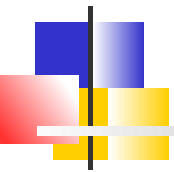
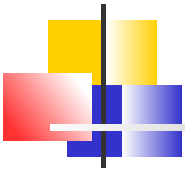
Software and Programming I

Object-Oriented Programming in Java



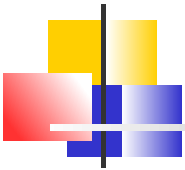
Roman Kontchakov / Carsten Fuhs

Birkbeck, University of London



Outline

- Object-Oriented Programming
- Public Interface of a Class
- Instance Variables
- Instance Methods and Constructors
  - Sections 8.1 – 8.6 (7.1 – 7.5 in 1/e)



Object-Oriented Programming

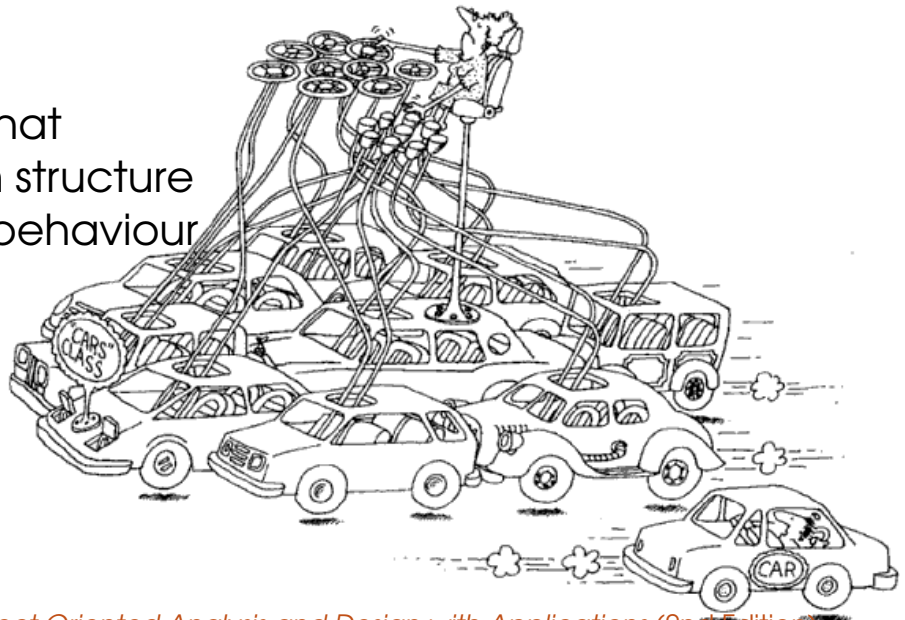
- Tasks are solved by collaborating **objects**
- Each object has its own set of **data**, together with a set of **methods** that act upon the data
- A **class** describes a set of objects with the same behaviour (i.e., methods)
- Objects are constructed with the `new` operator:

```
Scanner in = new Scanner(System.in);
```

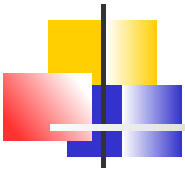
Classes and Objects

Class represents
a set of objects that
share a common structure
and a common behaviour

Objects
are instances
of classes



Booch, G.: *Object Oriented Analysis and Design with Applications* (2nd Edition) Addison-Wesley, 1994

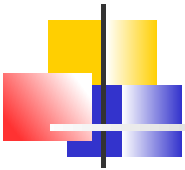


Example: Cash Register

- A cashier who rings up a sale presses a key to start the sale; then rings up each item.
  A display shows the amount owed as well as the total number of items purchased

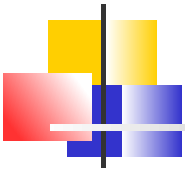
  (a use case description)

- We want the following methods on a cash register object:
  - add the price of an item
  - get the total amount owed and the count of items purchased
  - clear the cash register to start a new sale



Example: Cash Register (2)

```
1  /**
2      A simulated cash register
3  */
4  public class CashRegister  {
5      /* private data */
6      ...
7      /* methods (public interface) */
8      public void addItem(double price) { /* */ }
9      public double getTotal() { /* */ }
10     public int getCount() { /* */ }
11     public void clear() { /* */ }
12 }
```



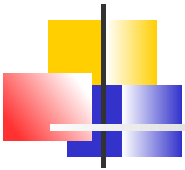
Using the CashRegister

- constructing an object (for example, in class `CashRegisterTest`)

```
CashRegister r1 = new CashRegister();
```

- invoking methods (again, in class `CashRegisterTest`)

```
r1.addItem(1.95);
r1.addItem(2.99);
System.out.println(r1.getTotal()
                        + " " + r1.getCount());
```

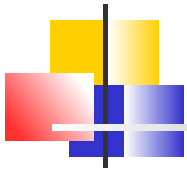


Instance Variables

- An object holds **instance variables** that are accessed by its methods

```
1 public class CashRegister  {
2     private int itemCount;
3     private double totalPrice;
4     // the rest of the class
5 }
```

- values of the instance variables determine the **state** of the object



Instance Variables

Every instance of a class has its **own** set of instance variables

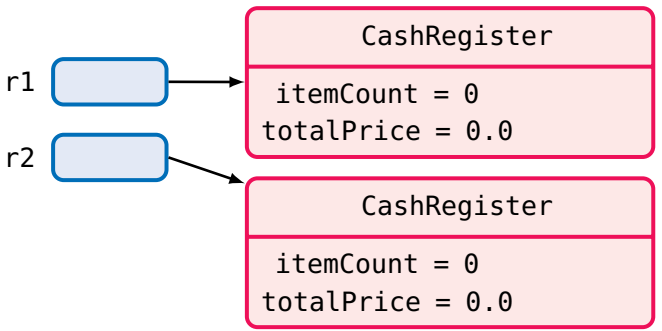
in class `CashRegisterTest`:

```
1   // constructing objects
2   CashRegister r1 = new CashRegister();
3   CashRegister r2 = new CashRegister();
4   // invoking methods
5   r1.addItem(7.67);
6   System.out.println(r1.getTotal() + " " +
7                      r1.getCount());
8   r2.addItem(1.95);
9   r2.addItem(2.99);
10  System.out.println(r2.getTotal() + " " +
11                     r2.getCount());
```

Instance Variables

Every instance of a class has its **own** set of instance variables

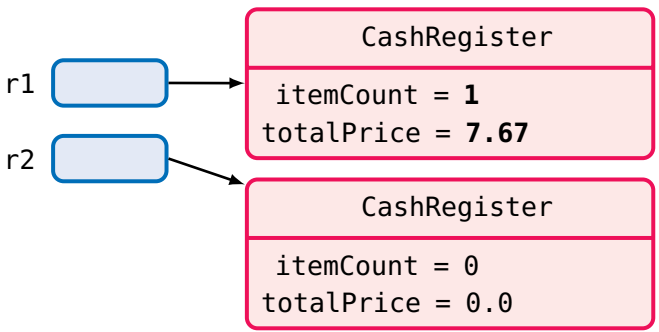
```
1  // constructing objects
2  CashRegister r1 = new CashRegister();
3  CashRegister r2 = new CashRegister();
```

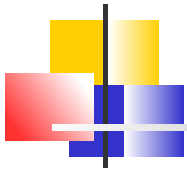


Instance Variables

Every instance of a class has its **own** set of instance variables

```
1  // constructing objects
2  CashRegister r1 = new CashRegister();
3  CashRegister r2 = new CashRegister();
4  // invoking methods
5  r1.addItem(7.67);
```

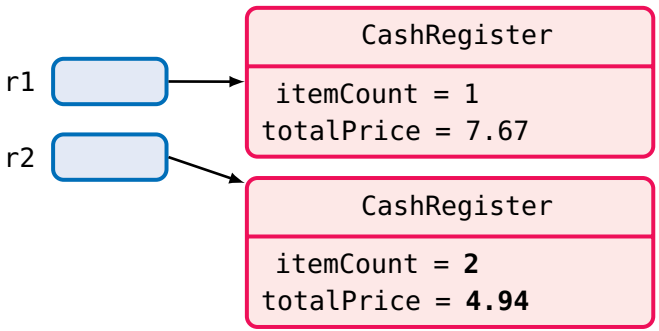




Instance Variables

Every instance of a class has its **own** set of instance variables

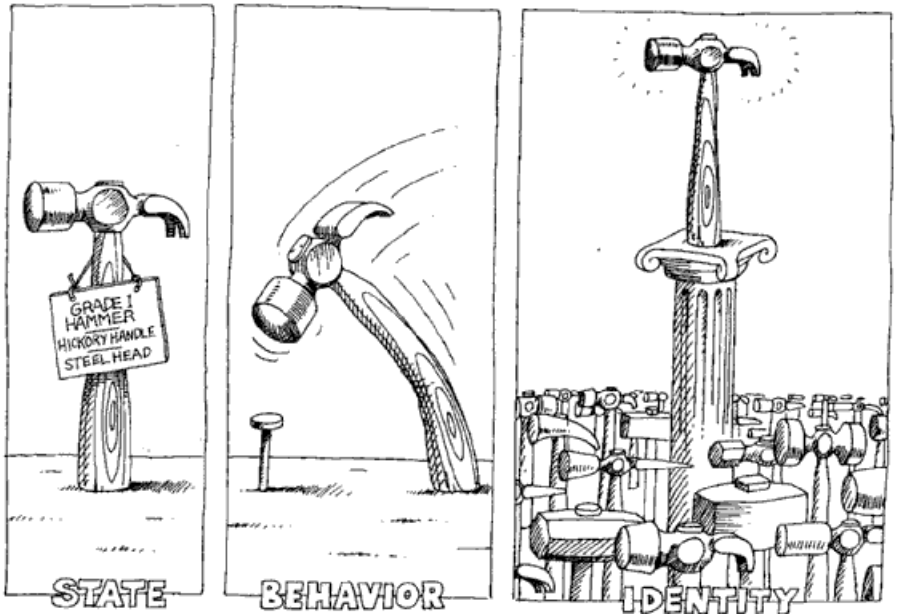
```
1  // constructing objects
2  CashRegister r1 = new CashRegister();
3  CashRegister r2 = new CashRegister();
4  // invoking methods
5  r1.addItem(7.67);
6
7
8  r2.addItem(1.95);
9  r2.addItem(2.99);
```



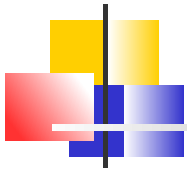
An **object reference** specifies the location of an object (similar to array references!)

Object: State, Behaviour & Identity

NB: equal
≠
identical



Booch, G.: *Object Oriented Analysis and Design with Applications* (2nd Edition) Addison-Wesley, 1994



Access Modifiers

- `private` instance variables (and methods)
  can only be accessed
  by the **methods** of the **same** class

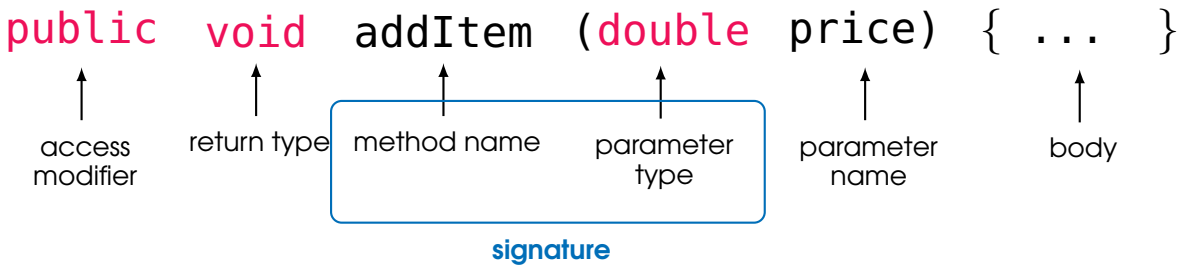
```
// can access instance variables like methods
r1.itemCount = 0;
// COMPILE-TIME ERROR in class CashRegisterTest
```

(it is on the level of **classes**, not individual objects!)

(any instance of `CashRegister` can change instance variables of
any other instance of `CashRegister` provided that it has a reference to it)

- `public` instance variables and methods
  can be accessed by the methods of **any** class

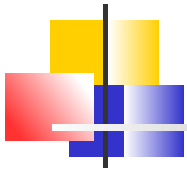
Instance Methods



- all instances variables should be private
- *most* methods should be public

encapsulation

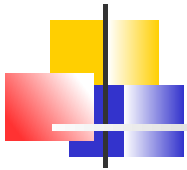
it is useful to classify the methods as
accessors and mutators



Instance Methods: Accessors

- An **accessor** method just queries the object for some information without changing it

```
1  public class CashRegister {
2      private int itemCount;
3      private double totalPrice;
4      // ...
5      public int getCount() {
6          return itemCount;
7      }
8      public double getTotal() {
9          return totalPrice;
10     }
11 }
```

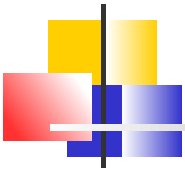


Instance Methods: Mutators

- A **mutator** method changes the object on which it operates

```
1 public class CashRegister {
2     private int itemCount;
3     private double totalPrice;
4     // ...
5     public void addItem(double price) {
6         itemCount++;
7         totalPrice += price;
8     }
9 }
```

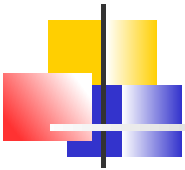
NB: no `return` statement is needed if the return type is `void`



Constructors

- A **constructor** initialises the instance variables of an object
- The name of the constructor is the name of the class (and no return type, not even **void**)

```
1 public class CashRegister {
2     private int itemCount;
3     private double totalPrice;
4     // ...
5     public CashRegister() {
6         itemCount = 0;
7         totalPrice = 0;
8     }
9 }
```

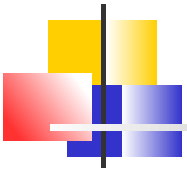


Constructors (cont.)

- By default, instance and class variables are initialised as follows:
  - numbers are initialised as `0`,
  - booleans as `false`, and
  - object and array references as `null` (special reference)

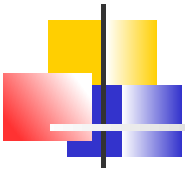
  what about `String s;`?

- If no constructor is provided, a constructor with no parameters is generated by the Java compiler (in bytecode only)



Cash Register Class

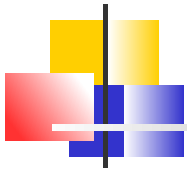
```
1  public class CashRegister  {
2      /* private data (instance variables) */
3      private int itemCount;
4      private double totalPrice;
5      /* methods (public interface) */
6      public void addItem(double price)
7              { itemCount++; totalPrice += price; }
8      public void clear()
9              { itemCount = 0; totalPrice = 0; }
10     public int getCount() { return itemCount; }
11     public double getTotal() { return totalPrice; }
12 }
```



Public Interface of the CashRegister Class

```
1  public class CashRegister  {
2
3
4
5      /* methods (public interface) */
6                  addItem(double       )
7              {                                    }
8                  clear()
9              {                                }
10                getCount() {                    }
11                   getTotal() {                    }
12 }
```

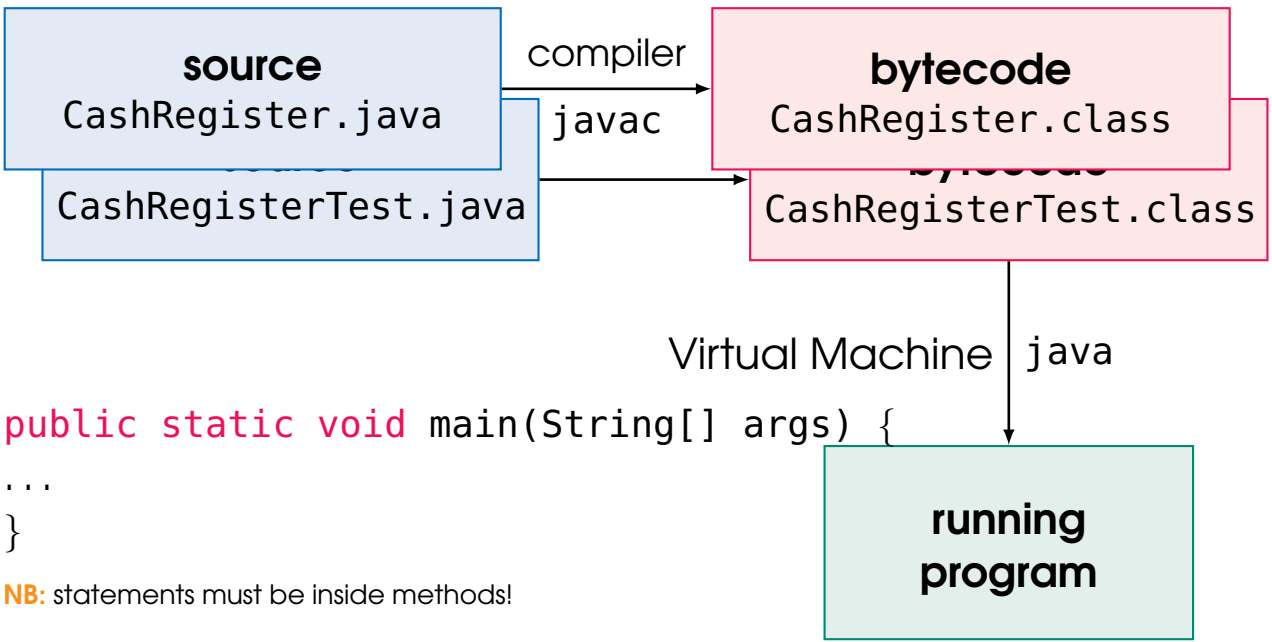
NB: the rest are implementation details,
which may be changed without affecting the users of the class



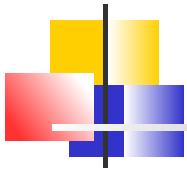
Testing a Class

```
1  public class CashRegisterTest {
2      public static void main(String[] args) {
3          CashRegister r1 = new CashRegister();
4          r1.addItem(2.95);
5          r1.addItem(1.99);
6          System.out.println(r1.getCount());
7          System.out.println((r1.getCount() == 2)
8                                  ? "OK" : "FAIL");
9          System.out.printf("%.2f\n", r1.getTotal());
10         System.out.println((r1.getTotal() == 4.94)
11                                 ? "OK" : "FAIL");
12     }
13 }
```

Java Compilation



NB: statements must be inside methods!



Encapsulation

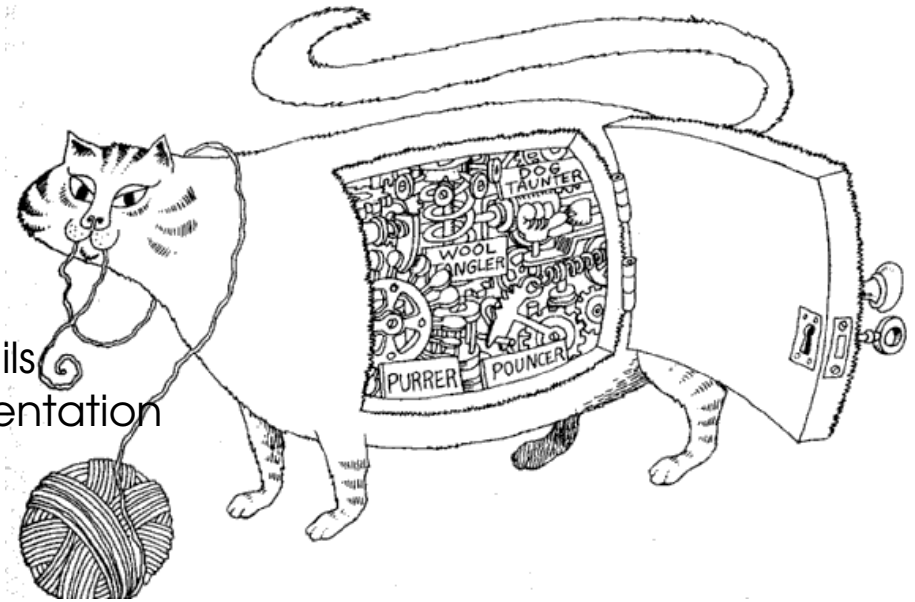
- Every class has a **public interface**:
  a collection of methods through which
  the objects of the class can be manipulated

- Encapsulation is the act of providing a public
  interface and **hiding** implementation details

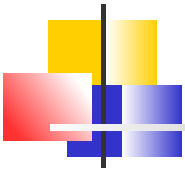
- Encapsulation enables
  changes in the implementation
  without affecting users of the class

Encapsulation

Encapsulation
hides the details
of the implementation
of an object

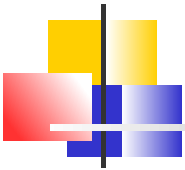


Booch, G.: *Object Oriented Analysis and Design with Applications* (2nd Edition) Addison-Wesley, 1994



Encapsulation: Why?

- What if we want to support a method `void undo()` (cancels the last item)?
- What if we want to implement `CashRegister` using the fixed-point arithmetic (so that `12.92` is `1292`)?
- Instance variables are "hidden" by declaring them `private`, but they are not hidden very well at all. . .



Take Home Messages

- Encapsulation enables
  changes in the implementation
- Public interface: a collection of public methods
- Methods: accessors and mutators
- Every instance of a class has
  its own set of instance variables
- All instance variables should be declared private
- A constructor initialises the instance variables