Introductory Programming
Object oriented programming I, sections 4.0-4.5

Anne Haxthausen[a]

IMM, DTU

1. Methods (declaration and invocation) (sections 4.2 (+ 4.4))
2. Classes and objects (sections 4.0-4.1 (+ 4.5))
   - how to define a class
   - how to create an object
   - how variables of class types behave
   - how to access the data and methods of an object
3. Encapsulation via visibility modifiers (public and private) (section 4.1)
4. Method overloading and overload resolution based on signatures (section 4.3)
5. Scope rules (sections 4.1, 4.2)
6. Summary

a. Parts of this material are inspired by/originate from a course at ITU developed by Niels Hallenberg and Peter Sestoft on the basis of a course at KVL developed by Morten Larsen and Peter Sestoft.

Functions

A (mathematical) *function* takes a number as argument and returns a number as result.

Example of a function: the function *square* takes a number x and returns $x \cdot x$ as result:

$$\textit{square}(x) = x \cdot x$$

Examples of applications of the function:

x	*square(x)*
1.2	1.44
4.4	19.36
3.0	9.00
43.0	1849.00

Methods in Java

Methods in Java (and procedures/subroutine in other languages) are similar to functions in mathematics: they can take arguments and they produce results.

A Java method `square` corresponding to the function *square*$(x) = x \cdot x$ can be declared in this way:

```
static double square(double x) {
  return x * x;
}
```

The method takes an *argument* `x` of type **double** and returns a *result* of type **double**. The result is `x` times `x`.

A method `seventimes` that multiplies a number with seven can be declared in this way:

```
static double seventimes(double x) {
  return x * 7.0;
}
```

The method takes an argument `x` of type **double** and returns a result of type **double**.
The result is seven times `x`.

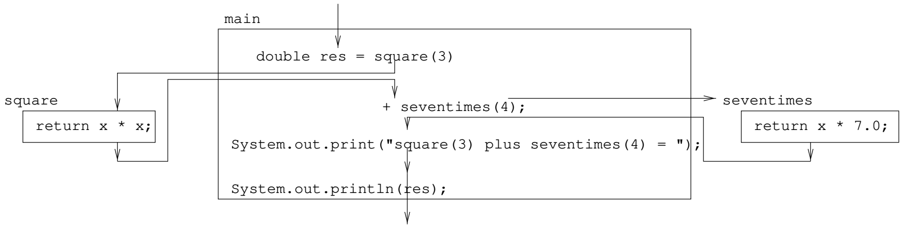
Methods (example: Methods1.java)

A method must be declared inside a class.

```
public class Methods1 {
  public static void main(String[] args) {
    double res = square(3) + seventimes(4);
    System.out.print("square(3) plus seventimes(4) = ");
    System.out.println(res);
  }
  static double square(double x) {
    return x * x;
  }
  static double seventimes(double x) {
    return x * 7.0;
  }
}
```

A method invocation `square(3)` or `seventimes(4)` is an expression. Hence, a method invocation can be a subexpression of another expression.

Execution of a program containing method invocations (Methods1.java)



General format for method declarations:

modifier result-type method-name (*parameters*) {

  statements and declarations

}

modifier can be **static** — you will learn about other possibilities later.

result-type is the type of values in the **return** statements; **void** if nothing is returned.

method-name is the name of the method.

parameters is a comma separated list of types and names of *formal* parameters.

statements and declarations make up the *method body* and are executed when the method is *invoked*.

General format for method invocations:

method-name (*arguments*)

arguments are a comma separated list of argument expressions (also called *actual* parameters).

Effect:

First the argument expressions are evaluated and their values assigned to the formal parameter names (that behave as variables); then the statements and declarations of the method body are executed.

When the method executes a return statement (of the form **return** *expression*;), *expression* is evaluated to some value v and the control of the program returns to the location where the invocation was made; the resulting value of the method invocation is v.

Why use methods?

A method encapsulates and gives name to a collection of statements (and declarations).

It is useful when:

- it constitutes a natural operation
- it can be used to decompose the body of a method that otherwise would have become longer than one page (40–50 lines).
  (Read section 4.4 about method decomposition.)
- it can be re-used, i.e. when (almost) the same piece of code can be used several places in the program.
  The alternative of programming with copy-and-paste gives un-readable and un-maintainable programs.

Classes and objects

Informally

- A *class* represents a concept: time, appointment, car, cow, person, ...
- An *object* represents a thing, an instance of a concept: a particular time, a particular car, a particular cow, a particular person, ...
- A class has a collection of *methods*: those operations (functions) that can be applied to its objects.

In Java

- A *class* corresponds to a type, like `int`, `double`, `boolean`, ...
- An *object* corresponds to a value, like `17`, `18.01`, `false`, ...
- A *method* corresponds to an operation, like $+$, $-$, ...

Three steps in using classes, objects and methods

1. Define a *class* (incl. its methods).
2. Create objects of the class.
3. Use the methods of the objects.

Some classes (e.g. `String`) are already defined in a class library. In that case you can skip step one.

If a class (e.g. `Keyboard`) has *static* methods, these can be used without step 2-3.

Definition of classes

A class contains declarations of its *members*:

- *data* (constants and variables) (also called *fields*)
- *methods*

A class declaration gives name to a class.

Example

```
class Time {
  private int hours, min; //hours and minutes since midnight

  public Time(int h, int m) {hours = h; min = m;}
  public int getmin() { return min; }
}
```

An object of class `Time` has fields `hours` and `min` representing a point in time, a constructor (method) `Time` that can be used for initializing these fields and a method `getmin`.

Creations of objects

An object is an instance of a particular class. It has fields and methods, as specified in the class. The *state* of the object is the contents of its fields. The methods of the object can be used for changing and reading its state.

In Java an object is created by applying the **new** operator. This invokes a *constructor*: a method which has the same name as the class and which initializes the object. A constructor has no result type.

Example

new `Time(12, 35)`
creates (a reference to) an object of class `Time` with `hours == 12` and `min == 35`.

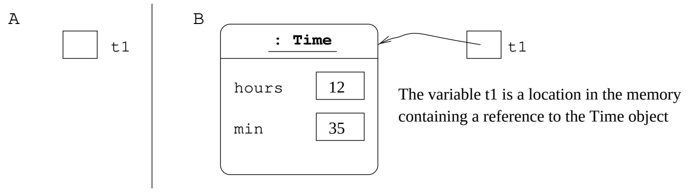
Variables having class types

A variable contains either

- a primitive value, or
- a *reference (henvisning)* to an object.

Example

```
Time t1;                //A   declaration
t1 = new Time(12, 35);  //B   initialization
```



Declaration and initialization can be made in one step:

```
Time t1 = new Time(12, 35);
```


Access to the data fields of an object

is made by the dot notation:

```
t1.min = (t1.min + 2) % 60 //only legal, if min is not private
```

Invocation of the methods of an object

is made by the dot notation:

```
... t1.getmin() ...
```


Description of methods in an extended Time class

Assume given a Time object t1.

`Time(h, m)` creates a new `Time` object representing the time by `hours` and `min`.

`t1.gethours()` returns for `t1` the number of hours since midnight.

`t1.getmin()` returns for `t1` the number of minutes (over hours) since midnight.

`t1.toString()` returns the time in the form hours.min, e.g. `12.27`.

`t1.passtime(m)` increases the time of `t1` with `m` minutes.

`t1.plus(m)` returns a new `Time` object that is `m` minutes ahead of `t1`.

`t1.to(t)` returns the number of minutes from `t1` to `t`.

`t1.before(t)` returns true if `t1` is earlier in the day than `t`.

An implementation of the Time class

```
class Time {
  private int hours, min; // hours and minutes since midnight

  public Time(int h, int m) {hours = h; min = m;}

  public int gethours() {return hours ;}

  public int getmin() { return min; }

  public String toString() { return hours + "." + min; }
```

```
    public void passtime(int m) {
      int totalmin = 60 * hours + min + m;
      hours = (totalmin / 60) % 24; min = totalmin % 60;
    }

    public Time plus(int m) {
      int totalmin = 60 * hours + min + m;
      return new Time( (totalmin / 60) % 24, totalmin % 60);
    }

    public int to(Time t)
     { return 60 * t.hours + t.min - 60 * hours - min; }

    public boolean before(Time t)
     { return (hours < t.hours) ||
              (hours == t.hours && min <= t.min);
     }
  }
```


Example of a program that uses `Time`

```
public class TestofTime {
  public static void main(String[] args) {
    Time clock1, clock2;

    clock1 = new Time(9,10);
    System.out.println("clock1 shows " + clock1);

    clock2 = clock1.plus(60);
    System.out.println("clock2 shows " + clock2);

    clock1.passtime(40);
    System.out.println("after 40 minutes clock1 shows " + clock1);
  }
}
```


Output from `TestofTime`

```
clock1 shows 9.10
clock2 shows 10.10
after 40 minutes clock1 shows 9.50
```


Encapsulation

By *encapsulation* of an object we mean that the user is only allowed to access the data of the object through its methods.

Example `clock1.min` ought to be illegal, but `clock1.getmin()` is ok

The user should only know the interface of the associated class:
for each method: name, argument types, result type, and what happens when it is *invoked*.
By hiding the internal data representation and algorithms (method bodies) for the user, we can change the implementation of this without problems.

Example

We could change the representation of time to number of minutes since midnight:

```
class Time {
  private int min; //minutes since midnight
  public Time(int h, int m) {min = (h * 60 + m) % 1440;}
  ...
}
```

without changing the invocations of the `Time` methods in the `TestofTime` class.

The visibility modifiers private and public

Encapsulation in Java is obtained by *visibility modifiers* in data and method declarations:

- **private**: for members that should only be directly used inside the class.
- **public**: for members that can also be directly referenced outside.

Recommendation:

- Declare variables **private** so that the state of the objects becomes encapsulated.
- Declare auxiliary methods **private**.
- Declare other methods (incl. constructors) **public**.
- Declare constants that should be known outside the class **public** (and **static**), otherwise **private**.

Class libraries, packages and import

Has been explained in lecture 2.

Signatures of methods

The *signature* of a Java method consists of *(i)* the name of the method, and *(ii)* the list of parameter types.

Examples

```
gethours()
passtime(int)
max(int, int)
is_ok(boolean)
```


Basic concept: overloading

When two or more methods are declared with

- the same name, but
- different signatures

the name is said to be *overloaded* (overlæst).

Example in Java:

The `Time` class could have had two constructors:

```
public Time(int h, int m) {hours = h; min = m;}
public Time(int h) {hours = h; min = 0;}
```

They have the same name, but different signatures:

`Time(int, int)` and `Time(int)` resp.

Overload resolution

If an overloaded name is used in a method call, which declared method does it belong to?

By *overload resolution* we mean the decision of this.
In Java, it is decided by matching the types of the arguments with the types in the signature.
Other programming languages have other rules.

Examples in Java

`Time(12, 10)` belongs to the first declaration

`Time(12)` belongs to the second declaration

Basic concept: scope

Definition

The *scope* of a variable or constant is that part of the program in which it can be referred to (i.e. read or assigned a new value).

Visibility rules for variables and constants in Java

In Java we distinguish between:

- *fields/instance data*: variables and constants declared in a class
- *local data*: variables and constants declared in a method

Rules:

- The scope of *a field* of a class is the *whole* class.
- The scope of *local data* goes from the point *after* its declaration until the end of that block in which it is declared.
- *Formal parameters* of a method behave as if they were declared in the top of the method body.
- The scope of variables declared in the header of a for loop is the remaining part of the loop.
- It is illegal to declare a new local variable/constant inside the scope for another local variable/constant with the same name.
- It is legal to declare a local variable/constant inside the scope for a field with the same name. In that case the local variable/constant will make a *shadow* over the field which then becomes invisible.

Scope: example 1

```
class Time
{
  public Time(int h, int m) { hours = h; min = m; }

  private int hours, min; //hours and minutes since midnight

  ...

  public Time plus(int m) {
      int totalmin = 60 * hours + min + m;
      return new Time( (totalmin / 60) % 24, totalmin % 60);
  }

}
```

Scope for loop variable: example

```
public void loop(){
  ...
  for (int n=1;                  // n invisible
        n<=10;                   // n visible
        n = n + 1)               // n visible
     System.out.println(n * n);  // n visible
  ...                            // n invisible
}
```


Scope: examples of shadowing

```
class Time
{
  private int hours, min; //min #1

  ...

  public Time plus(int min) { //min #2
    int totalmin = 60 * hours + getmin() + min; //min #2 visible
    return new Time(..., totalmin % 60);        //min #2 visible
  }
}
```


Summary: Java program structure

Program

Classes

Fields

Constructors

Methods

Declarations

Statements

Summary: classes

Two goals of classes in Java

(1) as "containers" for related methods (e.g. `Keyboard` contains `readInt` etc.):

In this case all data and methods are **static**.

(2) as blueprints for objects (e.g. `Time` as skeleton for `clock1` and `clock2`):

In this case some data and methods are not **static**.

Fields in the object contains the state of the object.
Constructors initialize the object (by initializing the fields).
Methods in the object makes it possible to change the state or read the state.

Visibility of fields and methods is indicated by **public** and **private**.

Summary: objects

An object is a (often composite) value belonging to a particular class.

An object can be *created* by the **new** operator (that invokes a constructor from the class):

```
clock1 = new Time(12, 35);
```

or with a method that returns an object:

```
clock2 = clock1.plus(60);
```

You can access (read or change) a **public** (but not a **private**) field in an object via the dot notation:

```
System.out.println(clock1.min);
clock1.min = 13;       //only legal, if min is not private
```

You can invoke a method of an object using the dot notation:

```
clock1.passtime(40);
```


Summary: declaration of methods

A method is declared with a name, parameters, a result type, and a body.

Special methods: constructors (have same name as the class, no result type is stated)

Summary: invocation of methods

General format for invocation of methods defined in the same class:

method-name (*arguments*)

Example (from `Methods1`**):** `seventimes(4)`

General format for invocation of static methods defined in another class:

class-name.method-name (*arguments*)

Example: `Keyboard.readInt()`

General format for invocation of methods of an object

object-name.method-name (*arguments*)

Example (from `TestofTime`**):** `clock1.passtime(40)`