Objects and Classes

Lecture 10 of TDA 540 Object-Oriented Programming

Jesper Cockx

Fall 2018

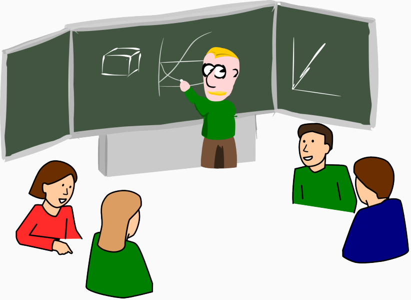
Chalmers University of Technology — Gothenburg University

History of object-oriented programming

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

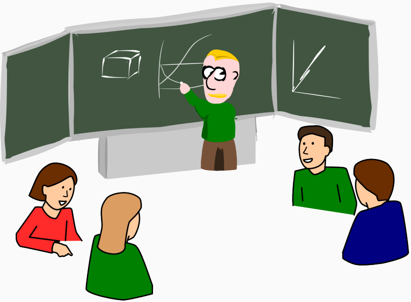
Objects

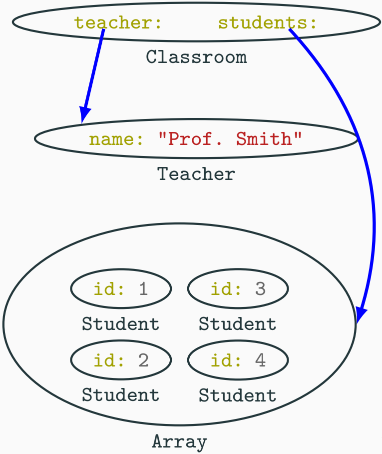
Object-oriented programming: a program consists of a collection of objects that interact with each other and together produce the desired result.



Objects

Object-oriented programming: a program consists of a collection of objects that interact with each other and together produce the desired result.





Objects in programming

Each object has an internal state and an external interface

Example object: a glass of water

What is the state? What is the interface?

State:

Operations:

Example object: a glass of water

What is the state? What is the interface?

State:

- Current volume of water
- Maximum volume

Operations:

Example object: a glass of water

What is the state? What is the interface?

State:

- Current volume of water
- Maximum volume

Operations:

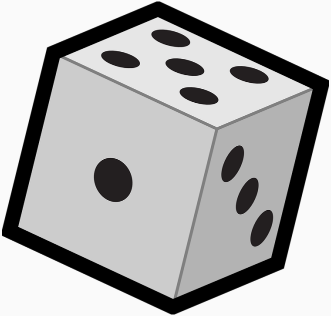
- Measure current volume
- Add water
- Remove water
- ...

Example object: six-sided die

What is the state? What is the interface?

State:

Operations:



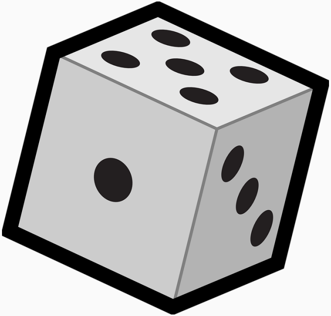
Example object: six-sided die

What is the state? What is the interface?

State:

- Current value on top

Operations:

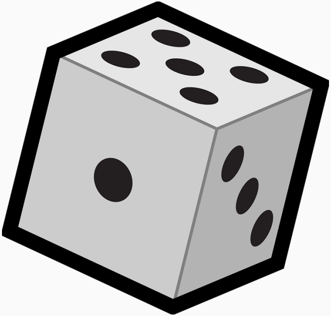


Example object: six-sided die

What is the state? What is the interface?

State:

- Current value on top



Operations:

- Read current value
- Roll the die
- ...

Example object: coin purse

What is the state? What is the interface?

State:

Operations:

Example object: coin purse

What is the state? What is the interface?

State:

- Number of coins of each kind

Operations:

Example object: coin purse

What is the state? What is the interface?

State:

- Number of coins of each kind

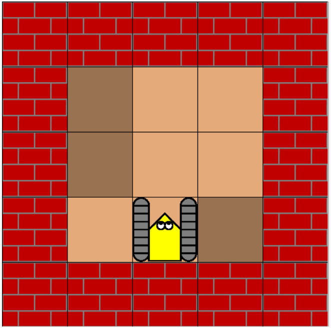
Operations:

- Count number of coins of one kind
- Count total value of all coins
- Add new coins
- Pay a given amount
- ...

Example object: robot

What is the state? What is the interface?

State:



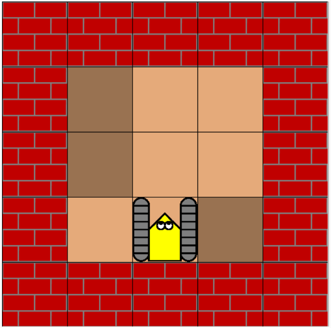
Operations:

Example object: robot

What is the state? What is the interface?

State:

- Current position
- Current direction



Operations:

Example object: robot

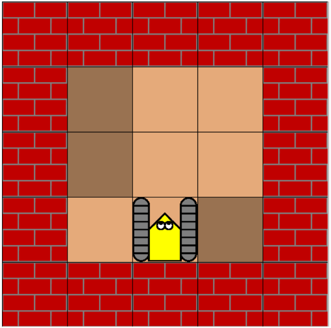
What is the state? What is the interface?

State:

- Current position
- Current direction

Operations:

- Get current position
- Get current direction
- Move one step
- Turn left/right
- ...



Classes

A class describes a collection of objects with the same interface:

- `Die` is the class of all dice
- `Glass` is the class of all water glasses
- `CoinPurse` is the class of all coin purses
- `Robot` is the class of all robots

```
class Glass {  // user-defined class
  private double volume; // state
  void addWater(double amount) { // operation
    volume = volume + amount;
  }
```

What is in a class?

A class defines:

- How objects of that class are represented in computer memory (the attributes)
- What methods are available on objects of the class (the methods)
- How to create new objects of that class (the constructors)

Each class also defines a new type.

Objects vs. classes

A class is a static entity:
it refers to a piece of code.

An object is a dynamic entity:
it is only created when the program executes.

An object is an instance of a certain class.

Attributes

An attribute (also called an instance variable or a field) represents part of an object's state.

- Each object has its own copy of the attributes
- Attributes can be of *primitive* or *reference* type
- `final` attributes cannot change once the object has been created

Attributes are declared in the class body:

```
class Glass {
  double volume;          // current contents in ml
  final double maxVolume; // maximum volume
  // ...
}
```

Methods

A method (also called an instance method or a member function) represents an operation that can be executed on objects of the class.

A method can modify the object state and/or return information about the object state.

Methods are declared in the class body:

```
class Glass {
  // ...
  public void addWater(double x) { volume += x; }
  // ...
}
```

Getters and setters

Two common kinds of methods:

- Getters (= accessors) return the value of one attribute:

```java
public double getVolume() {
  return volume;
}
```

- Setters (= mutators) change the value of one attribute:

```java
private void setVolume(double newVolume) {
  volume = newVolume;
}
```

Constructors

A constructor is a special method that creates a new object of the class.

- Constructors have the *same name* as the class
- A constructor has no return type (not even `void`!)
- It should give an initial value to all attributes (uninitialized attributes get default value)

Constructors are declared in the class body:

```
public Glass(double size) {
  volume = 0;
  maxVolume = size;
}
```

Using a constructor

To use a constructor, we use the `new` keyword:

```
Glass glass = new Glass(100);
```

The result of `new Glass(100)` is a reference to the new object.

Default constructors

If a class has no constructor, Java automatically generates one with no parameters.

For example, for `Glass` we get

```
public Glass() { }
```

It’s a good idea to always give a constructor.

The life of an object

What we can do with an object `obj`:

initialize: using a constructor

read state: using getters or other methods

modify state: using setters or other methods

dispose: implicit in Java when object is no longer used

```
Glass glass;
// create empty glass
glass = new Glass(300);
if (glass.getVolume() == 0) {
  System.out.println(
    "Glass is empty");
}
// add some water
glass.addWater(100);
System.out.println(
  "Now the glass contains "
  + glass.getVolume() + " ml.");
// glass is deleted
```

Garbage collection

In some languages (C++) the programmer has to ’destruct’ an object when it’s no longer needed.

In Java, this is done automatically when the object is no longer used (= garbage collection).

⇒ you don’t have to worry

Designing a class

- Think about the responsibilities of this class
- Specify the public interface (methods + constructors)
- Determine the instance variables
- Implement the constructors and methods
- Test the class

Testing a class

- Construct one or more objects
- Invoke one or more methods
- Print out the results
- Compare to the expected results

Live coding:
implementing and testing `Glass`

15 min. break

Kahoot: objects and classes

Encapsulation / information hiding

An important role of an object is to hide information from the rest of the program.

The client only has to know the public methods and constructors = the API (*Application Programming Interface*).

The (private) state can change while the rest of the program stays the same.

$\Rightarrow$ Abstraction!

Information hiding: example

Public interface:

```
class Glass {

  /* attributes invisible */

  Glass(double size) {
    /* body invisible */
  }

  double getVolume() {
    /* body invisible */
  }

  void addWater(double amount) {
    /* body invisible */
  }
}
```

Client code:

```
Glass glass;
glass = new Glass(500);

// we don't have to worry how addWater
// and getVolume are implemented

glass.addWater(300);
if (glass.getVolume() > 100) {
  System.out.println(
    "Can drink water!");
}
```

Visibility of members

The visibility of a class member (attribute or method) determines where in a program we can refer to that member:

- `private`: `x` is only visible in the enclosing class
- `protected`: `x` is visible within the same package
- `public`: `x` is visible everywhere in the program

Visibility of members: examples

```
package p;

class A {
  private int a;
  protected void x()
  { a = 3; }
  public void y()
  { a = 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.x();    // OK
    o.y();    // OK
    o.z();    // ERROR!
}
```

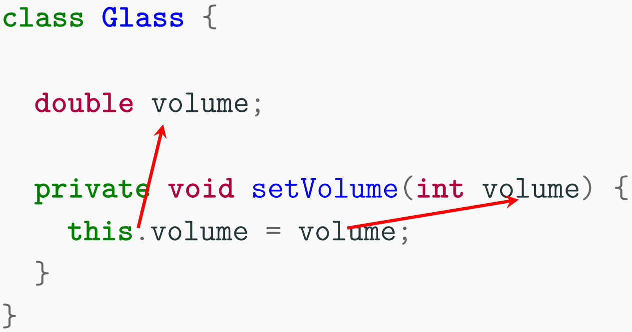
Visibility of members: examples

```
package p;

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

```
package q;

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

Shadowing and the `this` reference

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



```
class Glass {

  double volume;

  private void setVolume(int volume) {
    this.volume = volume;
  }
}
```

The local variable `volume` shadows the attribute `volume`.

Overloaded methods

Overloading: A class can have multiple methods with the same name but different signatures:

- Different number of arguments
- Different types of arguments
- Different return type

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

Overloading: example

```java
public class Glass {
  // ...
  public void addWater(double amount) {
    currentVolume += amount;
  }
  public void addWater(String amount) {
    addWater(Double.parseDouble(amount));
  }
  public void addWater() {
    addWater(100);
  }
}
```

Overloaded constructors

```
class Glass {
  double current;
  final double maximum;
  Glass() {
    this.maximum = 300;
    this.current = 0;
  }
  Glass(double maximum) {
    this.maximum = maximum;
    this.current = 0
  }
  Glass(double max, double curr) {
    this.maximum = max;
    this.current = curr;
  }
}
```

```
// client code
Glass g1 = new Glass();

Glass a2 = new Glass(200);

Glass a3 =
  new Glass(400, 200);
```

Static members

A static member belongs to the whole class, not an individual object.

- A `static` attribute is shared among all object of the class
- A `static` method can only use static attributes and other static methods
- A constructor can never be `static`

Static members are accessed using the class name:

```
class CoinPurse {
  static int[] COIN_SIZES =
    { 1 , 2 , 5 , 10 };
  // ...
}
```

```
int[] coins =
  CoinPurse.COIN_SIZES;
for (i : coins) {
  // ...
}
```

The main method

A `static` method can be called *without* any object.

The method `main` with signature

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

runs first whenever we run a Java program.

From `main` all objects in the program are created as the program continues executing.

When to use static members?

Instance members: state + operations of objects

OPERATION	INSTANCE
create object:	`Die d = new Die();`
modify object state:	`d.roll();`
read current object state:	`if (d.lastRoll() = 6) ...`

Static members: global operations + state

ITEM	STATIC
constant:	`double angle = Math.PI/4.0;`
math operation:	`double y = sqrt(x);`
global state:	`int[] sizes = CoinPurse.COIN_SIZES;`

Static or instance?

Rule of thumb:

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

1. Yes: you probably need a static member
2. No: you should go with an instance member

In most cases, the answer should be no!

The `Object` class

`Object` is a special class which contains all Java objects.

We say `Object` is a superclass of all other classes
(see next lecture for more about superclasses).

`Object` provides basic operation available on all objects:

- `public String toString()`: return a textual representation of the object
- `public boolean equals(Object obj)`: check if two objects have the same value
- `public int hashCode()` return a unique identification number of the object

We can override these methods in a class to give our own definition.

Overriding `Object` methods

```java
class Glass {
  public String toString() {
    return String.format("Glass(%f,%f)",
                         current, maximum);
  }

  public boolean equals(Object other) {
    // This does not work because 'other'
    // doesn't have type Glass!
    return this.current == other.current
        && this.maximum == other.maximum;
  }
}
```

Checking the dynamic type of references

```
variable instanceof RefType
```

is `true` if and only if `variable` refers to an object of class `RefType`.

- Use `instanceof` sparingly: in most cases checking the type explicitly is not needed.
- One case where it is useful is when overriding `equals`.

Using `instanceof`

```
class Glass {
  public boolean equals(Object other) {
    if (other instanceOf Glass) {
      otherGlass = (Glass)other;
      return this.current == otherGlass.current
          && this.maximum == otherGlass.maximum;
    } else {
      return false;
    }
  }
}
```

What's next?

Next lecture on Tuesday at 10:00:
Subclasses and interfaces.

To do:

- Read the book:
    - Today: chapter 8
    - Next lecture: chapter 9
- Continue on lab #5