How objects are defined?

Class-based languages

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

Object-based languages

- JavaScript, Self
- an object is defined and created without a class

Classes

A Java class

```
public class TimerClass {
   // private instance variables
   private int time; // in seconds, invariant: 0 <= time <= 3600

   // public instance methods
   public boolean isRunning() { // "this" is the target object of the method
       return this.time > 0;
   }
   public int getTime() { // "this" is the target object of the method
      return this.time;
   }
   public void tick() { // "this" is the target object of the method
      if (this.time > 0)
         this.time--;
   }
   public int reset(int minutes) { // "this" is the target object of the method
      if (minutes < 0 || minutes > 60)
         throw new IllegalArgumentException();
      int prevTime = this.time;
      this.time = minutes * 60;
      return prevTime;
   }
}
```

Classes

Classes and objects in a nutshell

- A class provides an implementation for objects of the same type
- Objects can be dynamically created from classes
- Objects created from class *C*, are called instances of *C*
- At runtime the number of instances may either increase, or decrease
- Instances are deallocated manually (C++) or automatically (garbage collection in Java, C#, JavaScript, Python, . . .)
- All instances of the same class share the same instance methods
- All instances of the same class have their own data (=internal state): instance variables are **not** shared!
- Objects created from class *C* have dynamic type *C*
- In statically typed language a class defines also a static type

Classes

Use of keyword `this` in instance methods

```
public int reset(int minutes) {
    if (minutes < 0 || minutes > 60)
        throw new IllegalArgumentException();
    int prevTime = this.time; // the "time" field of "this" is read
    this.time = minutes * 60; // the "time" field of "this" is updated
    return prevTime;
}
```

- **`this`** is the target object on which method `reset` is called
- equivalent terminology: **`this`** is the receiver of the message `reset`

Classes

Use of `TimerClass`

```
// creates a new object of type TimerClass and assigns its identity to t1
TimerClass t1 = new TimerClass();
// calls instance method reset on the object in t1 with argument 1
t1.reset(1);
// creates a new object of type TimerClass and assigns its identity to t2
TimerClass t2 = new TimerClass();
// calls instance method reset on the object in t1 with argument 2
t2.reset(2);
```

Remarks

- `t1` and `t2` have static type `TimerClass`: they can only be assigned expressions of static type compatible with `TimerClass`
- object creation
  - **Syntax**: `'new'CID '('(Exp ( ','Exp)*)? ')'`
  - **Semantics**: a new instance of class `CID` is dynamically created, arguments are used to properly initialize the fields of the newly created object

Fields

Field access

```
public int reset(int minutes) {
    if (minutes < 0 || minutes > 60)
       throw new IllegalArgumentException();
    int prevTime = this.time; // the "time" field of "this" is read
    this.time = minutes * 60; // the "time" field of "this" is updated
    return prevTime;
}
```

Remarks

- Syntax of field read: `Exp '.'FID`
- Static semantics of expression $e.f$
  - the static type of e must be a class C with accessible field f
  - if so, then the static type of $e.f$ is the type of field f in C
- Syntax of field update: `Exp '.'FID = Exp`
- Static semantics of statement $e_1.f = e_2$
  - the static type of e_1 must be a class C with accessible field f
  - the static type of e_2 must be *compatible* with the type of field f in C

Information hiding

Access modifiers

- **`private`** field/method: declaration is only visible within the class
- **`public`** field/method: declaration is visible outside the class (more precise definition later on)
- **`public class`**: the class declaration is visible everywhere in the program

Remarks

- **`private`** and **`public`** are not the only access modifiers in Java
- in many other OOL there are similar access modifiers
- in some OOL (as Smalltalk) fields are private on object basis: methods cannot access the fields of the other objects, even when they have the same dynamic type

A quick introduction to exceptions in Java

The `throw` statement

- when an error must be notified, an exception is thrown
- Syntax: `'throw'Exp`
- Static semantics of **`throw`** *e*: the static type of *e* must be an exception
- exceptions are special objects ⇒ exception types are special classes

Example

```
Throwable ex;
...
throw 42; // type error, int is not an exception type!
throw new NullPointerException(); // OK
throw ex; // OK
```

Assertions

An important feature for documenting and testing code

- Syntax (simplest case): `'assert'Exp`
- Static semantics: `Exp` must have static type **`boolean`** or `Boolean`
- Dynamic semantics: if assertions are enabled, then
  - `Exp` is evaluated in a boolean value *b*
  - if *b* is **`true`** then no further action is taken, else an exception of type `AssertionError` is thrown

Example

```
TimerClass t1 = new TimerClass(); // what is the initial time of t1?
t1.reset(1); // 1 minute, that is, 60 seconds
int seconds = 0;
while (t1.isRunning()) {
   t1.tick(); // one second per tick
   seconds++;
}
// these assertions are expected to hold!
assert seconds == 60;
assert !t1.isRunning();
```

Object values (1)

Objects as first-class values

In object-oriented languages objects are first-class values

- they can be assigned to variables
- passed as arguments to methods
- returned as values by methods

Object identity

Objects are represented by their identity
More in concrete:

identity = reference = address where the object is stored on the heap

Object values (2)

Assignment and argument passing

In most OOL (Java included) objects are assigned or passed by reference

Example

```
TimerClass t1 = new TimerClass();
TimerClass t2 = t1; // t2 and t1 refer to the same object
TimerClass t3 = null; // t3 refers to no object
assert t1 == t2 && t1 != t3;
```

Remarks

- `t2` contains the same object identity (that is, reference) as `t1`
- `==` tests whether two expressions evaluate to the same object identity
- **`null`**: useful constant value to denote “no object”
- `t3` refers to no object

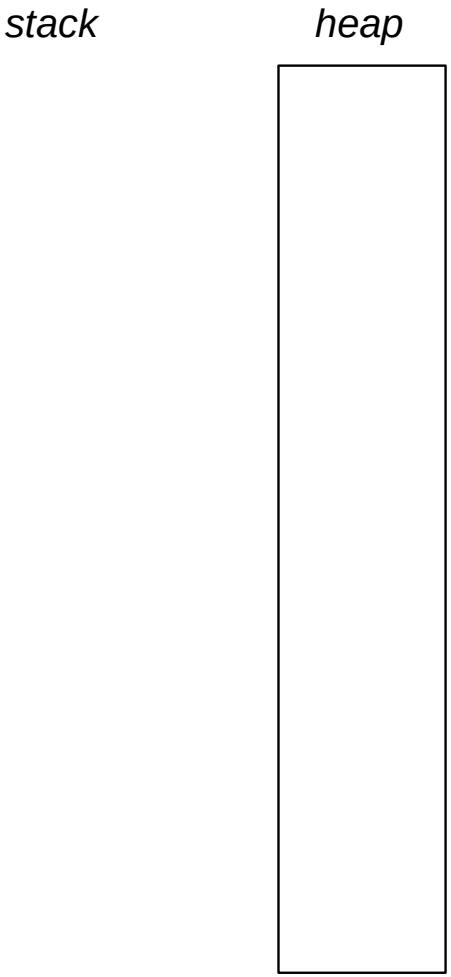
Classes

Another type of timer

```
public class AnotherTimerClass {
   private int seconds;// invariant: 0<=seconds<=59 && minutes==60 => seconds==0
   private int minutes;// invariant: 0<=minutes<=60
   public boolean isRunning() {
      ...
   }
   public int getTime() {
      ...
   }
   public void tick() {
      ...
   }
   public int reset(int minutes) {
      ...
   }
}
```

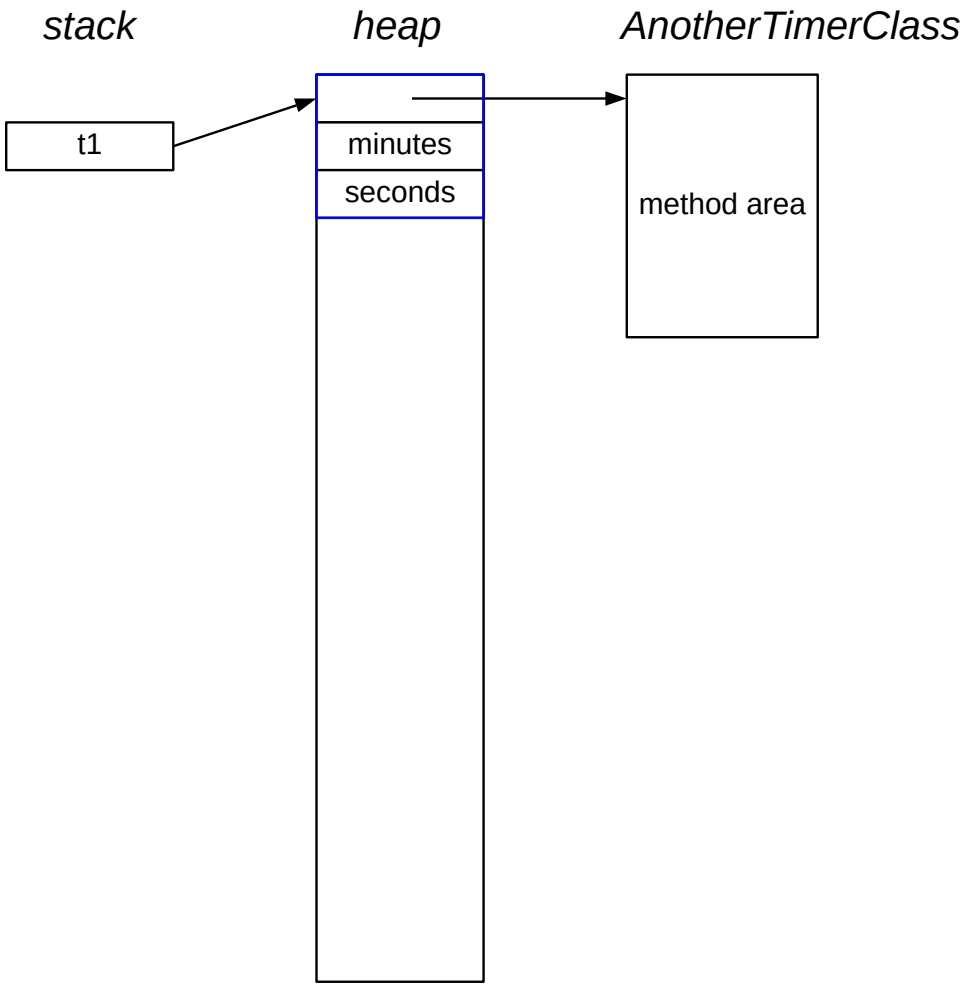
Example with memory model

```
AnotherTimerClass t1 = new AnotherTimerClass();
AnotherTimerClass t2 = new AnotherTimerClass();
TimerClass t3 = new TimerClass();
TimerClass t4 = t3;
```



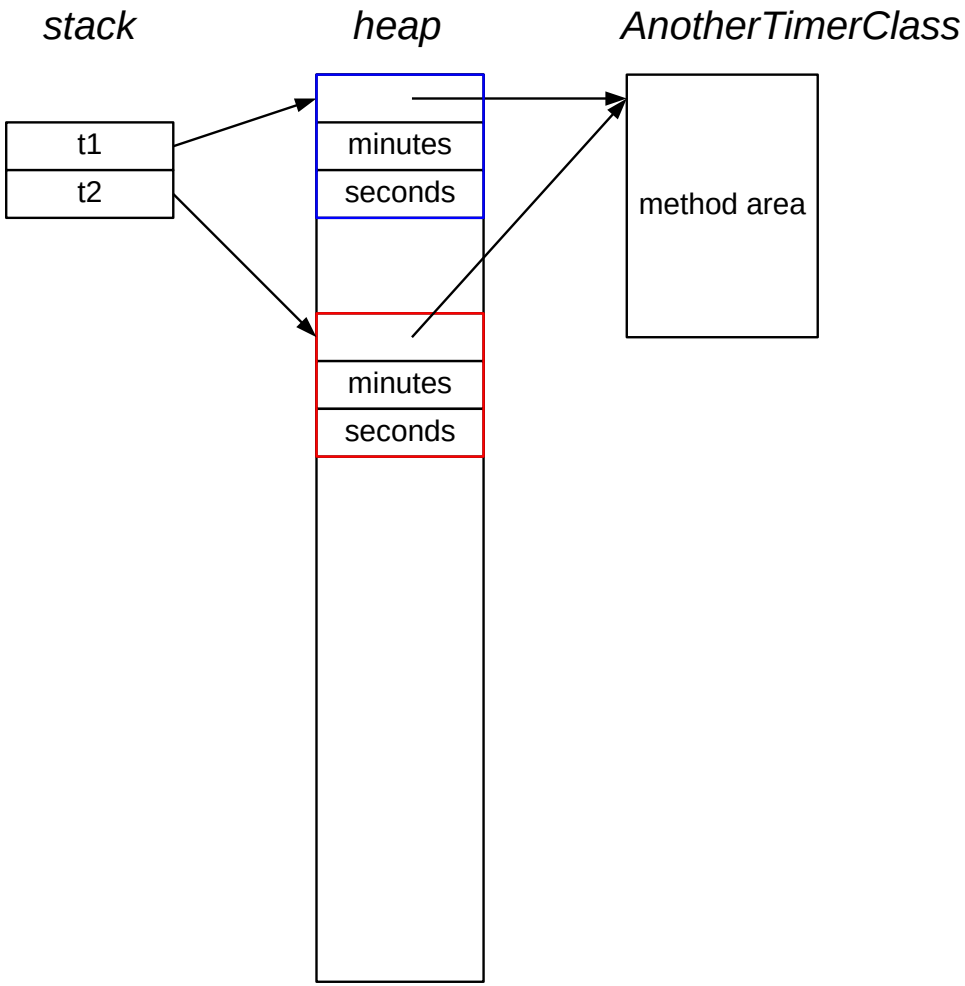
Example with memory model

```
AnotherTimerClass t1 = new AnotherTimerClass(); ⇐
AnotherTimerClass t2 = new AnotherTimerClass();
TimerClass t3 = new TimerClass();
TimerClass t4 = t3;
```



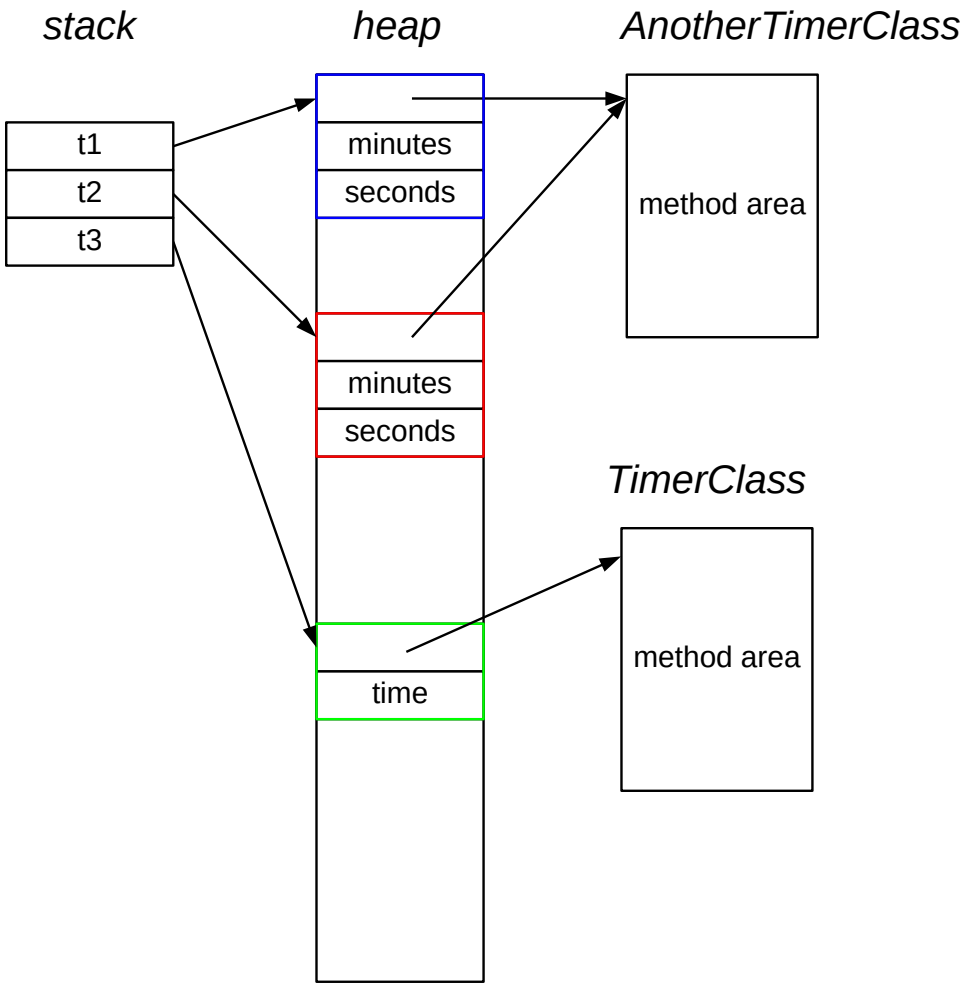
Example with memory model

```
AnotherTimerClass t1 = new AnotherTimerClass();
AnotherTimerClass t2 = new AnotherTimerClass(); ⇐
TimerClass t3 = new TimerClass();
TimerClass t4 = t3;
```



Example with memory model

```
AnotherTimerClass t1 = new AnotherTimerClass();
AnotherTimerClass t2 = new AnotherTimerClass();
TimerClass t3 = new TimerClass(); ⇐
TimerClass t4 = t3;
```



Example with memory model

```
AnotherTimerClass t1 = new AnotherTimerClass();
AnotherTimerClass t2 = new AnotherTimerClass();
TimerClass t3 = new TimerClass();
TimerClass t4 = t3;
```