Reading fields of objects

Example

```
public int getTime() {
   return this.time; // field 'time' of 'this' is read
}
```

Rules

- Syntax: `Exp '.'FID`
- Static semantics of $e.f$
  - the static type of e must be a class C defining field f, f must be visible
  - the static type of $e.f$ is the declared type of field f in C

Updating fields of objects

Example

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

Rules

- Syntax for updating a field: `Exp '.'FID = Exp`
- Static semantics of $e_1 . f = e_2$
  - the static type of e_1 must be a class C declaring field f, f must be visible
  - the static type of e_2 must be compatible with the declared type of field f in C

Information hiding

Access modifiers `private`/`public`

- `private` fields, methods: visible only within the class
- `public` fields, methods: visible also outside the class
- `public class`: visible everywhere in the program

Remarks

- in Java visibility of fields and methods is on class basis
- this is true for most object-oriented languages
- in some languages as Smalltalk, visibility of fields is on object basis

A quick introduction to exceptions in Java

Statements to manage exceptions

- to generate an exception the **`throw`** statement is used
  - Syntax: `'throw' Exp`
  - Static semantics of **`throw`** *e*: *e* must have an exception type
- exceptions are special objects
- exceptions are handled with the **`try-catch`** statement

A quick introduction to exceptions in Java

Example

```
Throwable ex;
...
throw new NullPointerException();  // correct
throw ex;                          // correct
throw 42;                          // type error! integers are not exceptions
...
try{
  readFile(fname);
}
catch(IOException e){
  readFile(defaultFile);
}
catch (Throwable e) {
   e.printStackTrace();
   error("Unexpected error.");
}
```

- `Throwable`, `NullPointerException`, `IOException` are predefined
- users can define their own exceptions with special classes

Assertions

An important feature for documenting and testing code

- Syntax: `'assert'Exp`
- Static semantics of **`assert`** *e* : *e* must have a boolean type
- Dynamic semantics of **`assert`** *e*: if the assertion is enabled, then
  - the value *v* of *e* is computed
  - if *v* is **`true`** then no further action is taken, else an exception of type `AssertionError` is thrown

  if the assertion is not enabled, then the assertion is not executed

Enabling assertions

- to enable assertions use the option `-ea`: `java -ea TimerClass`
- Remark: not supported by Java visualizer

Assertions

Demo

```
TimerClass t1 = new TimerClass();
t1.reset(1);              // t1 reset to 1 minute
int seconds = 0;
while (t1.isRunning()) {
   t1.tick();             // one second per tick
   seconds++;
}
assert seconds == 60;     // expected to hold
assert !t1.isRunning();   // expected to hold
```

Objects

Objects as references

- in Java, as in most languages, objects are references

  reference = address where the object is stored on the heap

- all these operations are by reference when objects are involved
  - assignment to variables
  - argument passing
  - return of values from method calls

Objects

Demo

```
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;
assert t3.isRunning();          // NullPointerException!
```

Remarks

- `t2` contains the same object reference as `t1`
- `==` tests whether two expressions refer to the same object
- **`null`**: predefined constant which means no object
- `t3` contains **`null`**: it refers to no object
- accessing fields or calling methods on **`null`** throws `NullPointerException`

Classes

Another type of timer

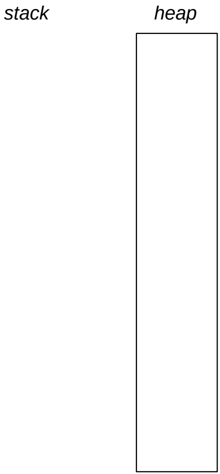
```
public class AnotherTimerClass {
   // total time (in seconds) equals seconds+60*minutes
   private int seconds;// invariant: 0<=seconds<=59
   private int minutes;// invariant: 0<=minutes<=60 && (minutes<60 || seconds==0)

   public boolean isRunning() { ... }
   public int getTime() { ... }
   public void tick() { ... }
   public int reset(int minutes) { ... }
}
```

Remarks

- no explicit relationship between `TimerClass` and `AnotherTimerClass`
- the two types are not compatible in Java, although they both provide implementations of timer objects which have the same interface

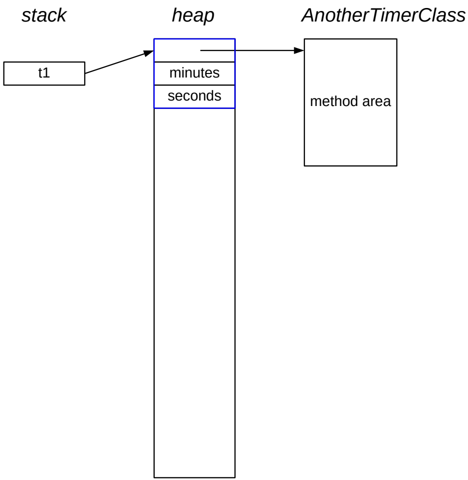
Demo

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



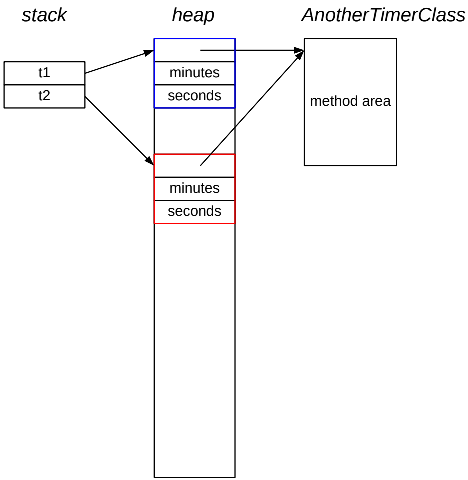
Demo

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



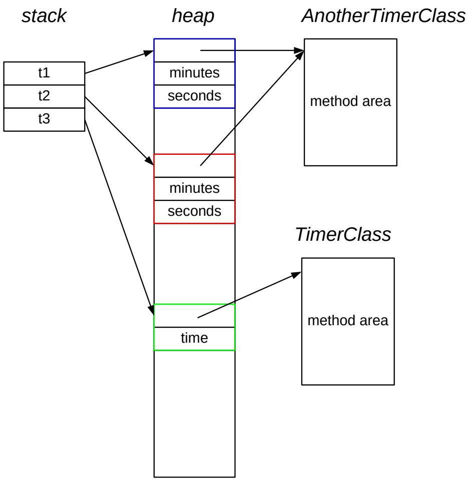
Demo

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



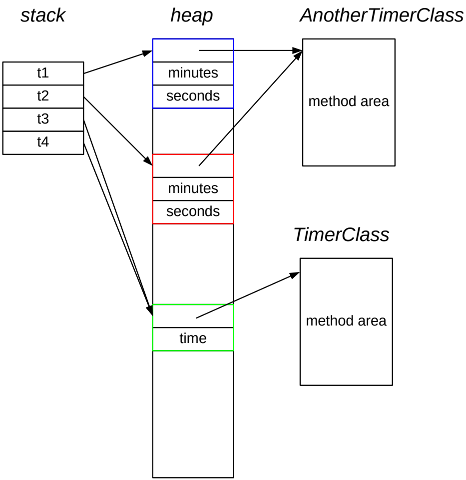
Demo

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



Demo

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



Classes and types

Reference types

- classes define types that can be used in programs
- terminology: a class is a reference type
- example: `TimerClass`, `AnotherTimerClass` are reference types

Meaning of reference types

A reference type *C* can contain

- references to objects of class *C*
- or **`null`**

Classes and types

Java is statically typed

- a static semantics with typing rules is defined
- types are verified at compile time

Example: variable assignment $x{=}e$
Rule: the (static) type of e must be compatible with the declared type of x

Demo

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

These assignments are not allowed

```
t1 = t2; // AnotherTimerClass not compatible with TimerClass
t2 = t1; // TimerClass not compatible with AnotherTimerClass
```

These assignments are allowed

```
t1 = null; // the type of null is compatible with all reference types
t2 = null; // the type of null is compatible with all reference types
```

Classes and types

Dynamic types

- dynamic types can be checked with the predefined operator **`instanceof`**
- Syntax: *e* **`instanceof`** *C*
- Dynamic semantics: is the value of *e* a reference to an object of a class compatible with *C*?

Demo

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

assert t1 instanceof TimerClass;
assert t2 instanceof AnotherTimerClass;
assert !(null instanceof TimerClass);        // null does not refer to any object
assert !(null instanceof AnotherTimerClass); // null does not refer to any object
```

Design by contract

Motivations

- formal specification of classes and object interfaces to guarantee correctness of software
- two parties are involved in the contract:
  - the developers of a class *C*
  - the clients of *C* (=programmers that use *C* in their programs)
- the contract in a nutshell: if clients use a class *C* correctly then
  - method calls on *C* objects behave in accordance with their specification
  - the state of any *C* object is always valid after
    - its creation
    - any method call on it

Formalization

- use a class correctly = pre-conditions
- method calls behave in accordance = post-conditions
- the state of any *C* object is always valid = invariants

Design by contract

Code contracts: pre-conditions, post-conditions, invariants

- pre-condition for method *m*, defined by a predicate *p*
  `requires` *p*: *p* must hold immediately before the execution of *m*
- post-condition for method *m*, defined by a predicate *p*
  `ensures` *p*: if the pre-condition of *m* holds, then *p* must hold immediately after the execution of *m*
- invariant for class *C*, defined by a predicate *p*
  `invariant` *p*: *p* must hold
  - immediately after creation of each instance of *C*;
  - immediately before the execution of each instance method of *C*;
  - immediately after the execution of each instance method of *C*, if the pre-condition of the method holds.

Design by contract

Syntactic entities

- standard logical connectives and quantifiers
- pre-conditions: the parameters of methods, `this` and object fields
- post-conditions: the parameters and the result of methods, `this` and the old (before the call) and new (after the call) values of the object fields
- invariants: object fields

Remarks

- Java does not offer native support for design by contract
- other languages do (Eiffel)
- pre/post-conditions and invariants in Java comments are useful

Design by contract

Example

```
public class TimerClass {
    private int time;
    /* invariant 0 <= time && time <= 3600; */
   public int getTime() {
    /* ensures result == this.time && this.time == old(this.time); */
      return this.time;
   }
    public boolean isRunning() {
    /* ensures result == this.time > 0 && this.time == old(this.time); */
      return this.getTime() > 0;
    }
    public int reset(int minutes) {
    /*  requires 0 <= minutes && minutes <= 60;
        ensures  result == old(this.time)
                 && this.time == minutes * 60; */
      if (minutes < 0 || minutes > 60)
         throw new IllegalArgumentException();
      int prevTime = this.getTime();
      this.time = minutes * 60;
      return prevTime;
    }
    ...
}
```

Class invariants

How class invariants can be ensured to hold?

- information hiding: object states changed in a controlled way only with methods, no arbitrary changes allowed!
- all methods preserve invariants
- initially, the invariant must be verified by construction
- constructors are used for initializing objects correctly, while guaranteeing information hiding

Constructors

Example of definition

`TimerClass` with three different constructors:

```
public class TimerClass {
    private int time; // 'time' in seconds

    public TimerClass() { // initializes 'time' with the default value 0
    }
    public TimerClass(int minutes) {
        if (minutes < 0 || minutes > 60)
            throw new IllegalArgumentException();
        this.time = minutes * 60;
    }
    public TimerClass(TimerClass otherTimer) { // copy constructor
        this.time = otherTimer.getTime();
    }
... // methods as before
}
```

Remark

In Java constructors can be overloaded

Constructors

Example of use of constructors

```
TimerClass t1 = new TimerClass();
TimerClass t2 = new TimerClass(42);
TimerClass t3 = new TimerClass(t2);
assert t1.getTime()==0 && t2.getTime()==42*60 &&
       t2.getTime()==t3.getTime();
```

Object creation and initialization in Java

Object fields can be also initialized with field initializers:

Example of variable initializer

```
public class TimerClass {
   private int time = 60; // 'time' in seconds, default is 1 minute

    public TimerClass() { // initializes 'time' with the default value 60
    }
    public TimerClass(int minutes) {
        if (minutes < 0 || minutes > 60)
            throw new IllegalArgumentException();
        this.time = minutes * 60;
    }
    public TimerClass(TimerClass otherTimer) { // copy constructor
        this.time = otherTimer.getTime();
    }
...
}
```

Object creation and initialization in Java

Simplified rules

1. immediately after object creation a default value is assigned to each field of the object
2. the default value depends on the type of the field:
   - 0 for **`int`** and other numerical types
   - **`false`** for **`boolean`**
   - **`null`** for reference types
3. if any, field initializers are executed in the left-to-right top-to-bottom textual order
4. the constructor of the class matching the number and types of parameters is called

Object creation and initialization in Java

Example

```
TimerClass timer1 = new TimerClass();
TimerClass timer2 = new TimerClass(1);
assert timer1.getTime() == timer2.getTime();
```

Object creation and initialization in Java

Overloaded constructors

- multiple constructors can be defined: they are overloaded
- they must differ either in the number or in the type of the parameters

Default constructor

- implicitly defined only if no constructor is provided
- it has no parameters and the empty body

Explicit constructor call

Example

```
public class Person {
    private String name; // 'name' is not optional
    /* invariant name != null */
    private String address; // 'address' is optional, can contain null

    public Person(String name) {
        if (name == null) // 'name' cannot be undefined
            throw new NullPointerException();
        this.name = name;
    }
    public Person(String name, String address) {
        this(name); // calls the constructor with a single argument
        this.address = address;
    }
    public String getName() { // getter method
        return this.name;
    }
    public String getAddress() { // getter method
        return this.address;
    }
}
```

Explicit constructor call

Rules

- a constructor may be explicitly called in another constructor
- syntax: `'this' '(' (Exp ( ',' Exp)*)? ')'`
- explicit call allowed only on the first line
- cyclic explicit constructor calls not allowed

Java convention for constructor parameters

```
this.address = address;
```

- **`this`**`.address`: field `address` of the object **`this`** that needs to be initialized
- `address`: the constructor parameter which contains the value to be assigned to field `address`

Analogously for

```
this.name = name;
```

Explicit constructor call

Example

```
Person sam = new Person("Samuele");
Person sim = new Person("Simone","Genova");
assert sam.getAddress()==null && sim.getAddress()!=null;
```

Remarks

Object fields in statically typed languages

- object fields cannot be added or removed dynamically
- non optional fields must always contain a well-defined value
- for an optional field of reference type **`null`** is the standard choice for "no value"
- for an optional field of primitive type (**`int`**, **`boolean`**, ...) there is no standard choice for "no value"
- **`null`** not compatible with primitive types (**`int`**, **`boolean`**, ...)

Strings in Java

- `String` is a predefined class
- strings are immutable objects
- string literals have a standard syntax