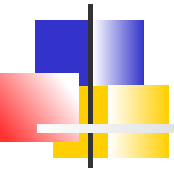
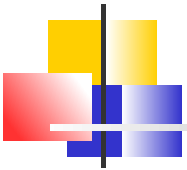
Software and Programming I

More on Objects and Classes



Roman Kontchakov / Carsten Fuhs

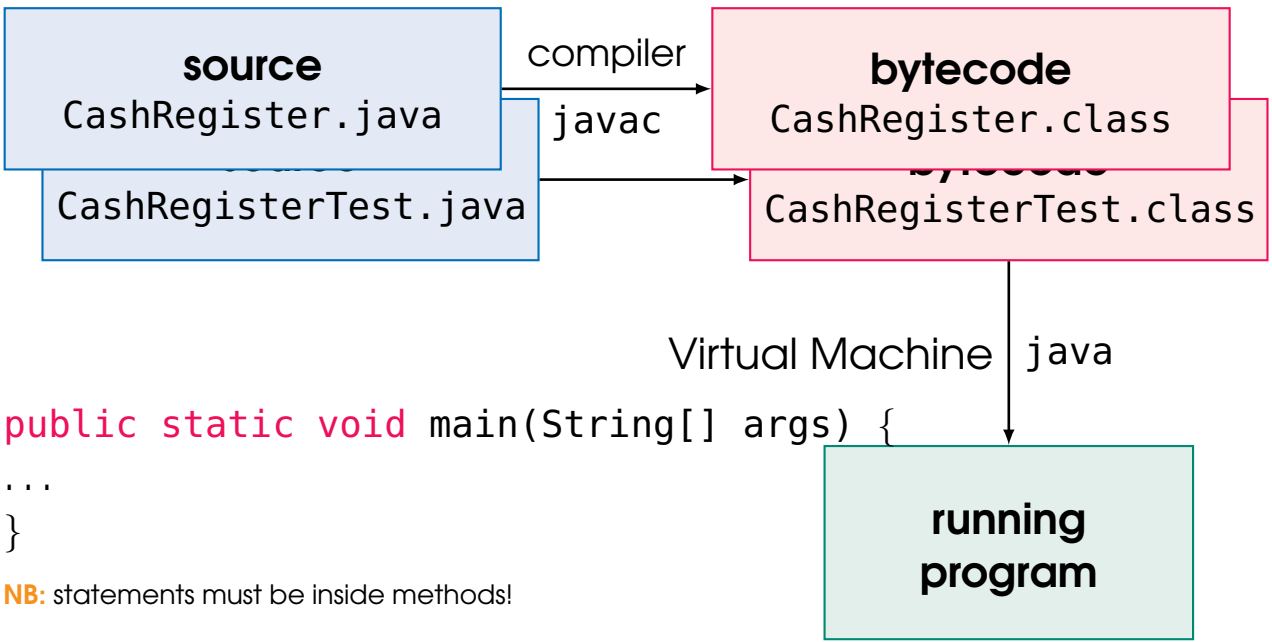
Birkbeck, University of London



Outline

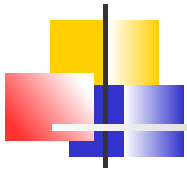
- Object References
- Class Variables and Methods
- Testing a Class
  - Sections 8.7 – 8.11
- slides are available at
  `www.dcs.bbk.ac.uk/~roman/sp1`

Java Compilation




```
public static void main(String[] args) {
...
}
```

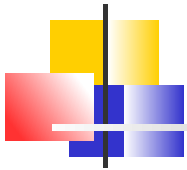
NB: statements must be inside methods!



Object-Oriented Programming

- Tasks are solved by collaborating **objects**
- Each object has its own set of encapsulated **data**, together with a set of **methods** that act upon the data (**public interface**)
- A **class** describes a set of objects with the same structure (i.e., data) and behaviour (i.e., methods)

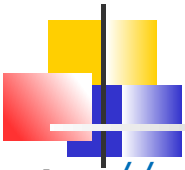
  NB: classes are (user-defined) types in Java
- Encapsulation enables **changes in the implementation** without affecting users of the class
  - all instance variables should be `private`
  - *most* methods should be `public`



Example: Cash Register

```
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 double getTotal() { return totalPrice; }
11     public int getCount() { return itemCount; }
12 }
```

`addItem(double)` and `clear()` are **mutators**; `getTotal()` and `getCount()` are **accessors**

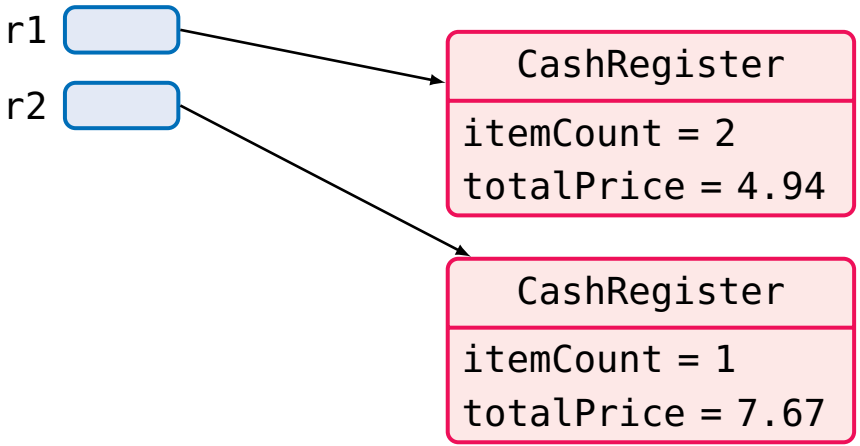


CashRegisterTest Class

```
1  // constructing objects
2  CashRegister r1 = new CashRegister();
3  CashRegister r2 = new CashRegister();
4  // invoking methods
5  r1.addItem(1.95);
6  r2.addItem(7.67);
7  r1.addItem(2.99);
8  System.out.println(r1.getTotal() + " " +
9                     r1.getCount());
10 System.out.println(r2.getTotal() + " " +
11                    r2.getCount());
12 r1.itemCount = 0; // COMPILE-TIME ERROR:
13                   //              private variable
```

Instance Variables

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

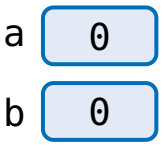


An **object reference** specifies the location of an object

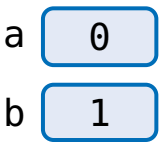
Primitive Datatypes: Values are Copied

```
1 int a = 0;
```


```
2 int b = a;
```



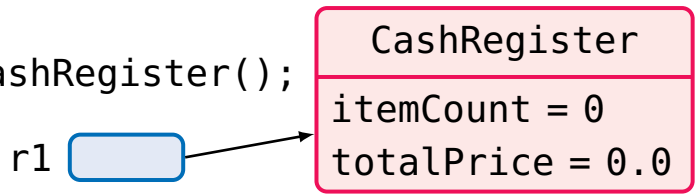

```
3 b++;
```



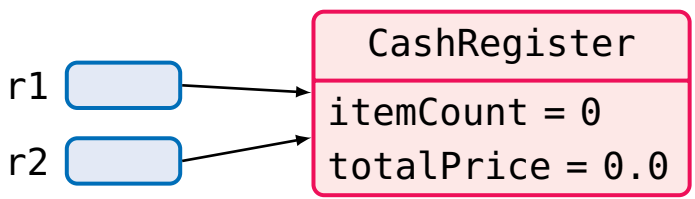
by executing `b++;` only `b` is changed, `a` remains the same

Objects: References are Copied

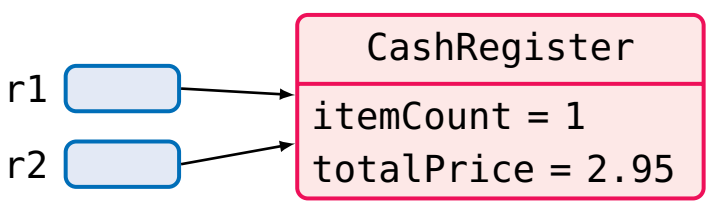
```
1 CashRegister r1 = new CashRegister();
```



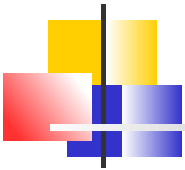

```
2 CashRegister r2 = r1;
```




```
3 r2.addItem(2.95);
```



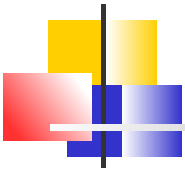
r1 and r2 refer to the **same object** and so, all modifications by methods on r2 are reflected in r1



The null Reference

The `null` reference refers to **no object**

```
1  public static void printName(String firstName,
2                 String middleInitial, String lastName) {
3    if (middleInitial == null)
4        System.out.println(firstName + " " + lastName);
5    else
6        System.out.println(firstName + " " +
7                           middleInitial + " " + lastName);
8  }
9  public static void test() {
10   printName("Alice", "T", "A"); // output: Alice T A
11   printName("Bob", null, "B"); // Bob B (1 space)
12   printName("Ceri", "", "C"); // Ceri  C (2 spaces)
13 }
```



The null Reference (2)

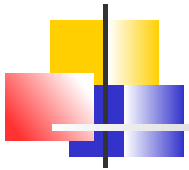
It is an **error** to invoke an instance method on a `null` reference:

```
1  CashRegister r1 = null;
2  // RUN-TIME ERROR: null pointer exception
3  System.out.println(r1.getTotal());
```

NB: without `= null` the above fragment will not compile

local variables must be initialised before use

NB: in contrast, instance and class variables get values by default

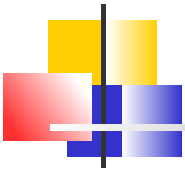


The this reference

In a method (or constructor) `this` refers to
the object the method is called on

```
1  public void addItem(double price) {
2      this.itemCount++; // itemCount++ is a shortcut
3      this.totalPrice += price; // a matter of taste
4  }
5  public CashRegister() {
6      this.clear(); //  clear() is a shortcut
7  }
```

NB: we shall see substantial uses of `this` later

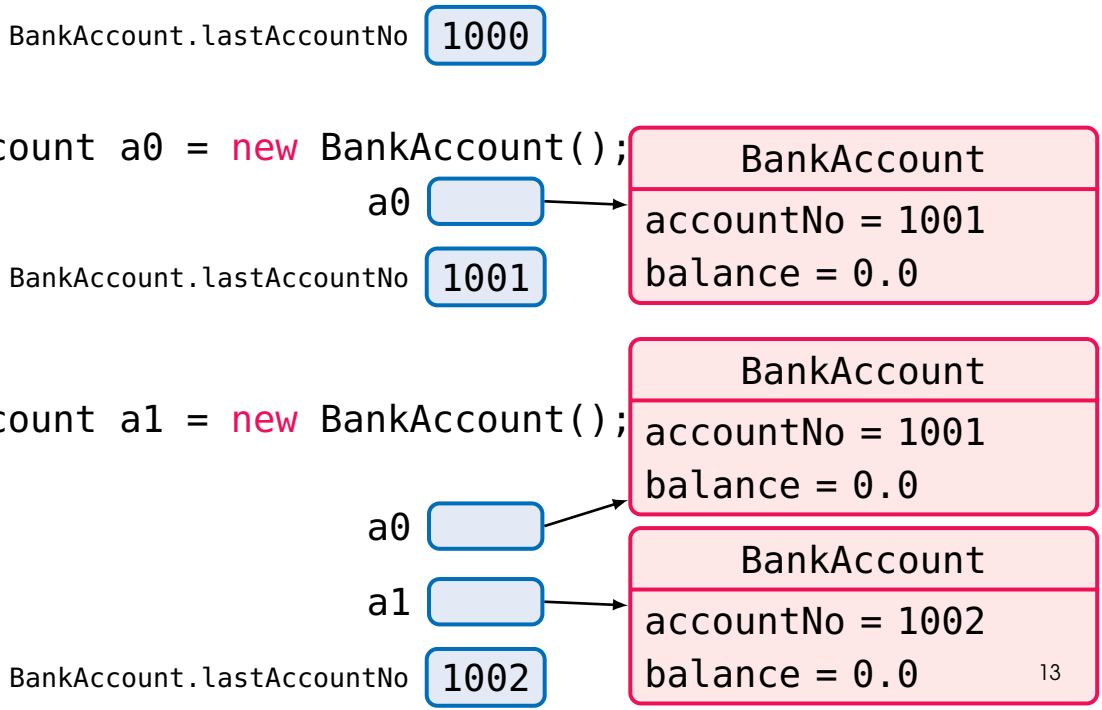


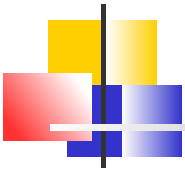
Class Variables

A `static` variable belongs to the class,
not to any object of the class

```
1  public class BankAccount {
2      private double balance; // instance variable
3      private int accountNo; // instance variable
4      // class variable
5      private static int lastAccountNo = 1000;
6
7      public BankAccount() {
8          lastAccountNo++; // one for all instances
9          accountNo = lastAccountNo;
10     }
11 }
```

Class Variables (2)





Class Variables (3)

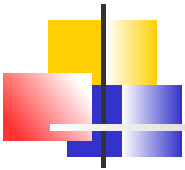
- in a variable declaration, `static` means that
  there is **one copy** of the variable
  shared by all instances of the class

  use *class-name.variable-name*
  or *instance-reference.variable-name* to access it

- in contrast, each instance has
  its **own copy** of all **instance variables** of the class

  use *instance-reference.variable-name* to access it

- `static` ≠ constant



Class Constants

a `final` variable: once it has been assigned,
it always keeps the same value

```
1 public class BankAccount {
2     public static final double OVERDRAFT_FEE = 29.95;
3 }
```

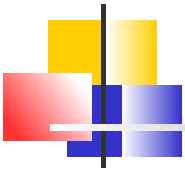
methods from any class can refer to this constant as
`BankAccount.OVERDRAFT_FEE`

NB: other examples include

`Math.PI`: `public static final double PI`

`System.out`: `public static final PrintStream out`

`javax.xml.XMLConstants.XMLNS_ATTRIBUTE`:
`public static final String XMLNS_ATTRIBUTE`



Class Methods

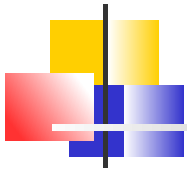
```
1 public class Financial {
2     public static double percentOf(double percentage,
3                                    double amount) {
4         return (percentage / 100) * amount;
5     }
6 }
```

class methods are **not** invoked on an object:

```
System.out.println(Financial.percentOf(50, 100));
```

NB: standard Java classes have many class methods:

e.g., `Math.abs`, `Math.sqrt`, ...



Class Methods (2)

class methods can access class variables
but **cannot** access **instance variables** (e.g. via `this`)

```
1  public class BankAccount {
2      private int accountNo; // instance variable
3      private static int lastAccountNo = 0; // class var
4      public static BankAccount createBankAccount() {
5          BankAccount a = new BankAccount();
6          lastAccountNo++; // one for all instances
7          a.accountNo = lastAccountNo; // OK
8          accountNo = lastAccountNo; // COMPILE-TIME
9                                     // ERROR: no object
10     }
11 }
```

Overloading

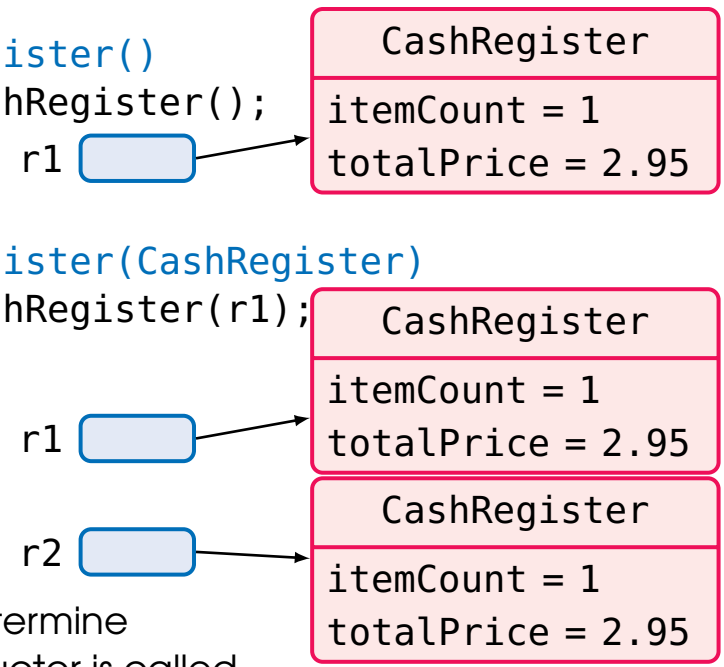
Methods (and constructors) can have the same name (provided their signatures (i.e., name and parameter types) are different)

```
1  public class CashRegister {
2      // [...]
3      public CashRegister() {
4          itemCount = 0; // or this.itemCount = 0;
5          totalPrice = 0; // or this.totalPrice = 0;
6      }
7      public CashRegister(CashRegister c) {
8          this.itemCount = c.itemCount;
9          this.totalPrice = c.totalPrice;
10     }
11 }
```

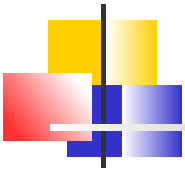
Overloading (2)

```
1 // constructor 1: CashRegister()
2 CashRegister r1 = new CashRegister();
3 r1.addItem(2.95);

4 // constructor 2: CashRegister(CashRegister)
5 CashRegister r2 = new CashRegister(r1);
```



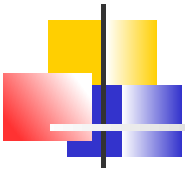
NB: the types of arguments determine which constructor is called



Overloading (3)

When a class has overloaded constructors, use `this` to delegate the initialisation process to another constructor:

```
1  public class BankAccount {
2      private double balance;
3
4      public BankAccount(double initial) {
5          balance = initial;
6      }
7      public BankAccount() {
8          this(1);   // initial balance of 1 pound
9          // same as balance = 1; but better code re-use
10     }
11 }
```



Take Home Messages

- encapsulation enables changes in implementation
  - all instance variables should be `private`
  - *most* methods should be `public` (public interface)
- every instance of a class has its own instance variables
- assignment statements copy
  - values for primitive datatypes
  - object references for object datatypes (classes)
- the `null` reference refers to no object
- instance variables and methods belong to `this` object
- `final` variables never change their values
- `static` variables and methods belong to the class