Initialization of class variables

Rules

- a class variable is initialized after its class has been loaded and linked
- a default value is assigned, as happens for instance variables
- class variable and static initializers are executed in textual order

Warning

Do not use constructors to initialize class variables!

Example

```
class Test {
   private static int max = 5; // class variable initializer
   private static int fact; // the default value would be 0
   static { // static initializer for Test.fact, computes factorial of Test.max
      int f=1;
      for (int i=1;i<=Test.max;i++) f*=i;
      Test.fact=f;
   }
   ...
   assert Test.fact==120;
}
```

Class methods

Instance versus class methods

- instance methods = invoked on an object (represented by **`this`**)
- class methods = invoked on a class (no object!)

Remark

`this` is undefined in the body of a class method

Java syntax and terminology

- Syntax: `CID '.' MID '(' (Exp ( ',' Exp)*)? ')'`
- Terminology: *class method*, or *static method*

Class methods

Example 1

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

    // class method for argument validation
    private static void checkMinutes(int minutes) {
        if (minutes < 0 || minutes > 60)
            throw new IllegalArgumentException();
    }
    public TimerClass(int minutes) {
        TimerClass.checkMinutes(minutes);
        this.time = minutes * 60;
    }
    public int reset(int minutes) {
        TimerClass.checkMinutes(minutes);
        int prevTime = this.time;
        this.time = minutes * 60;
        return prevTime;
    }
    ...
}
```

Class methods

Example 2

```
public class Item {
    private static long nextSN; // next unused serial number
    private int price; // in cents
    private long serialNumber;

    private static long getNextSN() {
        if (Item.nextSN < 0)
            throw new RuntimeException("No more serial numbers!");
        return Item.nextSN++;
    }
    public Item(int price) {
        if (price < 0)
            throw new IllegalArgumentException();
        this.price = price;
        this.serialNumber = Item.getNextSN();
    }
    public int getPrice() {
        return this.price;
    }
    public long getSerialNumber() {
        return this.serialNumber;
    }
}
```

Class methods

Example 3

```
public class Rectangle {
    private static int defaultSize = 1; // class variable initializer
    private int width = Rectangle.defaultSize;
    private int height = Rectangle.defaultSize;
    // invariant width > 0 && height > 0
    private static void checkSize(int size) { // static validation method
        if (size <= 0)
            throw new IllegalArgumentException();
    }
    public Rectangle(int width, int height) {
        Rectangle.checkSize(width);
        Rectangle.checkSize(height);
        this.width = width;
        this.height = height;
    }
    // static factory method
    public static Rectangle ofWidthHeight(int width,int height) {
        return new Rectangle(width, height);
    }
    ...
}
```

Class methods

Example 3

```
// can we distinguish width and height?
Rectangle r1 = new Rectangle(3, 5);
Rectangle r2 = Rectangle.ofWidthHeight(3, 5);
```

Class `Object`

In a nutshell

- `Object` is a very special predefined class
- any other object type is a subtype of `Object`
- if an expression has static type `Object`, then it can evaluate into
  - either an instance of a subclass of `Object` (that is, any class)
  - or **`null`**
  - or an array (more details later on)

Subtyping in OOL

In a nutshell

- a relationship between types
- a hierarchical classification (= taxonomy) of the types of values

Examples

- `TimerClass` is a subtype of `Object` (written `TimerClass` $\leq$ `Object`)
- intuition:

  a timer is necessarily an object

  but an object is not necessarily a timer

- `Rectangle` is a subtype of `Shape` (written `Rectangle` $\leq$ `Shape`)
- intuition:

  a rectangle is necessarily a shape

  but a shape is not necessarily a rectangle

Subtyping in OOL

Basic subtyping rules

- any object type is a subtype of `Object`
- object and primitive types are not comparable

Examples

- `TimerClass` $\leq$ `Object`
- `Person` $\leq$ `Object`
- `String` $\leq$ `Object`

Examples

- **`int`** $\not\leq$ `Object` and `Object` $\not\leq$ **`int`**
- **`boolean`** $\not\leq$ `Object` and `Object` $\not\leq$ **`boolean`**
- **`int`** $\not\leq$ `Person` and `Person` $\not\leq$ **`int`**
- **`boolean`** $\not\leq$ `Person` and `Person` $\not\leq$ **`boolean`**

Subtyping in OOL

It is a partial order

- reflexivity: $T \leq T$
- antisymmetry: $T_1 \leq T_2$ and $T_2 \leq T_1$ implies $T_1 = T_2$
- transitivity: $T_1 \leq T_2$ and $T_2 \leq T_3$ implies $T_1 \leq T_3$

Subtyping is not a **total** order on object types!

There exist object types that are not comparable

Examples

- `String` $\not\leq$ `TimerClass` and `TimerClass` $\not\leq$ `String`
- `Person` $\not\leq$ `TimerClass` and `TimerClass` $\not\leq$ `Person`
- `Person` $\not\leq$ `String` and `String` $\not\leq$ `Person`

Subtyping in OOL

Subtyping and static semantics

If a type T is expected, any subtype of T works fine!

- initialization/assignment of variables

  example: a variable of type `Shape` can be initiliazed/updated with an object of type `Rectangle`

- argument passing

  example: an object of type `Rectangle` can be passed to a parameter of type `Shape`

- returned value

  example: a method with return type `Shape` can return an object of type `Rectangle`

Object equality

Two types of equalities

- strong equality: `person1 == person2`
  `person1` and `person2` refer to the same person
- weak equality: `person1.equals(person2)`
  `person1` and `person2` refer to two persons with the same name and address, but the two persons might not be the same person

Remarks

- `person1==person2` implies `person1.equals(person2)`
- `person1.equals(person2)` does not imply `person1==person2`
- **`boolean`** `equals(Object)` is a very special Java method
  (more details later on)

Strings

Strings are **immutable** objects

```
String s1 = "a string";
String s2 = new String("a string"); // copy constructor
assert s1 != s2 && s1.equals(s2);
String s3 = "Hello " + "world"; // string concatenation operator
String s4 = "Hello ".concat("world"); // string concatenation method
assert s3 != s4 && s3.equals(s4);
```

Immutable and mutable objects

- immutable object: its instance variables cannot be changed after initialization
- mutable object: its instance variables can be changed after initialization

Remarks

- do not use `==` or `!=` for immutable objects!
- also for mutable objects be aware that `==` and `equals` behave differently

Object composition

Points

```
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Point(Point p) {
        this(p.x, p.y); // if p==null then p.x throws NullPointerException!
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
    public void move(int dx, int dy) {
        this.x += dx;
        this.y += dy;
    }
    public boolean overlaps(Point p) {
        return this.x == p.x && this.y == p.y; /* if p==null then p.x throws
            NullPointerException! */
    }
}
```

Object composition

Lines as pairs of points

```
public class Line {
    private Point a;
    private Point b;

    // invariant a != null && b != null && !a.overlaps(b)
    public Line(Point a, Point b) {
        if (a.overlaps(b)) /* if a==null||b==null then NullPointerException is
             thrown! */
            throw new IllegalArgumentException();
        this.a = a;
        this.b = b;
    }
    public void move(int dx, int dy) {
        this.a.move(dx, dy);
        this.b.move(dx, dy);
    }
    public boolean overlaps(Line l) { // "this" and "l" fully overlap
        return this.a.overlaps(l.a) && this.b.overlaps(l.b)
                || this.a.overlaps(l.b) && this.b.overlaps(l.a);
    }
}
```