The object-oriented computational paradigm

Computational paradigm =
high-level programming languages sharing the same execution model

Main computational paradigms

Paradigm	**Model**	**Launching a program corresponds to**
Imperative	state as memory abstraction	execute a command
Object-oriented	objects and method calls	call a method on an object
Functional	functions and application	evaluate an expression
Logic	Horn clauses and deduction	resolve a goal

Remarks: most modern programming languages support several paradigms!

A brief history

First object-oriented languages

- Simula (1965)
- Smalltalk (1972)
- Eiffel (1986)

Mainstream object-oriented languages

- statically typed: C++ (1985), Java (1995), C# (2000)
- dynamically typed: Python (1990), JavaScript/ECMAScript (1995)

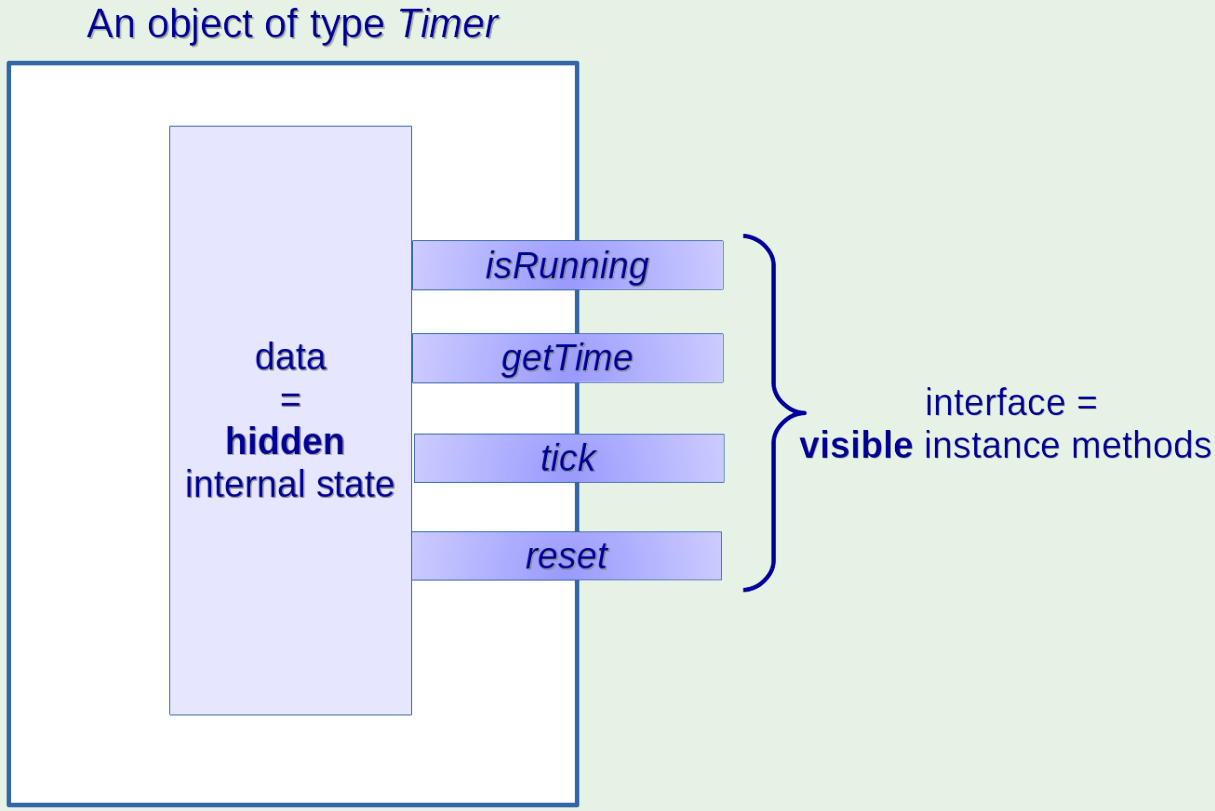
Most recent ones

- Scala (2004), strong integration of functional and o-o paradigms
- Kotlin (2011), Java “enhancement”, support for Android

Both statically typed and based on the Java Virtual Machine (JVM)

What is an object?

Example: a timer



Remark: instance is synonym of object

Object's data

Encapsulation

- object's data are preferably hidden from the outside
- it means that they are accessible only locally

Remarks

- object's state = object's data
- object's data can be modified $\Rightarrow$ object's state can change

Instance methods

Instance methods allow interactions with objects

- to interact with *o* a method on *o* can be called
- Terminology:
  - method call/method invocation/message sending
  - the selected object is called target/receiver
- A method call may
  - access/modify the data of the target object
  - have arguments and a returned value as a function call

Syntax of method call

```
Exp ::= Exp '.' MID '(' (Exp ( ',' Exp)*)? ')'
```

- `MID`: a valid method name
- Example:

```
// 'reset' called on 'timer' with argument '42'
timer.reset(42);
```

Instance methods

Specification of timer instance methods

- **boolean** `isRunning()`
  - checks whether the count down is not finished yet
  - returns true if the timer has not reached time 0
  - returns false if the timer has reached time 0
- **int** `getTime()`
  - returns the current time of the timer in seconds(= seconds until 0 is reached)
- **void** `tick()`
  - decreases of one second the time of the timer if greater than 0
  - no changes if the time is 0
- **int** `reset(int minutes)`
  - resets to `minutes` the time of the timer
  - the time is in minutes
  - returns the previous time of the timer in seconds
  - throws an exception if `minutes<0` or `minutes>60`

Instance methods

A closer look at the instance methods of a timer

- `isRunning` and `getTime` are query methods:
  - they inspect the internal state (=data) of the timer to show some details
  - they cannot modify it
- `tick` may modify the internal state (=data) of the timer
- `reset` is both a query and a modification method

Fields of an object

General facts

- the data of the object are stored in fields(=instance variables/attributes)
- the content of fields can usually be changed

Two possible implementations of the internal state of a timer

- ```
  // total time in seconds
  int time; // invariant: 0 <= time <= 3600
  ```

- ```
  // time in minutes and seconds (total = seconds+60*minutes)
  int seconds; // invariant: 0 <= seconds < 60
  int minutes; // invariant: 0 <= minutes <= 60
  ```

How objects are created?

Class-based languages

- most common approach: C#, C++, Java, Smalltalk, ...
- objects created through classes

Object-based languages

- JavaScript, Self
- objects created without a class

Remark

- objects are created dynamically
- they are dynamically allocated on the heap

Classes

A Java class for timers

```
public class TimerClass {
                    // private field of the object = hidden internal state
   private int time; // in seconds, invariant: 0 <= time <= 3600

   // public instance methods = interface of visible operations
   public boolean isRunning() {   // 'this' is the target of the method
       return this.getTime() > 0;
   }
   public int getTime() {         // 'this' is the target of the method
      return this.time;
   }
   public void tick() {           // 'this' is the target of the method
      if (this.isRunning())
         this.time--;
   }
   public int reset(int minutes) { // 'this' is the target of the method
      if (minutes < 0 || minutes > 60) // Java exceptions are special objects
         throw new IllegalArgumentException();
      int prevTime = this.getTime();
      this.time = minutes * 60;
      return prevTime;
   }
}
```

Classes

A demo with a main method for TimerClass

```
public class TimerClass {
   ...
   public static void main(String[] args) {
      // creates an object of class TimerClass, assigns its reference to t1
      TimerClass t1 = new TimerClass();

      // calls reset(1) on the object referenced by t1
      t1.reset(1);

      // creates an object of class TimerClass, assigns its reference to t2
      TimerClass t2 = new TimerClass();

      // calls reset(2) on the object referenced by t2
      t2.reset(2);
   }
}
```

Classes

Classes and objects in a nutshell

- A class provides an implementation for objects of the same type
- Objects are dynamically created from classes
- In most languages (Java, C#, JavaScript, Python, . . .) objects are references to dynamic memory locations
- Objects created from class *C*, are called instances of *C*
- Objects are deallocated automatically in most languages (garbage collection in Java, C#, JavaScript, Python, . . .), manually in C++
- All objects of the same class share the same instance methods
- Objects have their own fields, even though declared in the same class
- Method call on target *o* is expected to change only the internal state of *o*

Classes

Use of keyword `this` in instance methods

```
public void tick() {
   if (this.isRunning()) // 'isRunning()' is called on 'this'
      this.time--;       // 'time' of 'this' is read and modified
}
```

- **`this`** is the target object on which `tick()` and `isRunning()` are called
- the same object **`this`** is used to access its field `time`

Methods and fields of other objects

Inside a method, methods and fields different from **`this`** can be used.
Examples:

```
public boolean overlaps(Point p) {
    return this.x == p.x && this.y == p.y; // 'p' is not 'this'
}

public int compare(Shape shape1, Shape shape2) {
   double area1 = shape1.area(); // 'shape1' is not 'this'
   double area2 = shape2.area(); // 'shape2' is not 'this'
   return area1 < area2 ? -1 : area1 == area2 ? 0 : 1;
}
```

Classes

Object creation

- Syntax: `'new'CID '('(Exp ( ','Exp)*)? ')'`
  `CID` is a valid class name
- Semantics:
  - a new object of class `CID` is dynamically created
  - its fields are initialized
  - the reference to its dynamic memory location is returned
- Remark: during object creation arguments can be passed to properly initialize the fields of the object

Classes

Static and dynamic types

- Objects created from class *C* have dynamic type *C*
- Example:

  ```
  new TimerClass()
  ```

  returns a reference to an object of (dynamic) type `TimerClass`
- In languages as Java, C#, C++ a class defines also a static type
- Example:

  ```
  TimerClass t1, t2;
  ```

  declares variables of (static) type `TimerClass`.
  `t1` and `t2` can only contain values compatible with type `TimerClass`