type?

**Nightmare**

```java
int numberCrabs; int numberTurtles;
Crab[] crabs;
Turtle[] turtles;

moveAllAnimals () {
    for (int i=0; i < numberCrabs; i++)
        crabs[i].advance();
    for (int i=0; i < numberTurtles; i++)
        turtles[i].advance();
}

void addCrab (Crab c) {crabs[numberCrabs++]=c;}
void addTurtle (Turtle t) {
turtles[numberTurtles++]=t;}

addCrab(new Crab());
addTurtle(new Turtle());
```

**Polymorphism**

Use Polymorphism, or the capacity to treat an instance as one of its super classes

```java
int numberAnimals;
Animal[] animals;

void moveAllAnimals(){
    for (int i=0; i < numberAnimals; i++)
        animals[i].advance();
}

void addAnimal(Animal a) {
    animals[numberAnimals++]= a;
}

addAnimal(new Crab());
addAnimal(new Turtle());
```
Dynamic and Static type : Casts

```java
Animal animal;
animal = new Crab();
```

- **static type** Animal
- **dynamic type** Crab

- when calling an instance method the dynamic type is used.
- when calling a static method the static type is used.
- you can force the static type (only to super-classes of the dynamic type, or to the dynamic type) using casts:

```java
Crab c = (Crab) animal; // OK
Turtle t = (Turtle) animal; // Runtime ERROR
```

Dynamic dispatching

- When you call an instance method, the method used is the one provided by the dynamic class, this is called **dynamic dispatching**.
- It is the really powerful idea behind polymorphism:
  - You can treat a group of objects the same way
  - When you do an operation on one of the objects, the adequate operation will be chosen depending on the dynamic type of the object.
Multiple inheritance?

- We have added further classes to our animal class hierarchy: Swimming with method `swim()`, Walking with method `walk()`.
- As our turtle can both swim and walk we would like it to inherit from both classes.
- But in Java this is forbidden.
Multiple inheritance: problem

When we call `doIt()` on `C`, do we call `B1` or `B2` implementation?
Multiple answers to this problem (see for example Eiffel’s nice solution),
Java Answer: Interfaces.

Definition
An interface is a behavioural contract that a class decides to honor.

- Concretely, an interface is a collection of method signatures.
- If a class `implements` an interface, it has to provide a body for each of those methods.
- A class can implement multiple interfaces.
- An interface can extend another (single) interface.

Q: Why does it solves the multiple inheritance problem? A: We multiply interface, we do not multiply implementation...
Multiple inheritance with interfaces

interface A
   doIt();

     extends

interface B1
   doIt();

     implements

interface B2
   doIt();

     implements

class C
   C.doIt()

B2 and B1 asked for a method doIt, C provides it, no ambiguity

public interface Swimming {
   void swim();
}
public interface Walking {
   void walk();
}
class Turtle extends Animal
   implements Swimming, Walking{
   void swim() { /* swim implementation */}
   void walk() { /* walk implementation */}
}
To factorize code, creating classes hierarchies is important.

- Each class should hide its implementation to make code robust and maintainable.
- With polymorphism one can design elegant, factorised code.
- When an object implements different behaviours, one should use interfaces.