



Objects and Classes

Lecture 10 of TDA 540 (Objektorienterad Programmering)

Carlo A. Furia Alex Gerdes

Chalmers University of Technology – Gothenburg University

Fall 2017

All labs have been published

Descriptions of the **four remaining lab assignments** (“Laboration 5–8”) are available on the course website:

<http://www.cse.chalmers.se/edu/year/2017/course/TDA540/>

- Do not work on them all at once: **plan** your time wisely
- Some **Java topics** used in the labs will be covered during the **upcoming classes**
- The TAs will start grading submissions only **after** each lab's **deadline**

Object-oriented programming: a minimal history

Mid 1960s Ole-Johan Dahl and Kristen Nygaard develop **SIMULA 67**, the first object-oriented programming language



1970s Alan Kay, Adele Goldberg, and others develop **Smalltalk**, a popular object-oriented language, and introduce the term “**object-oriented programming**”



Mid 1980s Bertrand Meyer develops **Eiffel**, which popularized object technology for the whole software development lifecycle



Mid 1980s Bjarne Stroustrup's **C++** adds object-orientation to C, making it a widely used programming paradigm

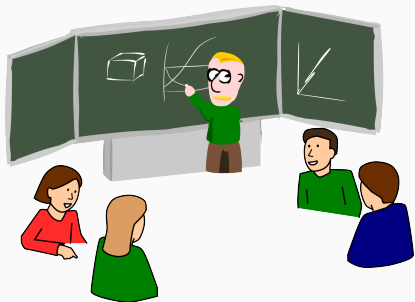


Today many programming languages also support some form of **object-oriented features**



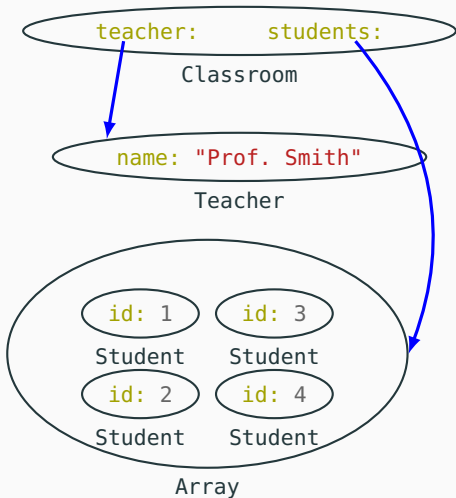
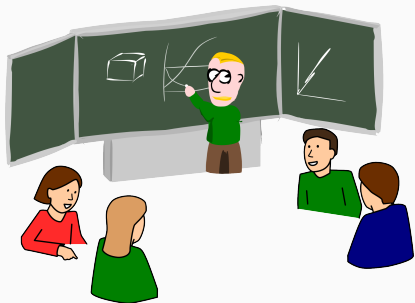
Objects and objects

The basic principles of object-oriented programming are inspired by the idea of modeling **real objects** as **data structures** in a program running on a computer.



Objects and objects

The basic principles of object-oriented programming are inspired by the idea of modeling **real objects** as **data structures** in a program running on a computer.



Objects-oriented programming languages and objects

Object-oriented programming provides expressive features to **abstract**, **modularize**, and facilitate **reuse** of complex programs.

In practice:

- abstraction of complex data types
- inheritance
- polymorphism
- dynamic binding/dispatching

Classes

An object-oriented program is an organized collection of **classes**.

A **class** is a **module** of code that defines data (the **state**) and **operations** on those data (abstract data type).

- a class defines a **type**
- “class” is a **static** notion: it refers to program text

```
class BankAccount // user-defined class
{
    Integer balance; // data: how much money in the account

    void deposit(Integer amount) { // operation:
        // add 'amount' to 'balance'
        balance = balance + amount;
    }
}
```

Objects

Objects are instances of classes.

- an object stores values of certain type (object state = value)
- “object” is a dynamic notion: objects exist when a program executes

CLASS	OBJECT
model/mold	instance
static	dynamic
type	value of a type
Integer	42
BankAccount	balance: 42

Objects and object references

Since objects are dynamic entities while classes are static entities, how can we refer to objects in the program text? Variables of **reference** types (or simply **references**) provide a means to do so.

```
class BankAccount
```

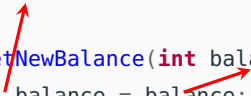
```
{  
  
    Integer balance; // 'balance' is the name of a reference  
                    // attached to an object of class Integer  
                    // whose state represents an integer value  
  
    void deposit(Integer amount) {  
        balance = balance + amount;  
        // change the state of the object of class Integer  
        // attached to reference 'balance'  
    }  
}
```

The object reference `this`

Every class implicitly has a special reference `this`, which refers to the **current object** of the enclosing class.

```
class BankAccount
{
    Integer balance;

    void setNewBalance(int balance) {
        this.balance = balance;
    }
}
```

A diagram with two red arrows. One arrow starts from the word 'this' in the line 'this.balance = balance;' and points upwards and to the left towards the word 'balance' in the line 'Integer balance;'. The second arrow starts from the parameter 'balance' in the line 'void setNewBalance(int balance) {' and points upwards and to the left towards the same 'balance' field.

The life of an object

What we can do with an object obj:

initialize: before first using obj, we have to **create** it

read state: in an expression that refers to some components of obj's state (directly or through a method call)

modify state: by calling a method on obj

dispose: implicit in Java when an object becomes unreachable

```
BankAccount account;  
account = new BankAccount(); // create a new object  
if (account.balance == 0) { // expression referring to  
    // 'balance' in 'account'  
    System.out.println("No money available");  
}  
account.deposit(1000); // modify 'balance' in 'account'  
System.out.println("Now you have " + account.balance + " kr.");
```

What is in a class?

The components of a class are called its **members**:

attributes: represent data components

methods: represent operations

constructors: special methods for initializing objects

```
class BankAccount {  
    // attribute  
    Integer balance;  
    // method  
    void deposit(Integer amount) { balance = balance + amount; }  
    // constructor  
    BankAccount() { balance = 0; }  
}
```

Visibility of members

The **visibility** of a class member defines where in a program we can **refer to** that member (read the value of an attribute, or call a method).

Visibility modifiers are keywords to define the visibility of a member x :

- **private**: x is only visible in the enclosing class
- **default** (no keyword): x is visible within the package where the enclosing class is defined
- **protected**: x is visible within the same package and in every subclass of the enclosing class
- **public**: x is visible everywhere in the program

Visibility modifiers **cannot** be applied to **constructors**.

Visibility of members: examples

```
package p;
```

```
class A {  
    private int a;  
    int b;  
    protected void x()  
    { a = 3; }  
    public void y()  
    { b = 4; }  
    private void z()  
    { a = b; }  
}
```

```
package p;
```

```
class Z {  
    public static  
    void main(String[] args) {  
        A o = new A();  
        o.a = 1; // ERROR!  
        o.b = 2; // OK  
        o.x(); // OK  
        o.y(); // OK  
        o.z(); // ERROR!  
    }  
}
```

Visibility of classes

The **visibility** of a class defines where in a program we can **refer to** and **create** objects of that class.

Visibility modifiers are keywords to define the visibility of a (top level) class `c`:

- **default** (no keyword): `c` is visible within the package where it is defined
- **public**: `c` is visible everywhere in the program

A **constructor** has the same visibility of its enclosing class.

A class is **top level** when it is not defined inside another class. The opposite is a **nested class**, which can have any visibility level (like any other class member). This course mainly deals with top-level classes.

Attributes

Attributes are also called **instance variables** or **fields**.

- Each attribute represents part of the **state** of the object of the class where the attribute is declared
- Attributes are **declared** within a class's curly braces, outside any method's body
- They should not be confused with local variables (declared inside method bodies)
- Attributes are **visible** (that is accessible) at least within the methods of the class where they are declared
- When creating an object, attributes are implicitly **initialized** to default values according to their types
- Constructors can introduce different initialization values

Attributes: initialization

```
class BankAccount
```

```
{
```

```
    // money in the account
```

```
    Integer balance = 100;
```

```
    // name of the owner
```

```
    String owner;
```

```
    // year when account
```

```
    // was opened
```

```
    int openYear;
```

```
}
```

```
BankAccount account;
```

```
account = new BankAccount();
```

```
// initialization of attributes:
```

```
// account.balance == 100
```

```
// account.owner == null
```

```
// account.openYear == 0
```

Methods

Methods are also called **instance methods** or **member functions**.

- Each method represents an **operation** that can be executed on objects of the class where the method is declared
- Two kinds of operations (one method can do both):
 - procedures (commands)** modify the object state
 - functions (queries)** return information about the object state
- Methods are **declared** within a class's curly braces:

```
t0 methodName(t1 a1, t2 a2, ...) { /* body/implementation */ }
```

- may have arguments (use () after method name if no arguments)
 - must have a return type t0 (use **void** if method returns no value)
- Methods are **visible** (that is callable) at least within the methods of the class where they are declared

Method declaration examples

```
class BankAccount {  
    private int balance;  
  
    // procedure: modify state, no returned value  
    void deposit(int amount)  
    { balance = balance + amount; }  
  
    // function: return value, do not modify state  
    int getBalance()  
    { return balance; }  
  
    // procedure and function (function with side effects)  
    int withdrawAndBalance(int amount)  
    { balance = balance - amount;  
      return balance; }  
}
```

Methods good practices: getters and setters

1. Keep attributes **private/protected**, so that they cannot be modified **directly** by objects of other classes
2. Provide **getter** methods to **indirectly** get the value of attributes
3. Provide **setter** methods to **indirectly** set the value of attributes

```
class BankAccount {  
  
    private int balance;  
  
    int getBalance()  
    { return balance; }  
  
    void setBalance(int amount)  
    { balance = amount; }  
}
```

Advantages of using getters and setters

- **Decoupling**: you can change the internal representation of an object's state, without affecting other classes
- **Encapsulation**: the class retains control over how and when its state is changed

First version:

```
class BankAccount {  
  
    private int balance;  
  
    int getBalance()  
    { return balance; }  
  
    void setBalance(int amount)  
    { balance = amount; }  
}
```

Second version:

```
class BankAccount {  
    private int checking;  
    private int savings;  
  
    int getBalance()  
    { return checking + savings; }  
  
    void withdraw(int amount)  
    { if (checking > amount)  
        checking = checking - amount; }  
}
```

Information hiding

Information hiding is the practice of restricting the visibility of attributes, decoupling their private and public representations, and controlling how to modify the state and who can modify it.

Information hiding is one way in which object-oriented programs provide **abstraction**.

The publicly visible attributes and methods of a class are the class's **public interface** or **API** (Application Programming Interface).

Note that Java's keyword **interface** identifies a specific mechanism to define public interfaces in Java, but the concept of public interface is more general (in fact we have not discussed Java's **interface** yet).

Clients

A piece of code that manipulates objects of a class `C` is a **client** of `C`.

- Information hiding is a way of **decoupling** the **client's view** and the internal representation of a class
- As long as a class does not change its **public interface**, clients do not have to worry about changes in the class's internal representation

Public interface:

```
class BankAccount {  
  
    int getBalance()  
    { /* ... */ }  
  
    void withdraw(int amount)  
    { /* ... */ }  
}
```

Client code:

```
BankAccount account;  
account = new BankAccount();  
  
// we don't have to worry how getBalance  
// computes the balance from  
// the internal representation  
if (account.getBalance() > 100)  
    System.out.println("Can buy lunch!");
```

Constructors

Constructors are **special** methods:

- They must have the **same name** as the class where they are defined
- They have **no return type** (not even **void**)
- They are responsible for initializing attributes of the enclosing class (in a way different from the default values)
- If a class `C` declares no constructors, `new C()` gives an object of class `C` with all attributes initialized to default values (**default constructor**)
- A class may have **multiple constructors** with different arguments
- A constructor is invoked implicitly when evaluating a **new** expression

Constructors: examples

Notice the usage of **this** to refer to attributes with the same name as a method's argument.

```
class Account {
    int balance;
    String owner;

    Account()
    { balance = 100; }

    Account(int balance)
    { this.balance = balance; }

    Account(String owner)
    { this.owner = owner; }
}

// client code
Account a1 = new Account();
Account a2 = new Account(3000);
Account a3 = new Account("John Doe");
```

Overloading

Overloading means declaring several methods in the same class that have the **same name** but **different signatures**: their arguments differ in number, type, or both.

Calls to overloaded methods pick the right method based on the number and type of **actual arguments**.

```
class BankAccount {  
    int balance;  
  
    void deposit(int amount) { balance = balance + amount; }  
  
    void deposit() { deposit(100); }  
  
    void deposit(double amount) { deposit((int) amount); }  
  
    void deposit(String amount) { deposit(Double.parseDouble(amount)); }  
}
```

Constants

The keyword **final** specifies that an attribute, argument, or local variable is **constant**:

- they cannot be changed after they are initialized
- **final** attributes **must** be **explicitly** initialized by every constructor (or directly where the attribute is declared)

Style tip: constant attribute names normally are in all caps.

```
class Dice
```

```
{
```

```
    final int SIDES;
```

```
    Dice() { SIDES = 6; }
```

```
    Dice(int sides) { SIDES = sides; }
```

```
    void setSides(int sides) { SIDES = sides; } // Error!
```

```
}
```

Static members

Attributes and methods declared using the keyword **static** relate to the whole **class** where they are declared, as opposed to each instance (object) independently from the others. Thus, **static members** behave very differently from the **instance** members we have seen in this class so far.

```
class Bank {  
  
    static double interest = 0.02;  
  
    static double interest(double amount)  
    { return amount * (1 + interest); }  
  
    if (new Date() == NEW_YEAR)  
    {  
        balance =  
            Bank.interest(balance);  
    }  
}
```

Static members

Attributes and methods declared using the keyword **static** relate to the whole **class** where they are declared, as opposed to each instance (object) independently from the others. Thus, **static members** behave very differently from the **instance** members we have seen in this class so far.

- a **static attribute** is a state component shared by every object of the class where it is declared
- a **static method** can only reference static members, as well as its local variables and arguments
- **static** members are accessed using their class name instead of an object reference; thus, they are accessed without creating objects of their enclosing class

The main method

The method `main` with signature

```
public static void main(String[] args)
```

is a static method that **runs first** whenever we run a Java program.

From `main` (which does not need any objects to run because it is a **static** method) all objects in the program are created as the program continues executing.

When to use static members?

In **object-oriented** programming:

- **instance** members are the **norm**
- static members are used only for **special cases**

Instance members capture the **state** of and **operations** on **objects**:

OPERATION	INSTANCE
create object:	<code>Account a = new Account();</code>
modify object state:	<code>a.deposit(100);</code>
read current object state:	<code>if (a.balance() > 100) ...</code>

Static members capture **global** operations and state that are available independent of the created objects:

ITEM	STATIC
constant:	<code>double angle = Math.PI/4.0;</code>
math operation:	<code>double cathetus = hypotenuse * Math.cos(angle);</code>
global state:	<code>double interest = BankAccount.interest;</code>

Static or instance?

Rule of thumb to **choose** whether member m should be static:

Does it make sense to call (method) or access (attribute)
 m **independent of** specific **objects** of its class?

1. If the answer is **yes**, then you probably need a static member;
2. If the answer is **no**, then you should go with an instance member.

In **most** cases, the answer should be **no**!

Static or instance: examples

```
class BankAccount {  
  
    private int balance;  
  
    public  
    void withdraw(int amount) {  
        balance = balance - amount;  
    }  
  
    public  
    void deposit(int amount) {  
        balance = balance + amount;  
    }  
}
```

Both `withdraw` and `deposit` modify the state (attribute `balance`) of the current **object**: they must be instance methods.

Static or instance: examples

```
class BankAccount {  
  
    private static final double interest = 0.02;  
  
    public static double interest() {  
        return interest;  
    }  
  
    public static int percentInterest() {  
        return (int) (interest() * 100);  
    }  
}
```

Both `interest` and `percentInterest` are **independent of** specific instances of class `BankAccount`, as they both depend on constant `interest`, which is shared by all instances of `BankAccount`: they must be static methods.

Static or instance: examples

```
public class Car {  
    public static int addInterest(BankAccount account,  
                                  double interest) {  
        double withInterest = account.balance() * interest;  
        account.deposit((int) withInterest);  
    }  
}
```

The operation `addInterest` is independent of objects of class `Car`, as it only operates on objects of class `account`. Therefore, it is technically OK that it is a static method; however, it probably indicates **questionable design**: `addInterest` should probably be an **instance method of class `BankAccount`**.

Inheritance

Inheritance is a mechanism to **reuse** previously defined classes in the definition of new classes.

```
class C extends B
```

declares a class C that **inherits** from a class B.

- all members of B are also **implicitly members** of C
- C is a **subclass** (descendant, heir) of B;
conversely, B is a **superclass** (predecessor, ancestor) of C

```
class Account {  
    int balance;  
  
    void deposit(int amount)  
    { balance += amount; }  
}
```

```
class FullAccount extends Account {  
    void withdraw(int amount)  
    { deposit(-amount); }  
  
    void close()  
    { balance = 0; }  
}
```

Overriding

When creating a class by inheritance, we can also **override** (that is, redefine) any methods that are inherited from the superclass.

- a method's **signature** cannot change when overriding it (except for return types: see covariant redefinition rule)
- a method's **visibility** can only increase (e.g. from **protected** to **public**)
- a **static** method cannot be overridden

Inheritance and overriding support **flexible code reuse**.

```
class Account {
    int balance;

    void withdraw(int amount)
    { balance -= amount; }
}

class NoOverdrawnAccount extends Account {
    // redefinition of withdraw
    @Override
    void withdraw(int amount)
    { if (amount <= balance)
        balance -= amount; }
}
```

super: referencing the superclass

The keyword **super** denotes a reference to an instance of the superclass. It is useful to reuse code while overriding.

```
class Account {
    int balance;

    void withdraw(int amount)
    { balance -= amount; }
}

class NoOverdrawnAccount extends Account {
    @Override
    void withdraw(int amount)
    { if (amount <= balance)
        super.withdraw(amount); }
    // call withdraw's implementation
    // in Account
}
```

Shadowing: local variables over attributes

It is forbidden to have multiple variables (local variables or attributes) with the same name declared in the same scope (definition block). However, it is possible to have variables with the **same name** declared in **different but overlapping** scopes.

In this cases one variable implicitly **shadows** the other (that is only one variable is accessible):

- a local variable shadows an attribute with the same name
- use **this** to access the attribute

```
class BankAccount {  
    int balance;  
    void setBalance(int balance) {  
        // two variables named 'balance' are visible here:  
        // 1. the method argument 'balance'  
        // 2. the class attribute 'balance'  
        // 1. shadows 2.  
        this.balance = balance; } }  
}
```

Shadowing: attributes over attributes

It is forbidden to have multiple variables (local variables or attributes) with the same name declared in the same scope (definition block). However, it is possible to have variables with the **same name** declared in **different but overlapping** scopes.

In this cases one variable implicitly **shadows** the other (that is only one variable is accessible):

- a subclass attribute shadows a superclass attribute with the same name (note: this is not the same as overriding; suggestion: **avoid redefining attributes!**)
- use **super** to access the attribute in the superclass

```
class Car {
    String factoryId = "ZZZ0000";
}

class Volvo extends Car {
    String factoryId = "VLV0000";
}

System.out.println(new Car().factoryId); // "ZZZ0000"
System.out.println(new Volvo().factoryId); // "VLV0000"
```


The keyword `final` can also be used to restrict inheritance:

- a `method` marked as `final` cannot be overridden
- a `class` marked as `final` cannot be inherited from

Inheritance and types

Every class C corresponds to a type, which is a set of values and operations on those values. If C is a subclass of another class B , we call the type of C a **subtype** of the type of B .

Informally, the type of C is a **more specialized variant** of the type of B . “More specialized” means that everything we can do on objects of class B , we can also do on objects of class C (inheritance); but the latter may also offer more features (new attributes and methods).

We say that C and B are related by the “**is a**” relation: an object of class C **is an** object of class B (but not vice versa).

```
class Car {  
    void openDoor()  
    { /* ... */ }  
}
```

```
class Convertible extends Car {  
    void openTop()  
    { /* ... */ }  
}
```

A convertible **is a** car!

Static or instance: examples

```
public class Car extends Vehicle {  
  
    String factoryId() {  
        return "Car-" + super.id();  
    }  
  
}
```

Method `factoryId` depends on `super`, which is a reference to an `object` of the superclass `Vehicle`: it must be an instance member, otherwise `super` may not be defined.

Static or instance: examples

```
public class X extends Y {  
  
    @Override  
    public int z() {  
        return 42;  
    }  
}
```

Method `z` overrides a method with the same signature in the superclass `Y` of `X`. **Overriding** only applies to instance methods, and hence `z` cannot be static.

The Object class

The system class `Object` is implicitly a superclass of all Java classes: even when we do not use **extends**, every class **implicitly inherits from `Object`**, which provides a number of basic operations.

Methods of `Object` that is useful to override:

- **public boolean equals**(`Object obj`) is used to compare objects by value (according to the specific states of your objects)
- **public int hashCode**() is used to return a unique integer value for different object values, and should be consistent with `equals`:
`o1.equals(o2)` if and only if `o1.hashCode() == o2.hashCode()`

```
class BankAccount {  
    public boolean equals(Object other)  
        // if 'other' is not of class BankAccount, this does not work!  
    { return this.balance == other.balance; }  
  
    public int hashCode()  
    { return this.balance; } }
```

Checking the dynamic type of references

Sometimes it is useful to check the type of a reference within the program. To this end, the expression

```
variable instanceof RefType
```

evaluates to **true** if and only if `variable` is attached to an object whose **type** is `RefType` or a **subtype** of `RefType`.

- use **instanceof sparingly**: in most cases checking the type explicitly is not needed (type checking does that when compiling)
- the one case where it is useful is when **overriding equals**, which must take an argument of type `Object`

```
class BankAccount {  
    boolean equals(Object other) {  
        if (!(other instanceof BankAccount))  
            return false; // a different type, so cannot be value-equal  
        else { return this.balance == other.balance; } }  
}
```