Introductory Programming
Object oriented programming II; sections 5.0, 5.1, 5.4, 5.5

Anne Haxthausen, IMM, DTU

1. References (section 5.0)
   - the **null** reference
   - the **this** reference
   - aliases
     (a) in assignment statements
     (b) when invoking methods (with objects as parameters)
     (c) important implications of aliases
     (d) equality of references versus equality of objects
2. The **static** modifier (section 5.1)
3. Nested classes (read on your own: section 5.4)
4. Interfaces (section 5.5)

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. Translated into English by Morten P. Lindegaard.

The Time example revisited

```
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;  }
}
```

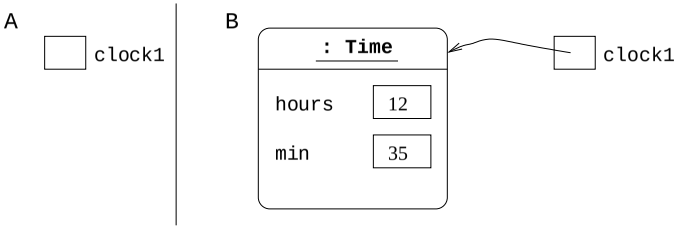

The null reference

A variable of a class type contains a reference to (i.e. the *address* of) an object of the given class.

Before the variable is initialized, it will contain the special reference value **null** which denotes that the variable does not (yet) refer to an object.

Example

```
Time clock1;                //A: clock1 == null after declaration
clock1 = new Time(12, 35); //B: clock1 != null after initializatior
```



The this reference

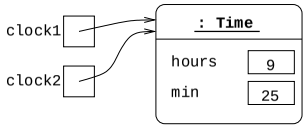
```
class Time {
  private int hours, min; //min #1

  ...

  public void passtime(int min) { //min #2  (shadows min #1)
    int totalmin = 60 * this.hours + this.min + min;
    this.hours = (totalmin / 60) % 24; this.min = totalmin % 60;  }
  }
}
```

Object aliases in assigment

Two variables may refer to the same object – they are said to be *aliases*.

Example

```
Time clock1 = new Time(9, 25);
Time clock2 = clock1;
```



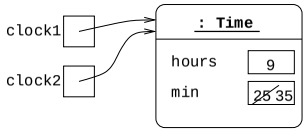
Important implications of aliases

Watch out!

If you use aliases, a change in the state of an object is a change for *all* the references that refer to the object.

Example

```
Time clock1 = new Time(9, 25);
Time clock2 = clock1;
clock1.passtime(10);
```



Equality of references versus equality of objects

Equality of references: `==`

```
clock1 == clock2
```

decides whether `clock1` and `clock2` refer to the same object – i.e. are aliases

Equality of objects: In the `Time` class, we can define our own equality as a method named `equals`:

```
boolean equals(Time clock2) {
  return
    this.min == clock2.min && this.hours == clock2.hours;
}
```

Two objects may then be compared as follows:

```
clock1.equals(clock2)
```


Equality: examples

```
public class TimeAliasing {
  public static void main(String[] args) {
    Time clock1, clock2;
    clock1 = new Time(9, 10);  clock2 = new Time(9, 10);
    printTimes(clock1, clock2);
    clock1.passtime(40);  printTimes(clock1, clock2);
    clock2 = clock1;  printTimes(clock1, clock2);
    clock1.passtime(40);  printTimes(clock1, clock2);
  }
  static void printTimes(Time clock1, Time clock2) {
    System.out.println("clock1 shows " + clock1);
    System.out.println("clock2 shows " + clock2);
    System.out.println("clock1 == clock2: " + (clock1 == clock2));
    System.out.println("clock1.equals(clock2): "
                       + clock1.equals(clock2) +"\n");
} }
```

Output from TimeAliasing

```
clock1 shows 9.10
clock2 shows 9.10
clock1 == clock2: false
clock1.equals(clock2): true

clock1 shows 9.50
clock2 shows 9.10
clock1 == clock2: false
clock1.equals(clock2): false

clock1 shows 9.50
clock2 shows 9.50
clock1 == clock2: true
clock1.equals(clock2): true

clock1 shows 10.30
clock2 shows 10.30
clock1 == clock2: true
clock1.equals(clock2): true
```


What happens to variables that are used as actual parameters for a method?

Java uses *call-by-value* when values are passed to a method – amounts to copying the value of the *actual* parameter to the *formal* parameter in the method. After a call of the method, the actual parameter (here: variable) has the same value as before the call.

Meaning:

- when a variable of a class type is an argument of a method, the *reference* to the object is copied – not the object itself
- if the method body changes the formal parameter reference (makes it refer to another object), it does *not* influence the argument (see example1)
- if the method body changes the state of the object that the formal parameter refers to, the same happens to the argument (see example2)

Primitive values as arguments for methods: example

What is the output of the following program?

```
public class CallOfMethod {
  public static void main(String[] args) {
    int a = 2;
    System.out.println("a is " + a);
    f(a);
    System.out.println("a is " + a);
  }

  static void f(int x) { x = x + 10; }
}
```


Objects as arguments for methods: example1

What is the output of the following program?

```
public class CallOfMethod1 {
  public static void main(String[] args) {
    Time a = new Time(9, 25);
    System.out.println("a is " + a);
    f(a);
    System.out.println("a is " + a);
  }

  static void f(Time x) { x = new Time(8, 17); }
}
```

Objects as arguments for methods: example2

What is the output of the following program?

```
public class CallOfMethod2 {
  public static void main(String[] args) {
    Time a = new Time(9, 25);
    System.out.println("a is " + a);
    f(a);
    System.out.println("a is " + a);
  }

  static void f(Time x) { x.passtime(10); }
}
```


Static fields

We have already seen that *methods* may be declared as **static** so that they can be used via the name of the class without creating an object.

The same goes for the *fields* in a class. Then all objects that are instances of the class will have a common location in the memory for the field; not individual copies.

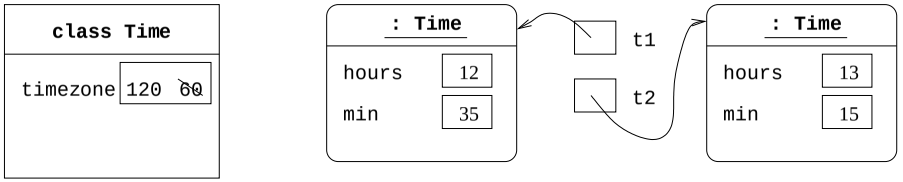
Example

A class field `timezone` could be used for the time zone common to all points in time.

```
class Time {
  static int timezone = 60; //minutes east of Greenwich
  ... everything else as before ...
}
```


Static fields, example continued

```
Time t1 = new Time(12, 35);
Time t2 = new Time(13, 15);

Time.timezone = 120;
```



Interfaces

So far, we have used the concept of *interface* for the **public** constants and methods of an object. This is consistent with the more formal concept of *interface* in Java.

Definition: A Java *interface* consists of:

- a collection of public constants and/or
- a series of public *abstract* methods, i.e. methods without bodies

Notice: no constructor, no variable fields.

Definition: A class *implements* an interface by giving method implementations (bodies) for *all* the abstract methods of the interface and possibly additional fields, methods, and constructors.

Notice: an interface may have several implementations and a class may implement several interfaces.

Interfaces: example

```
interface TimeInterface {

   public int gethours();

   public int getmin();

   public String toString();

   public void passtime(int m);
}
```


Implementation of interfaces: example 1

```
class Time1 implements TimeInterface {
  private int hours, min; //hours and minutes since midnight

  public Time1(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;
  }

}
```


Implementation of interfaces: example 2

```
class Time2 implements TimeInterface {
  private int min; //minutes since midnight

  public Time2(int h, int m) {min = (h * 60 + m) % 1440;}

  public int gethours() {return ... ;}

  public int getmin() { return ... ; }

  public String toString() { return ... ; }

  public void passtime(int m) { min = ... ;  }
}
```


What are interfaces used for (1)?

An interface specifies public methods and constants that an implementing class (or classes) must offer (as a minimum).

In software development, interfaces may be used for the *WHAT before HOW principle*:

1. First an interface is made: it is a specification of WHAT a class must offer.
2. Afterwards, classes that implement the interface are made: they define HOW data and algorithms are.

The interface may be used as a contract between the people writing the class and the people using the class.

What are interfaces used for (2)?

The **private** modifier may be used in classes for *"information hiding"*: only public methods and constants are exported. By declaring the variable fields in a class as **private**, it is possible to replace them and likewise replace method bodies without needing to change method calls.

Interfaces are also used for *"information hiding"* with similar effect. But if you regret an implementation, i.e. `Time1`, you do not need to change it – just make a new implementation, i.e. `Time2`, and replace the use of `Time1` with `Time2`.