Class fields

It is possible to define class fields

Example

```
public class Item {
    private static long nextSN; // class field for the next serial number
    private int price;          // object field (value is in cents)
    private long serialNumber;  // object field
    /* invariant price>=0 && serialNumber>=0 &&
       \forall Item o,o'; o.serialNumber==o'.serialNumber  ==> o==o'; */
    public Item(int price) {
        if (price < 0)
            throw new IllegalArgumentException();
        this.price = price;
        this.serialNumber = Item.nextSN++;
    }
    public int getPrice() {
        return this.price;
    }
    public long getSerialNumber() {
        return this.serialNumber;
    }
}
```

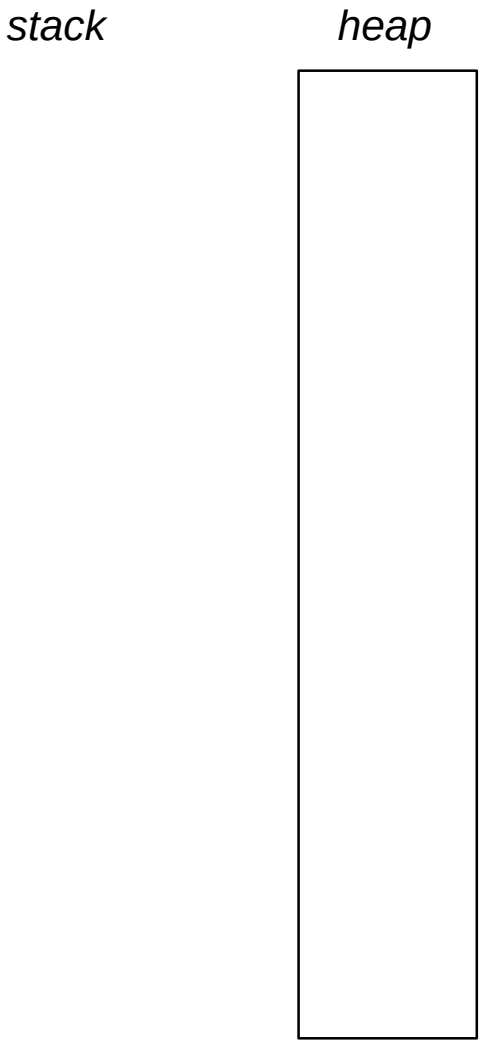
Class fields

Demo

```
Item item1 = new Item(6150);
Item item2 = new Item(1400);
assert item1.getPrice() == 6150 && item1.getSerialNumber() == 0;
assert item2.getPrice() == 1400 && item2.getSerialNumber() == 1;
```

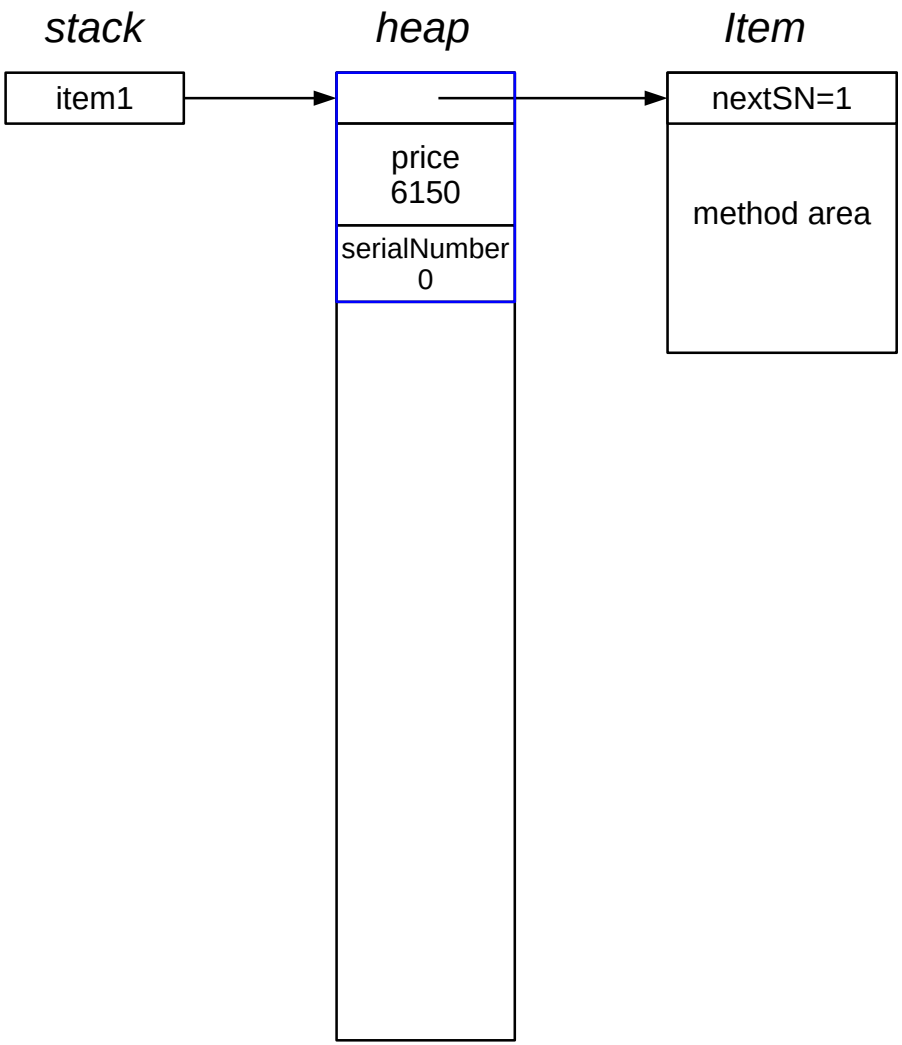
Demo

```
Item item1 = new Item(6150);
Item item2 = new Item(1400);
```



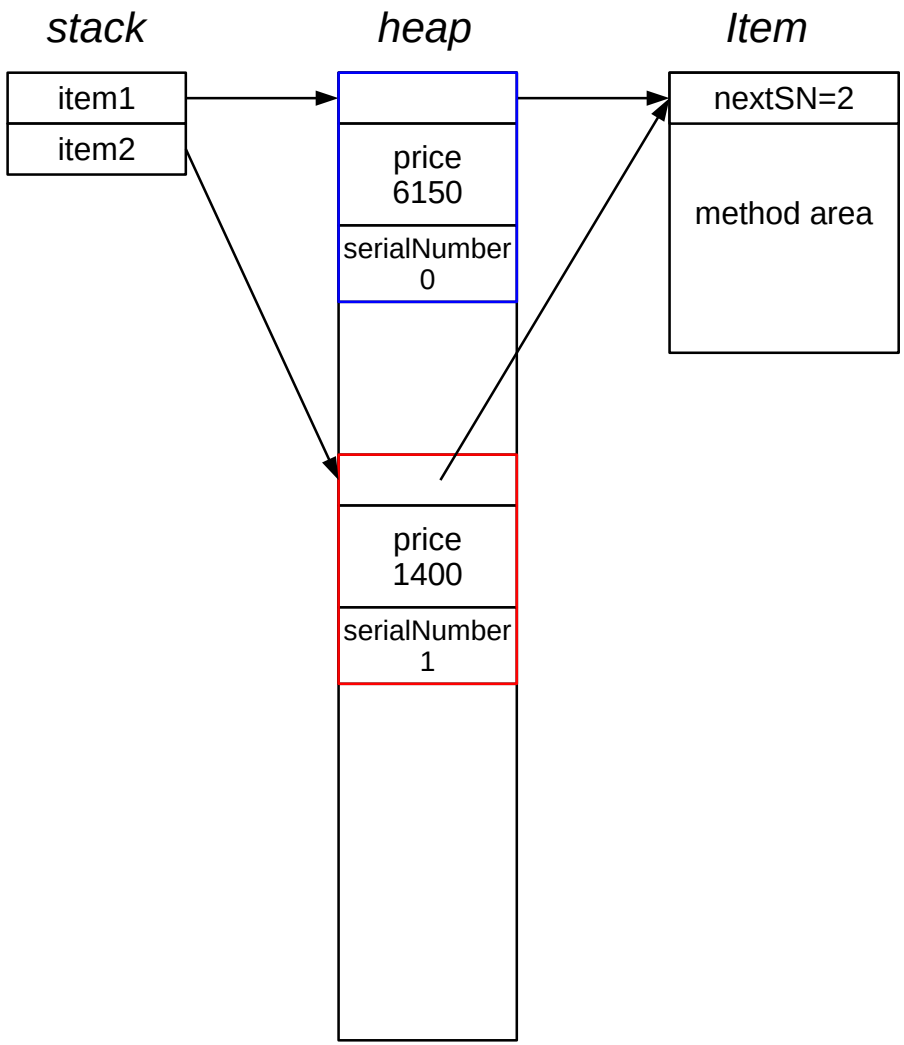
Demo

```
Item item1 = new Item(6150); ⇐
Item item2 = new Item(1400);
```



Demo

```
Item item1 = new Item(6150);
Item item2 = new Item(1400); ⇐
```



Class fields

Object versus class fields

- object fields
  - each object of `Item` has fields `price` and `serialNumber`
  - objects of `Item` do not have field `nextSN`
- class fields
  - class `Item` has field `nextSN`
  - class `Item` does not have fields `price` and `serialNumber`

Java syntax and terminology

- Syntax:
  - field read: `CID '.'FID`
  - field update: `CID '.'FID '='Exp`

  `CID` class identifier, `FID` field identifier
- Terminology: class field (or static field, or class variable, or static variable)

Initialization of class fields

Example

```
public class Test {
    private static int val = 5; // class field initializers
    private static int fact = 1;
    static { // initializes Test.fact with the factorial of Test.val
        for (int i = 1; i <= Test.val; i++)
            Test.fact *= i;
    }
    public static void main(String[] args) {
        assert Test.val==5 && Test.fact==120;
    }
}
```

Rules

- a default value is assigned to class fields, as happens for object fields
- class field and static initializers are executed in textual order

Warning

Do not use constructors to initialize class fields!

Class methods

Object versus class methods

- object methods called on an object
- class methods called on a class

Remark

`this` is undefined in the body of a class method, there is no target object

Java syntax and terminology

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

Class methods

Example 1: code refactory

```
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();
    }

    // class method for conversion to seconds
    private static int toSeconds(int minutes) {
        return minutes * 60;
    }
    public TimerClass(int minutes) {
        TimerClass.checkMinutes(minutes);
        this.time = TimerClass.toSeconds(minutes);
    }
    public int reset(int minutes) {
        TimerClass.checkMinutes(minutes);
        int prevTime = this.time;
        this.time = TimerClass.toSeconds(minutes);
        return prevTime;
    }
    ...
}
```

Class methods

Example 2: controlled access to class fields

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

    // class method for controlled modification of Item.nextSN
    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: static factory methods

```
public class Rectangle {
    private static int defaultSize = 1;        // class field 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);
    }
    ...
}
Rectangle r1 = new Rectangle(3, 5);            // which width and height?
Rectangle r2 = Rectangle.ofWidthHeight(3, 5); // width=3, height=5
```

Class `Object`

In a nutshell

- `Object` is a special predefined class
- any reference type is a subtype of `Object`
- if an expression has static type `Object`, then its value can be
  - either a reference to an instance of any class
  - or **`null`**
  - or a reference to an array (more details later on)

Subtyping

In a nutshell

- a relationship between types
- a taxonomy of types

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

Basic subtyping rules

- any reference type is a subtype of `Object`
- reference and primitive types are not comparable
- subtyping is a partial order

Examples

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

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!

- there exist reference types that are not comparable
- primitive and reference types 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`
- `Person` $\not\leq$ **`int`** and **`int`** $\not\leq$ `Person`

Subtyping

Subtyping makes typechecking more flexible

Rule: If a type T is required, then any subtype of T is correct

Static semantics rules and subtyping in a nutshell

Examples with `Rectangle` $\leq$ `Shape`

- initialization/assignment of variables of any kind

  a variable of type `Shape` can be initialized/updated with an object of type `Rectangle`, not the other way round

- argument passing

  an object of type `Rectangle` can be passed to a parameter of type `Shape`, not the other way round

- returned value

  a method with return type `Shape` can return an object of type `Rectangle`, not the other way round

Object equality

Two types of equality relations

- strong equality: `person1 == person2`
  `person1` and `person2` refer to the same object
- weak equality: `person1.equals(person2)`
  `person1` and `person2` refer to two objects where fields have the same values, but the objects may be distinct (=different references)

Remarks

- `==` and `!=` are predefined operators
- `person1==person2` implies `person1.equals(person2)`
- `person1.equals(person2)` does not imply `person1==person2`
- **`boolean`** `equals(Object)` is a predefined Java method
  - it can be called on any object
  - classes can redefine it (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: all fields cannot be changed after initialization
- mutable object: some fields can be changed after initialization

Remarks

- never use `==` or `!=` for immutable objects!
- usually `==` and `equals` behave differently also for mutable objects