Ca' Foscari University of Venice

Encapsulation and information hiding

Object oriented programming, module 1

Pietro Ferrara
pietro.ferrara@unive.it

Ca' Foscari University of Venice

Abstraction and interfaces

- A class defines a contract specifying the interface of the objects
  - Method signature represents the structure
  - Method semantics (meaning) needs to be documented externally
- This allows to abstract away the internal implementation

```
class Car {
  //Add the given amount to the fuel tank
  void refuel(double amount) {…}
  //Increment the speed
  void accelerate(double a) {…}
  //Stop the car
  void fullBreak() {…}
}
```

From Lecture 3, side 14

Ca' Foscari University of Venice

Encapsulation

- Fundamental OO concept
- Encapsulation: data is bundled with the methods that operate on it
- Needed to hide internal information
  - A program with "good" encapsulation maximize information hiding
  - The class interface is clearer
  - … and more reusable!
- We might ignore the fuelType of the car
  - Just refuel with the tank

```
class Car {
 double speed;
 double fuel;
 FuelType fuelType;
void refuel(FuelTank tank) {
  if(! tank.type.equals(fuelType))
   throw new Exception();
  else fuel += tank.amount;
 }
 void accelerate(double a) {
  speed += a;
  fuel -= a*FUEL_CONS;
 }
 void fullBreak() {
  speed = 0.0;
 }}
```

Ca' Foscari University of Venice

Goal of encapsulation

- Goal of encapsulation:
  - Guarantee the consistency of the computational state
- Encapsulation is not limited to classes
  - A capsule might comprise several classes, a package, …
- Maybe accelerating too much?
  - fuel might be negative!
- But anybody could modify field fuel!

```
class Car {
 double fuel;
 double speed;
 void accelerate(double a) {
  speed += a;
  fuel -= a*FUEL_CONS;
 }
}
```

Ca' Foscari University of Venice

Modifiers

- Access modifiers: fields and methods
  - Only public for classes
  - Will be covered during the course
- Concurrency modifiers: fields and methods
  - Not covered
- Static: fields, methods
  - Will be covered today
- Final: fields, methods, classes
  - Will be covered during the course (fields today)

ods, classes

ed during the course

From Lecture 3, side 20

public
no modifier (default)
protected
private

Access

synchronized
volatile

Concurrency

static
final
abstract

Others

Ca' Foscari University of Venice

Information hiding

- Minimize the exposed interfaces
  - Minimizing dependencies
- Information hiding:
  - Limit the access to object states
- A client must have
  - Access all information to use the module
  - No access to any other data of the object
- Access modifiers hide information
  - From other classes or packages

```
class Car {
 private double fuel;
 private double speed;
 void ac elerate(double a) {
  if(a<0      turn;
  doub          spt = a*FUEL_CONS;

  else {
   speed += fuel* FUEL_CONS;
   fuel = 0.0;
 }
}
```

Only code in class Car can access these fields!

Ca' Foscari University of Venice

Access modifiers

	Same class	Same package	Subclasses	Everywhere
public	👍	👍	👍	👍
protected	👍	👍	👍	👎
<default>	👍	👍	👎	👎
private	👍	👎	👎	👎

- Java 9+ introduce a further layer
  - but no new access modifiers
- Not covered in this course

Covered in the next lectures!

Ca' Foscari University of Venice

Access modifiers

	Same class	Same package	Everywhere
public	👍	👍	👍
[redacted]	[redacted]	[redacted]	[redacted]
<default>	👍	👍	👎
private	👍	👎	👎

Ca' Foscari University of Venice

Different views

- Access modifiers define different views of the same classes
- public: everything that represent the external interface
  - The minimal functionalities and data provided by the class
  - It will (hopefully) **never** change in the future
- protected/default: what needs to be accessed by the same sw unit
  - Might be changed requiring some local code rewriting
- private: what should be completely hidden from outside
  - Inner implementation details that might be modified/replaced

Ad-hoc examples

```
package a;
public class A {
  private int a_private;
  int a_default;
  public int a_public;
}
```

```
package b;
import a.*;
class B {
  private int b_private;
  void foo(A a, B b, B1 b1) {
✔ a.a_public;
□ a.a_default;
□ a.a_private;
✔ this.b_private;
✔ b.b_private;
✔ b1.b1_public;
✔ b1.b1_default;
□ b1. b1_private;
}
```

```
package b;
class B1 {
  private int b1_private;
  int b1_default;
  public int b1_public;
}
```

Ca' Foscari University of Venice

Examples of information hiding

```
package it.unive.dais.po1.fuel;
class FuelTank {
  FuelType type;
  double amount;
  FuelTank(…) {…}
}
```

```
package it.unive.dais.po1.fuel;
class FuelType {
  String name;
  double costPerLiter;
  double fuelConsumption;
  FuelType(…) {…}
}
```

```
package it.unive.dais.po1.car;
import it.unive.dais.po1.fuel.*;
class Car {
  double speed;
  FuelType fuelType;
  double fuel;
  void refuel(FuelTank tank) {…}
  void accelerate(double a) {…}
  void fullBreak() {…}
}
```

These are only internal details
None should modify
speed/fuel[type]

This is something we want to provide externally!

Ca' Foscari University of Venice

Examples of information hiding

```
package it.unive.dais.po1.fuel;
class FuelTank {
  FuelType type;
  double amount;
  FuelTank(…) {…}
}
```

```
package it.unive.dais.po1.fuel;
class FuelType {
  String name;
  double costPerLiter;
  double fuelConsumption;
  FuelType(…) {…}
```

```
package it.unive.dais.po1.car;
import it.unive.dais.po1.fuel.*;
class Car {
  private double speed;
  private FuelType fuelType;
  private double fuel;
  public void refuel(FuelTank
     tank) {…}
  public void accelerate(double a)
     {…}
```

We need to build up fuel types externally

These are only internal details
None should modify name, costPerLiter, fuelConsumption

Ca' Foscari University of Venice

Examples of information hiding

```
package it.unive.dais.po1.fuel;
class FuelTank {
 FuelType type;
 double amount;
 FuelTank(…) {…}
}
```

```
package it.unive.dais.po1.fuel;
class FuelType {
 private String name;
 private double costPerLiter;
 private double
    fuelConsumption;
 public FuelType(…) {…}
}
```

```
package it.unive.dais.po1.car;
import it.unive.dais.po1.fuel.*;
class Car {
 private double speed;
 private FuelType fuelType;
 private double fuel;
 public void refuel(FuelTank
    tank) {…}
 public void accelerate(double a)
    {…}
 public void fullBreak() {…}
}
```

We need to know and modify the amount of fuel we have in a tank!

Examples of information hiding

```
package it.unive.dais.po1.fuel;
class FuelTank {
  private FuelType type;
  public double amount;
  public FuelTank(…) {…}
}
```

```
package it.unive.dais.po1.fuel;
class FuelType {
  private String name;
  private double costPerLiter;
  private double
     fuelConsumption;
  public FuelType(…) {…}
}
```

```
package it.unive.dais.po1.car;
import it.unive.dais.po1.fuel.*;
class Car {
  private double speed;
  private FuelType fuelType;
  private double fuel;
  public void refuel(FuelTank
     tank) {…}
  public void accelerate(double a)
     {…}
  public void fullBreak() {…}
}
```

- Can we see the car speed?
- Can we have a negative amount of fuel?
- And how can I check the fuel type when refueling my car?

Ca' Foscari University of Venice

getters

- Provide read only access to a field
  - final if it's constant through object lifecycle
  - private + getter otherwise
- Usually, "return <field>"
- But not necessarily a field
  - Compute a value from the state of the object
- It does not change the state of the object
- Less restrictive access than the field

```
package it.unive.dais.po1.car;
import it.unive.dais.po1.fuel.*;
class Car {
 private double speed;
 private FuelType fuelType;
 private double fuel;
 public double getSpeed() {
  return speed;
 }
 public double getValueOfFuel() {
  return fuel*fuelType.getCost();
 }
}
```

Ca' Foscari University of Venice

setters

- Provide write only access to the field
- Usually, this.<field>=<value>
- Does not return a value
- Might check if the value satisfies some constraints
- Mutate the state of the object
- Less restrictive access than the field

```
package it.unive.dais.po1.car;
import it.unive.dais.po1.fuel.*;
class Car {
  private double speed;
  private FuelType fuelType;
  private double fuel;
  public void setSpeed(double s) {
      this.speed = s;
  }
  public void setSpeed(double s) {
     if(s<0)
      this.speed=0
     else this.speed = s;
  }
}
```

Ca' Foscari University of Venice

getters and setters

- Advantage: all the accesses to object states are controlled by a method
  - Easier to check and constrain object invariants
- Disadvantage: you must call a method to modify a field
  - Less efficient

```
Car[] allMyCars = …;
int totalFuel = 0;

for(int i=0; i < allMyCars.length; i++)
  totalFuel += allMyCars[i].getFuel();
//Create local environment for the
     call, load the parameters, get
     this, get the field, return it,
     destroy the local environment of
     the call

for(int i=0; i < allMyCars.length; i++)
  totalFuel += allMyCars[i].fuel;
//get the field value
```

Ca' Foscari University of Venice

Maximizing information hiding

- Tradeoff between functionality and information hiding
- Why not: everything is public, and… who cares???
- As soon as something is part of the interface, it will be used
  - You cannot blame the others because they should have not used something that was available!
- And, for sure, it will be used in the wrong way!
- *A client must have*
  - *Access all information to use the module*
  - *No access to any other data of the object*

Typical workflow

1. Decide what should be the public interface of your sw unit
2. All the rest (methods and fields) must be private
3. While developing the sw unit, you will need to relax the access
    1. Some data and functionalities are needed by other parts (classes of the same package or of other packages) of the sw unit
    2. You might reconsider some public interface of your sw unit

Ca' Foscari University of Venice

Textbook

- Lecture notes: Chapter 4
- Encapsulation and/vs information hiding:
  - https://www.infoworld.com/article/2075271/encapsulation-is-not-information-hiding.html
- Arnold&others:
  - Access modifiers: Section 2.3 & 2.6.6
- Getter and setters: https://dzone.com/articles/java-getter-and-setter-basics-common-mistakes-and