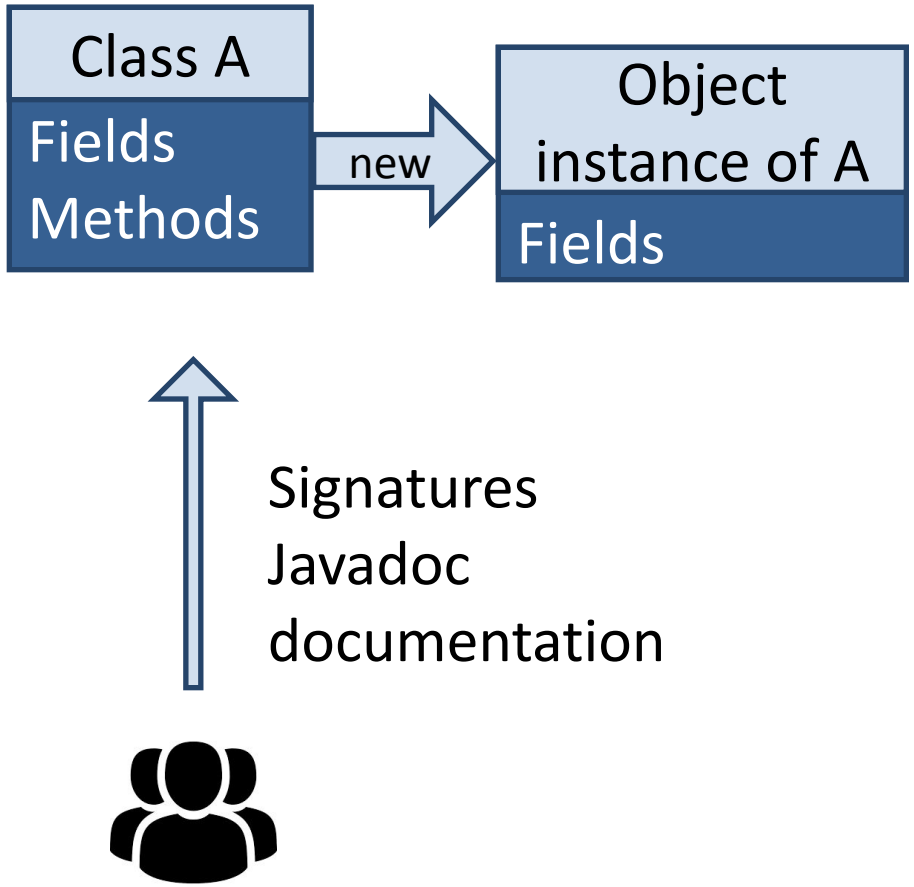
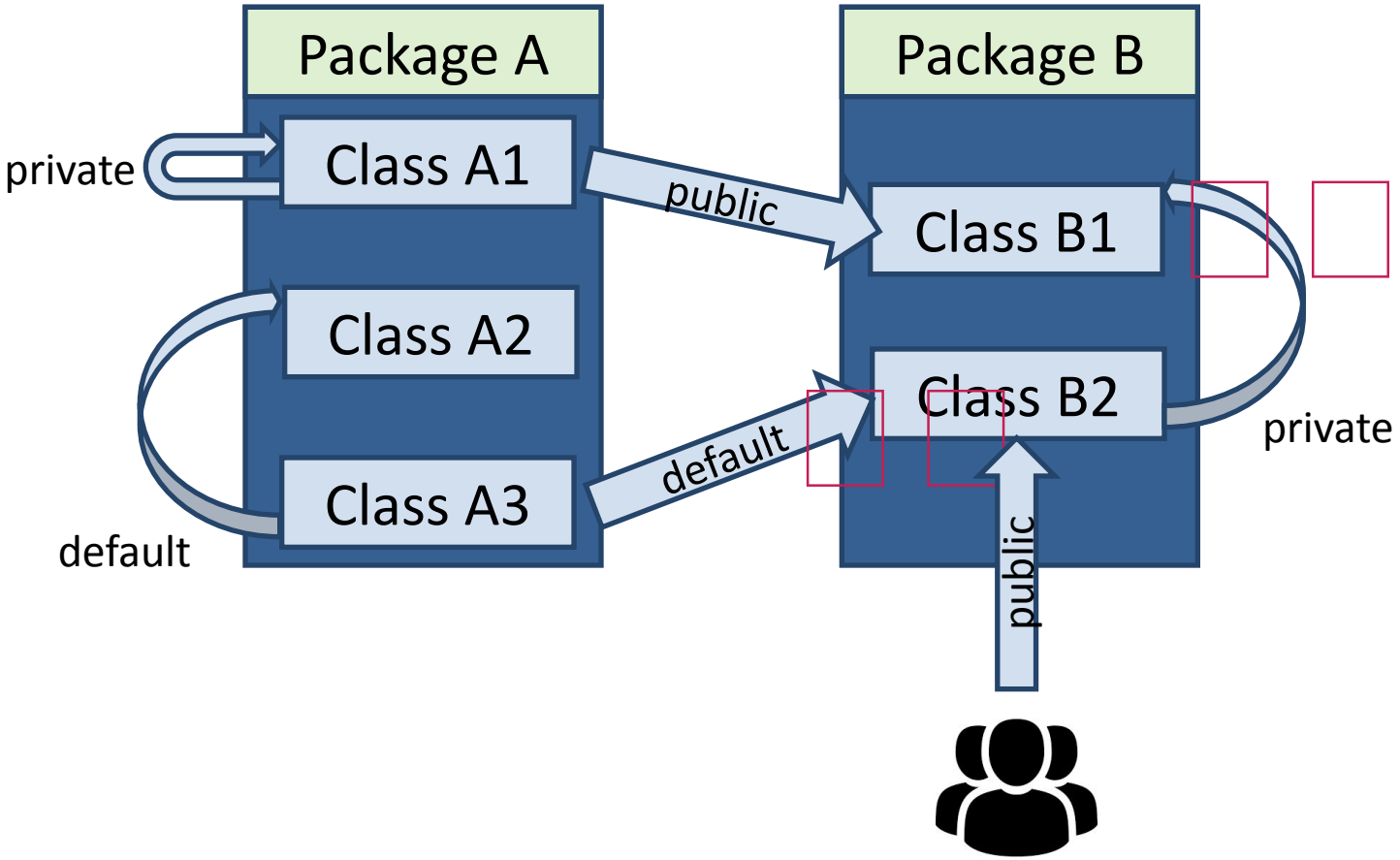
Ca' Foscari University of Venice

Summary





Ca' Foscari
University
of Venice

Classes as contracts

Object oriented programming, module 1

Pietro Ferrara

pietro.ferrara@unive.it

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() {…}
}
```

Ca' Foscari University of Venice

Contracts

A contract is a legally binding agreement that defines and governs the rights and duties between or among its parties. A contract is legally enforceable when it meets the requirements of applicable law. A contract typically involves the exchange of goods, services, money, or a promise of any of those. In the event of a breach of contract, the injured party may seek judicial remedies such as damages or cancellation.

- Rights -> what you need to provide me
- Duties -> what I promise I will provide you
- Programs might be thought as contracts
  - You need to provide me a correct input
  - I promise I will provide you a correct output

Source code as a contract?

- Methods' signature: functionalities that are offered by a class
  - Parameters are what is needed
  - Return type is what is promised
  - Names matter (e.g., accelerate)
- We can see fields in the same way
- Very partial and ambiguous contract
  - Names matter but they are concise
  - A return or parameter type is not so expressive
  - And we have a lot of inner details we want to hide…

```
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;
 }}
```

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 speed;
 private double fuel;
 private 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;
 }}
```

Comments

- We all know that comments are important, isn't it?
  - And we all properly comment our code, isn't it?
- However, we need to distinguish among 2 types of comments
  - Source code comments explain what a part of the code computes
    - single line (starting with //)
    - multi line (starting with /* and ending with */)
  - Documentation (javadoc) comments explain the API of the library
    - Start with /** and end with */
- Javadoc command then generates HTML documentation pages
  - And they are used by IDEs for showing documentation

What contracts can we already write?

- Methods and fields signatures specify the syntax of OO components
- Information hiding is needed to hide everything that is an internal detail
  - How I compute the results, not what!
- Comments specify the semantics of OO components
  - Written in almost informal text
- Modular reasoning
  - Signatures and comments
  - Only public components

```
class Car {
 private double speed;
 private double fuel;
 private FuelType fuelType;
 /**
    * Accelerate the vehicle by the given
     amount of km/h.
    * If the increase is negative, it does
     not accelerate
    * @param a the increase of speed
*/
 public void accelerate(double a) {
  if(a>=0)
    this.speed += a;
 }
}
```

Ca' Foscari University of Venice

Preconditions

- Preconditions ~~ what the client needs to guarantee
  - Before calling the method!
- Method signature is part of the precondition
  - Number and types of parameters
  - Name of the parameters
- [Javadoc] comments is part of the precondition
  - @param tells something more than the parameter
  - The text contains some more information
- Preconditions might restrict the allowed values
  - E.g., "the increase must be greater than zero"

```
class Car {
/**
    * Accelerate the vehicle by the given
     amount of km/h.
    * If the increase is negative, it does
     not accelerate
    * @param a the increase of speed
*/
 public void accelerate(double a) {
  if(a>=0)
    this.speed += a;
 }
}
```

Ca' Foscari University of Venice

Postconditions

- Postconditions ~~ what the code guarantees
  - After calling the method!
- Method signature is part of the postcondition
  - Type of the returned value
- [Javadoc] comments is part of the postcondition
  - @return tells something more than the parameter
  - The text contains some more information
- Postcondition might restrict the resulting values
  - E.g., "the speed is the initial speed + the amount"
    - Be careful, this should be more abstract than the implementation!

```
class Car {
/**
   * Accelerate the vehicle by the given amount of km/h.
   * If the increase is negative, it does not accelerate
   * @param a the increase of speed
*/
 public void accelerate(double a) {
  if(a>=0)
    this.speed += a;
 }
}
```

Ca' Foscari University of Venice

Object invariants

- Object invariants ~~ what the code guarantees and what we need to guarantee
  - After and before calling the method, respectively!
- Specified on the class, not a single method
  - Invariants apply to all methods
  - They can talk only about fields' values
    - No parameters or return values
- They are both a pre and post conditions of all the methods
  - Smth we must guarantee when calling a method
  - Smth the method guarantees after its execution

```
class Car {
 /**
  * The speed of the vehicle
  * This is always greater then or equal to 0
  */
  private double speed;
/**
  * The amount of fuel in the vehicle
  * This is always greater then or equal to 0
  */
  private double fuel;
}
```

Formal specification of classes

- All these specifications can be more or less "formal"
  - Text is completely informal
    - Thus, ambiguous but expressive
    - Up to us what we write there
    - Up to the developer who wrote the code
  - Static types are quite formal
    - Thus, unambiguous but inexpressive
    - We cannot specify so much
- Theoretically, we could fully specify classes through mathematical formulae

```
class Car {
 //Invariant: speed >= 0
  private double speed;

//Precondition: a >= 0
//Postcondition: this.speed = prev(this.speed)+a
//Postcondition: a>=0    =>
          this.speed = prev(this.speed)+a
//Postcondition: a<0 =>
          this.speed = prev(this.speed)
public void accelerate(double a) {
   if(a>=0)
     this.speed += a;
  }
}
```

Ca' Foscari University of Venice

Formal specification of classes

- All these specifications can be more or less "formal"
  - Text is completely informal
    - Thus, ambiguous but expressive
    - Up to us what we write there
    - Up to the developer who wrote the code
  - Static types are quite formal
    - Thus, unambiguous but inexpressive
    - We cannot specify so much
- Theoretically, we could fully specify classes through mathematical formulae

```
class Car {
void accelerate(double a) {
      double fuelConsumed = a*fuelType.litresPerKmH;
      if(fuelConsumed < fuel) {
         speed += amount; fuel -= fuelConsumed;}
     else {
         double increaseSpeed = fuel /
     fuelType.litresPerKmH;
         speed += increaseSpeed; fuel = 0;}
   }
}
//Postcondition: a*fuelType.litresPerKmH < fuel   =>
       speed = prev(speed) + amount AND
       fuel = prev(fuel) - a*fuelType.litresPerKmH
//Postcondition: a*fuelType.litresPerKmH >= fuel   =>
          … enjoy!!! ☺️
```

Ca' Foscari University of Venice

The Java Modelling Language

- https://www.cs.ucf.edu/~leavens/JML/index.shtml
- Very relevant research project
- Not so much impact in industry
- Complex specification
  - Requires
  - Ensures
  - Loop invariant
    - To verify the program

```
public class MaxByElimination {

  //@ requires a != null && a.length > 0;
  //@ ensures 0 <= \result < a.length;
  //@ ensures (\forall int i; 0 <= i < a.length; a[i] <= a[\re
sult]);
  public static int max(int[] a) {
    int x = 0;
    int y = a.length-1;

    //@ loop_invariant 0 <= x <= y < a.length;
    // So far either a[y] is the largest or a[x] is the larges
t of everything beyond x and beyond y (not including a[x] and
a[y])
    /*@ loop_invariant ((\forall int i; 0<=i && i<x; a[i] <= a
[y]) && (\forall int i; y < i && i < a.length; a[i] <= a[y]))
                        ||  ((\forall int i; 0<=i && i<x; a[i]
<= a[x]) && (\forall int i; y < i && i < a.length; a[i] <= a
[x]));
     */
    //@ decreases y-x;
    while (x != y) {
      if (a[x] <= a[y]) x++;
      else y--;
    }
    return x;
  }
}
```

Ca' Foscari University of Venice

Specifying class contracts

- The programming language obliges to specify signatures
  - Needed to interface different program components
- The programming language provides structed comments
  - They are not mandatory
  - They are written in plain language
    - Extremely expressive, but it needs to be “manually” processed
- There are approaches to specify more formally contracts
  - The risk is to get a mathematical copy of the code
    - Yep, it can be automatically inferred, but this is a research theme
  - They never became popular for many [good|bad] reasons

Ca' Foscari University of Venice

A hierarchical view of classes

```
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() {…}
}
```

```
class Bicycle {
  //Increment the speed
  void accelerate(double a) {…}
  //Stop the car
  void fullBreak() {…}
}
```

```
class Truck {
  //Add the given amount to the fuel tank
  void refuel(double amount) {…}
  //Increment the speed
  void accelerate(double a) {…}
  //Stop the car
  void fullBreak() {…}
  //Load some stuff
  void chargeLoad(double l) {..}
  //Unload all the stuff
   double unload() {..}
}
```

Looking to the contracts:

Truck => Car
Car => Bicycle

If I need only to accelerate or break, it does not matter if I have bicycles, trucks, or cars

Ca' Foscari University of Venice

The substitution principle (spoiler!)

An object o1 instance of class C1 can be substituted by an object o2 of class C2 if class C2 provides the same or a wider interface (fields and methods) of class C1

- All the program points accessing a member of C1 can still access the same members with an instance of C2!
- To run a race, I need to accelerate
  - Cars, trucks and bicycles all provide the interface to accelerate!

```
class Vehicle {…}
class Car {…}
class Truck {…}
class Bicycle {…}

int race(Vehicle v1, Vehicle v2, double length) {…}

Car c1 = new Car(), c2 = new Car();
Truck t = new Truck();
Bicycle b = new Bicycle();

race(c1, c2, 100);
race(c1, t, 100);
race(t, b, 100);
race(c1, b, 100);
```

Textbook

- Lecture notes: Chapter 6
- Arnold et al.:
  - 12.9.1: pre and postconditions (through assert statements)
- Budd 10.6 (pre and postconditions)
- Book about design by contract (only if you're really interested to deepen these topics much more than needed for the course): Bertrand Meyer: "Object-Oriented Software Construction" 2nd edition (available in the lib @ BAS)