Object Oriented Programming

PART II

Objects

Beniamin Bogosel

Aurel Vlaicu University, Arad

Object Oriented Programming (OOP): basic ideas

⋆ OOP: sophisticated and **efficient** code

- encapsulation
- inheritance
- polymorphism
- abstraction

Main concept: Objects - instances of classes that have unique attributes and behavior

Objects are **data structures** which contain:

- attributes
- methods

Encapsulation

⋆ limit direct access to certain properties and methods to prevent unwanted access: example: password in a mail account
⋆ allow access only using a well defined interface (for example a procedure to change one's password)
⋆ main purpose of encapsulation is **not Security**: restrict access to sensible parts of code.
⋆ private, protected, public attributes for each attribute or method

- private: only accessible within the object
- protected: accessible within the class hierarchy
- public: accessible from everywhere

Examples:

- driving a car: we give a few inputs, but we don't know what's inside
- lowering/raising electric car windows: only a button, don't know what's inside

Inheritance

⋆ create new classes based on existing ones
⋆ saves effort in writing code: if a more complex class is needed we don't necessarily need to start from scratch; we can start from a given class
⋆ makes code more modular, easier to maintain

Example: database for keeping up track of animals in zoo

- we have a class "Animal" which contains: age, date of arrival, vaccination status, health status, cost to maintain
- if we wish to have an object which models the "Elephants" in the zoo we don't need to start from scratch.
- We define a new object which inherits the "Animal" class and start from there

Polymorphism

$\star$ it allows objects or methods to take multiple forms

$\star$ **method overloading**: have a function (which does the same prescribed thing!) behave differently depending on inputs

Example: Compute the area of a triangle:

- two inputs: assume height h and base b are given – return $hb/2$
- three inputs: assume side a, side b and angle between them θ are given – return $\frac{1}{2}ab\sin\theta$

Why? To make code more adaptable and flexible, easier to use.

Abstraction

⋆ focus on essential features in an applications, hiding unnecessary complicated details from the user
⋆ make code easier to understand.
⋆ reduce code duplication, making it easier to add new features and modify the ones which already exist
⋆ Find the right level of abstraction: too much abstraction can lead to complicated code!
Example:

- you have a class Polygon representing polygons in the plane
- inside the class you have a method "intersect" to find intersection of two polygons
- when writing `P1.intersect(P2)` we don't see (or care ☺) about what happens inside

Nothing is perfect...

$\star$ the OOP principles should be used in a **reasonable way**: do not overdo it!

Examples:

- Do not make everything an object: a point in the plane can be simply represented with a vector!
- Use inheritance reasonably:
  - do not use too many levels of inheritance unless this is needed or natural
  - all properties of the parent class are passed to the children inheriting classes: be careful, encapsulation might be lost!
- Use abstraction reasonably: if not the code will become hard to maintain

Example of a class: Box

```
public class Box {

    // ATTRIBUTES
    private boolean closed;
    private String  name;
    private int     value;

    // CONSTRUCTORS
    public Box(String n, int v) {
        this.name = n;
        this.value = v;
        this.closed = true;
    }
    public Box(int v) {
        this("Standard", v);
    }

    // REQUESTS
    public String name() {
        return this.name;
    }
    public int contents() {
        return this.value;
    }
    public boolean closed() {
        return this.closed;
    }

    // COMMANDS
    public void fill(int v) {
        if (this.closed()) {
            throw new AssertionError();
        }
        this.value = v;
    }
    public void open() {
        this.closed = false;
    }
    public void close() {
        this.closed = true;
    }
}
```

Attribute

`[visibility] [final] <Type> <id>`

Attribute: defines a field for the class instances

- ⋆ visibility: `private | protected | public`
- ⋆ `private`: accessible uniquely inside the text of the class
- ⋆ `public`: accessible everywhere
- ⋆ `final`: define a constant

Why bother with accessibility?

⋆ prevent unwanted access to data

```
class Box {
    private String pass;
    public void clear(String passWord) {
        if (pass.equals(passWord)) {
            if (closed()) {
                open();
            }
            value = 0;
            close();
        } else {
            System.out.println("Ha Ha !
                ");
        }
    }

}
```

```
class Thief {
  public steal(Box b)
      {
    b.clear(b.pass);
  }
}
```

Method

`[visibility] <Type> <id> ([<args>])[<ErrorClause>] <body>`

⋆ Head: visibility+signature+error clause

⋆ signature: Type+Name+(Args) (shows how to call the method)

⋆ Body: instruction block, actions realized by the object

```
public void open() {
    this.closed =
        false;
}
```

Type of value returned by method

⋆ type written before the name of the method

```
public int contents() {
    return value;
}
```

⋆ if there's no value returned use `void`

```
public void close() {
    closed = true;
}
```

Overloading methods

⋆ defined multiple times with same name
⋆ addition operation: works for `int`, `float`, `String`
⋆ overloaded methods should **do the same thing** but for different types of inputs
⋆ there's no point in overloading a method to do two different things! it makes code incoherent and unpredictable!

Arguments for a method

⋆ Each argument is defined with its type

```
public double Area(double height, double base)
```

⋆ if no argument is to be given, use void parantheses `()`

```
public void close()
```

⋆ Arguments are passed

- uniquely by value
- formal parameter = local variable

Arguments passed by value: primitive types

```
void withoutSideEffect(int n) {
    // A1
    n = 2 * n;
    // A2
    System.out.println(n);
}
```

then calling:

```
int x = 5;
withoutSideEffect(x);  // 10
System.out.println(x); // 5
```

Arguments passed by value: object types

```
void sideEffect(Box b1, Box b2) {
    // A1
    b1 = b2;
    b1.open();
    // A2
}
```

then calling:

```
Box bt1 = new Box();
Box bt2 = new Box();
System.out.println(bt1.closed()); // true
System.out.println(bt2.closed()); // true
sideEffect(bt1, bt2);
System.out.println(bt1.closed()); // true
System.out.println(bt2.closed()); // false
```

⋆ the value it passes for objects is the reference address of the object, not the object itself!

Constructor

⋆ Procedure defined in a class

- has the **exactly** same hame as the class
- it initializes an instance of the class

```
Box b = new Box("something",0);
```

Constructors

- it is used only with `new`
- does not `return` an expression (but can contain a `return`)
- is not a method

Constructors...

⋆ can be overloaded:

```
public Box(String s, int v) {...}
public Box(int v) {...}
```

⋆ a constructor can call another from the same class

```
public Box(int v){
    this("Standard",v);
    // code from here
}
```

`this(...);` must be on the first line of such a constructor

Creating an object in Java

`new <IdClass>([args])`

⋆ This creates a new instance of `<IdClass>`

- memory allocated
- a reference is created

⋆ standard initialization for fields

- numeric: 0
- char: `'\u0000'`
- boolean: `false`
- reference: `null`

⋆ the constructor may change some fields (initialization) ⋆ it returns the reference (memory adress)

Example

```
class Box {
    // ATTRIBUTES
    private boolean closed;
    private String  name;
    private int     value;
    // CONSTRUCTORS
    public Box(String n, int v){
        this.name = n;
        this.value = v;
        this.closed = true;
    }
    public Box(int v){
        this("Standard",v);
    }
    ...
}
```

⋆ `new Box(3);`

⋆ create a box with attributes: name: "Standard", value = 3, which is "closed"

⋆ the value of the expression is the reference in memory

Class without explicit constructor

```
class Box {
    // ATTRIBUTES
    private boolean closed;
    private String  name;
    private int     value;
    // CONSTRUCTORS
    public Box(){
    // does nothing
    }
...
```

⋆ Simply use `new Box();`

⋆ Attributes are initialized with default values.

⋆ If you don't write the constructor explicitly, compiler will generate a no-args constructor by default.

Keyword: this

⋆ this = current instance

```
public void fill(int v) {
    if (this.closed()) {
        ...
    }
    this.value = v;
}
```

⋆ this(...) = call of another constructor

```
class X {
    X(String s, int i) { ... }
    X(String s) {
        this(s, 0);
        ...
    }
}
```

Classes are also objects

- At the execution:
  - class represented by a particular object
  - instance of the class `Class`
  - different state and behavior for its instances
- Class attribute
  - external field for the instances
  - accessible for all instances
  - initialization when changing the class
- Class method
  - a certain service for the class object
  - its action is independent of the instances
  - cannot use `this`

Class atributes

⋆ Declaration of attribute using `static`

- static members belong **to the class**
- can be accessed without creating instances of the class
- shared among the common instances

⋆ Initialize class attributes: – in the same time as their declaration

```
class X {
    private static int nbInstances = 0;
    ...
}
```

– or in a static block;

```
class X {
    private static int nbInstances;
    static {
        nbInstances = 0;
    }
    ...
}
```

Defining a class method

⋆ again, with `static`

```
class X {
    private static int nbInstances = 0;
    // Constructor
    public X() {
        nbInstances += 1;
    }
    public static int getNbInstances() {
        return nbInstances;
    }
    ...
}
```

Calling a static caracteristic

⋆ If target is not precised: the target is the encompassing class

```
class X {
    private static int nbInstances = 0;
    public X() {
        nbInstances += 1; // modifies the attribute of X
    }
}
```

⋆ otherwise, put target before `X.getInstancesNb()`

Mutable objects

⋆ A mutable object can modify its state

- values of the fields can be changed
- changing the state does not lead to the creation of a new object

⋆ A mutable class = instances are mutable

Example: `Box`

– methods: `fill, open, close`

Counterexample: `String`

– once a string is defined, it is not modified

Need to pay attention at mutability!

```
class MutPoint {
    private int x, y;
    public MutPoint(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
    public void move(int dx, int dy) { x += dx; y += dy; } /// This class is MUTABLE
    ...
}
```

```
class Circle {
    private MutPoint center;
    private double radius;
    public Circle(MutPoint c, double r) {
        center = c;
        radius = r;
    }
    public MutPoint getCenter() {
        return center;
    }
    public void move(int dx, int dy) {
        center.move(dx, dy);
    }
    ...
}
```

```
MutPoint p = new MutPoint(0, 0);
Circle c = new Circle(p, 5.0);
p.move(3, 3);      // The Circle MOVED even though we did not tell it to move!
```

Fix this: Solution 1

```
class Circle {
    private MutPoint center;
    private double radius;
    public Circle(MutPoint c, double r) {
        center = new MutPoint(c); // a new point is created, different from c
        radius = r;
    }
    public MutPoint getCenter() {
        return new MutPoint(center); // when getting the center we don't get the address of the original one
    }
    public void move(int dx, int dy) {
        center.move(dx, dy);
    }
    ...
}
```

```
class MutPoint {
    public MutPoint(MutPoint p) {  // Replicating constructor; build a double of current point
        this.x = p.getX();
        this.y = p.getY();
    }
    ...
}
```

```
MutPoint p = new MutPoint(0, 0);
Circle c = new Circle(p, 5.0);
p.move(3, 3);    // the circle does not move now...
```

Fix this: Solution 2

```
class NonMutPoint { // non-mutable version
    private int x, y;
    public NonMutPoint(x, y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
    ...
    public NonMuTPoint move(int dx, int dy) { return new NonMutPoint(x+dx,y+dy); }
}
```

```
class Circle {
    private NonMutPoint center;
    private double radius;
    public Circle(NonMutPoint c, double r) {
        center = c;
        radius = r;
    }
    public NonMutPoint getCenter() {
        return center;
    }
    public void move(int dx, int dy) {
        center = center.move(dx, dy);
    }
    ...
}
```

```
NonMutPoint p = new NonMutPoint(0, 0);
Circle c = new Circle(p, 5.0);
p = p.move(3, 3);
```

Testing if two objects are identical

⋆ Identical objects = same reference/address in memory

```
Box a = new Box("Treasure", 10);
Box b = a;
```

⋆ After the execution, the references given to `a` and `b` are identical
⋆ the boolean expression `a==b` has the value `true`

Testing equivalence of two objects

⋆ **Equivalent objects**: same class, identical contents, indistinguishable

```
String name = "Treasure";
Box c = new Box(String, 10);
Box d = new Box(String, 10);
```

⋆ objects referenced in `c` and `d` are equivalent, but `c!=d`

⋆ the boolean expression `c.equals(d)` has the value `true`

The boolean method `equals(Object)`

⋆ is present in every class

```
public boolean equals(Object o) {
    return this == o;
}
```

⋆ Every reference type is compatible with `Object`
⋆ An entity of type `Object` accepts any object
⋆ adding a method `equals` inside a class allows to define an equivalent relation less fine than the identity