Object-Oriented Programming

The object-oriented programming paradigm

Anders Jonsson & Federico Heras

2023-24

Course topics

Topic 1 Introduction and the concept of objects

Topic 2 The object-oriented programming paradigm

Topic 3 Object modelling and relations between objects

Topic 4 Inheritance and polymorphism

Topic 5 Abstract classes and interfaces

Topic 6 Reuse and study of problems solved using objects

Representing an object

- Object represented by two categories of information:
    - Attributes (or variables) that describe characteristics
    - Methods (or functions) that implement behavior

Theory session 2

Classes and instances

Implementation

Encapsulation

Fundamental concepts

Similarity among objects

- Object represented by two categories of information:
    - Attributes (or variables) that describe characteristics
    - Methods (or functions) that implement behavior
- How are objects similar in terms of this information?

Similarity among objects

- What do we mean by “key”?

Similarity among objects

- What do we mean by “key”?

- object with known functionality (open door)
- different keys have different characteristics
- however, we can group characteristics in categories

Object families

- Object family: set of objects of the same type
- The descriptive categories (the attributes) are *shared*
- The functionalities (the methods) are *shared*
- Each individual object has concrete characteristics within each category

Example

Object family: “Key”

- Attributes: color, shape, door
- Methods: openDoor

color=silver,
shape=round,
door=kitchen

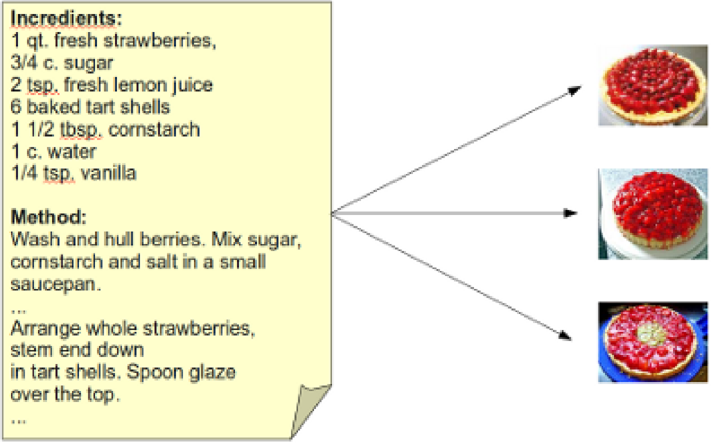
color=black,
shape=oval,
door=attic

color=gold,
shape=square,
door=bathroom

Classes and instances

- Fundamental units of object-oriented programming
- **Class**:
  - Abstract idea
  - Represents an object family
  - Defines the attributes and methods that are shared by all objects in this family
- **Instance**:
  - Concrete object
  - Belongs to an object family (that is, a class)
  - Assigns a concrete value to each attribute of the class

Relationship class-instance



Incredients:
1 qt. fresh strawberries,
3/4 c. sugar
2 tsp. fresh lemon juice
6 baked tart shells
1 1/2 tbsp. cornstarch
1 c. water
1/4 tsp. vanilla

Method:
Wash and hull berries. Mix sugar,
cornstarch and salt in a small
saucepan.
...
Arrange whole strawberries,
stem end down
in tart shells. Spoon glaze
over the top.
...

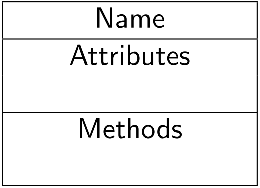
Elements of a class

- **Attribute**:
  - Describes a category that is common to all objects of the class
  - Implemented as a *variable*
  - When defining the class the *value* of the attribute is unknown!
- **Method**:
  - Describes a behavior that is common to all objects of the class
  - Implemented as a *procedure* or a *function*
  - Acts on the attributes, without knowing their concrete value
- The attributes and methods form the instance members of a class

Instances

- Each instance is created from a class definition
- Creation of an instance = assignment of values to the attributes
- Constructor: special method to create instances
- A method can only be applied on a concrete instance!

Graphical representation of a class



Name

Attributes

Methods

Exercises

- Define a class that represents persons
- Define a class that represents numbers
- Define a class that represents circles

Theory session 2

Classes and instances

Implementation

Encapsulation

Fundamental concepts

Implementation

- Translate the design to code
- There exist a variety of object-oriented programming languages
- Most are text-based, e.g. Java, C++ and Python
- Encapsulation: the code of an object is inaccessible from the outside

Review: variables

- **Variable**: an identifier that stores a value
- Strongly typed (Java, C, C++): each variable has an associated type and can only store values of that type

```
int value = 5;
String name = "John";
```

- Weakly typed (Python): a variable has no associated type

```
value = 5
value = ’John’
```

Review: procedures and functions

- **Function**: an identifier associated with a return type and a list of typed arguments

  ```
  int sum( int a, int b ) {
      int result = a + b;
      return result;
  }
  String concatenate( String s, String t ) {
      String result = s + t;
      return result;
  }
  ```

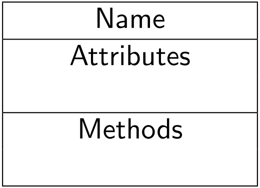
- Procedure: function whose return type is `void`

Types and classes

- A class can be used as a type! (of a variable, of a function, etc.)
- This applies to any class, both existing a newly defined
- An instance is stored in a variable whose type is the corresponding class
- An instance can also be returned by a function, used as an argument, etc.

Implementation of classes

- To implement the code of a class, the following components are needed:
  - Header: where the class is named
  - List of attributes
  - List of methods (including one or more constructors)
- The design already incorporates all of this information!



Name

Attributes

Methods

Attributes

- An attribute is simply a variable
- However, each instance has its own copy of the attribute
- The value of the attribute is assigned in the constructor method
- The value of the attribute can be modified in other methods

Methods

- A method is a function or a procedure
- The method has access to its arguments and to the attributes
- Listing the attributes as arguments to a method is an error!
- A method can only be applied on a concrete instance!

Special methods

- Constructor: create a new instance
- Getter: return the current value of an attribute
- Setter: change the value of an attribute

Class definition in Java

```
class Key {      // class header
   String color; // attribute "color"
   String shape; // attribute "shape"
   Door door;    // attribute "door"

   // constructor method
   Key( String c, String s, Door d ) {
      color = c;
      shape = s;
      door = d;
   }

   // getter
   String getColor() {
      return color;
   }
}
```

Create instances

```
class KeyProgram {
   public static void main( String[] args ) {
      Door d1 = new Door( "kitchen" );
      Door d2 = new Door( "attic" );
      Door d3 = new Door( "bathroom" );

      // instances of "Key"
      Key k1 = new Key( "silver", "round", d1 );
      Key k2 = new Key( "black", "oval", d2 );
      Key k3 = new Key( "gold", "square", d3 );
   }
}
```

Naming standards

- A class always starts with an uppercase letter
- An attribute or method starts with a lowercase letter
- Each subsequence word starts with an uppercase letter
- Examples:
  - `class MyClass`
  - `int myAttribute`
  - `void myMethod()`
- Not strict requirements (violations will not lead to compiler errors)

Theory session 2

Classes and instances

Implementation

Encapsulation

Fundamental concepts

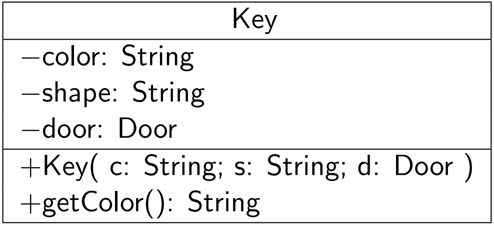
Visibility

- To each instance member we can associate a level of visibility
- Private visibility: internal member, restricted access
- Public visibility: external member, open access
- Mechanism for achieving encapsulation

Application of visibility

- Common practice:
  - all attributes have private visibility
  - most methods have public visibility
  - auxiliary methods have private visibility
- To access the value of an attribute it is necessary to call a method
- Not strict requirements (violations will not lead to compiler errors)

Example



Key

−color: String
−shape: String
−door: Door

+Key(c: String; s: String; d: Door)
+getColor(): String

Class definition in Java

```java
public class Key {       // class header
   private String color; // attribute "color"
   private String shape; // attribute "shape"
   private Door door;    // attribute "door"

   // constructor method
   public Key( String c, String s, Door d ) {
      color = c;
      shape = s;
      door = d;
   }

   // getter
   public String getColor() {
      return color;
   }
}
```

Class definition in C++

```
class Key {           // class header
private:              // common visibility
   std::string color; // attribute "color"
   std::string shape; // attribute "shape"
   Door * door;       // attribute "door"

public:               // common visibility
   Key( std::string c, std::string s, Door * d ) {
      color = c;
      shape = s;
      door = d;
   }

   std::string getColor() {
      return color;
   }
};
```

Exercises

- Implement the Java code of the classes defined earlier

Sending messages

- Instances interact by sending messages
- Send message = delegate a task to another instance
- Each message consists in three components:
  1. The receiving instance
  2. The method to be invoked (among the methods of the associated class)
  3. An assignment of values to the arguments of the method

Receiving messages

- When receiving a message, an instance executes the indicated method
  - The arguments take the values specified in the message
  - The method is also influenced by the current value of the attributes
- If the method is a procedure, it returns nothing
- If the method is a function, it returns the appropriate result

Example

```
public class Key {       // class header
   private String color; // attribute "color"
   private String shape; // attribute "shape"
   private Door door;    // attribute "door"

   // constructor method
   public Key( String c, String s, Door d ) {
      color = c;
      shape = s;
      door = d;
   }

   public void unlock( Door someDoor ) {
      if ( door == someDoor ) door.unlock();
   }
}
```

Example

```
public class KeyProgram {
   public static void main( String[] args ) {
      Door d1 = new Door( "kitchen" );
      Door d2 = new Door( "attic" );
      Door d3 = new Door( "bathroom" );

      Key k1 = new Key( "silver", "round", d1 );
      Key k2 = new Key( "black", "oval", d2 );
      Key k3 = new Key( "gold", "square", d3 );

      k1.unlock( d2 ); // fails
      k3.unlock( d3 ); // works
   }
}
```

API of the “Key” class

- The API describes the public (visible) members of a class
- The implementation of methods is not considered public

```
public class Key {
   public Key( String c, String s, Door d );
   public String getColor();
   public void unlock( Door someDoor );
}
```

Calls vs messages

- Call (to a procedure or function):
  - Does not have an addressee
  - Can only be interpreted in one way
- Message:
  - Has a concrete addressee (an instance)
  - The interpretation of the message depends on the instance

Object-oriented design

1. Identify the objects that will participate in the solution
2. If an object is already defined, reuse and/or modify
3. If an object is not defined, create a new definition
4. Determine how the objects interact in the solution

Theory session 2

Classes and instances

Implementation

Encapsulation

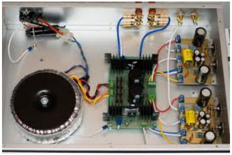
Fundamental concepts

Abstraction

- “The process of removing physical, spatial, or temporal details or attributes in the study of objects or systems to focus attention on details of greater importance”
- Focus on the essential
- Hide what is irrelevant
- Important tool for reducing the complexity of a problem

Encapsulation

- Hide the internal representation of an object
- Mechanism that increases the level of abstraction
- Example: amplifier



$\Rightarrow$

Encapsulation

- The external access to the object is restricted
- API (Application Programming Interface): description of the external interface
- Only the object itself has access to the internal representation

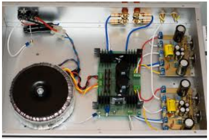
Reuse

- “Action or practice of using an item [...] to fulfil a different function”
- The ability to reuse relies [...] on the ability to build larger things from smaller parts
- Take advantage of existing elements
- Avoid duplicating the effort needed to create a new element
- Important tool for reducing the work effort

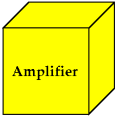
Reuse in object-oriented programming

- In object-oriented programming there are many different concepts that can be reused:
    - reuse the code
    - reuse the object
    - reuse the API
    - reuse the design

Reuse the object

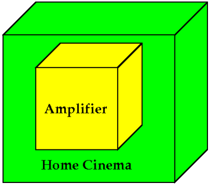


⇒



Amplifier

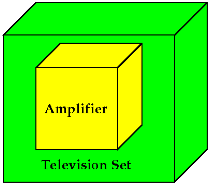
⇒



Amplifier

Home Cinema

⇒



Amplifier

Television Set

Objectives

- Main objectives in object-oriented programming:
  - Design and implement programs that behave correctly
  - Promote abstraction, for example using encapsulation
  - Promote reuse of different program components
- To achieve these objectives it is necessary to
  - learn all the concepts presented in the theory sessions
  - translate these concepts to design and implementation
  - practice the design and implementation of concrete programs

Summary

- Classes and instances
- Instance members (attributes and methods)
- Implementation
- Visibility of members (public or private)
- Special methods
- Messages
- Abstraction and encapsulation