Software and Programming I

Inheritance and Subclasses

Roman Kontchakov / Carsten Fuhs

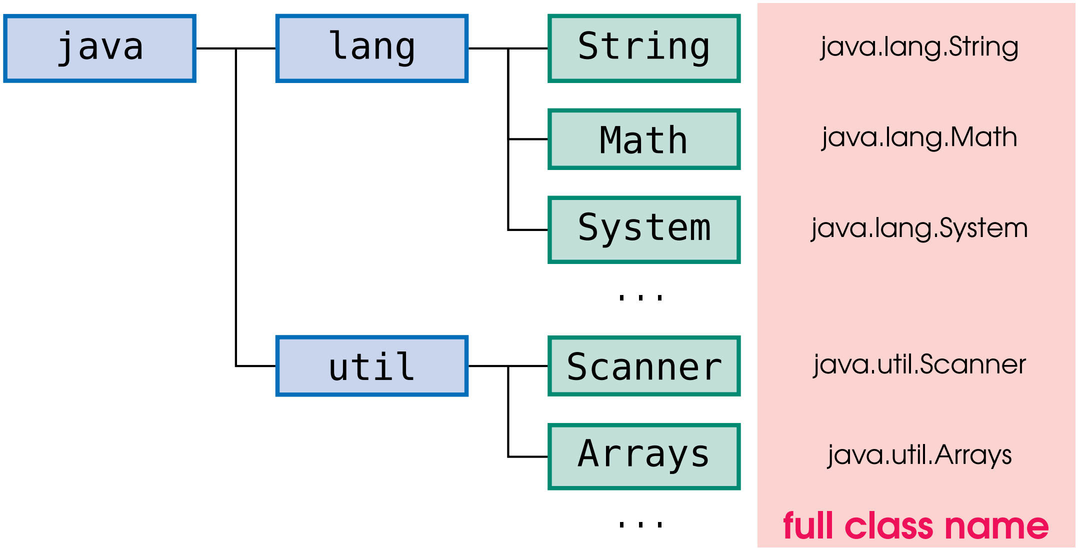
Birkbeck, University of London

Outline

- Packages
- Inheritance
- Polymorphism
  - Sections 9.1 – 9.4
- slides are available at
  `www.dcs.bbk.ac.uk/~roman/sp1`

Packages (1)

- a **package** is a set of related classes, e.g., `java.util`



Packages (2)

- the `import` directive lets you refer to a class from a package by its class name, without the package prefix

```java
import java.util.Scanner;
public class Foo {
...
    Scanner input = new Scanner(System.in);
...
}
```

without this directive one must use the **full class name**

```java
java.util.Scanner input = new java.util.Scanner(System.in);
```

- all classes in `java.lang` are imported by default

Packages (3)

- to import **all classes** of the package `java.util`, use

```
import java.util.*;
```

- to import a particular **class method**, use, e.g.,

```
import static java.lang.Math.abs;
```

  then, one can write `abs(x)` instead of `Math.abs(x)`;

- to import all class methods of a particular class, use, e.g.,

```
import static java.lang.Math.*;
```

  then, one can also write `PI` instead of `Math.PI`;

NB: do not overuse `import` — clashing names would have to be qualified anyway

Packages (4)

- to put a class in a package, use

  `package packagename;`

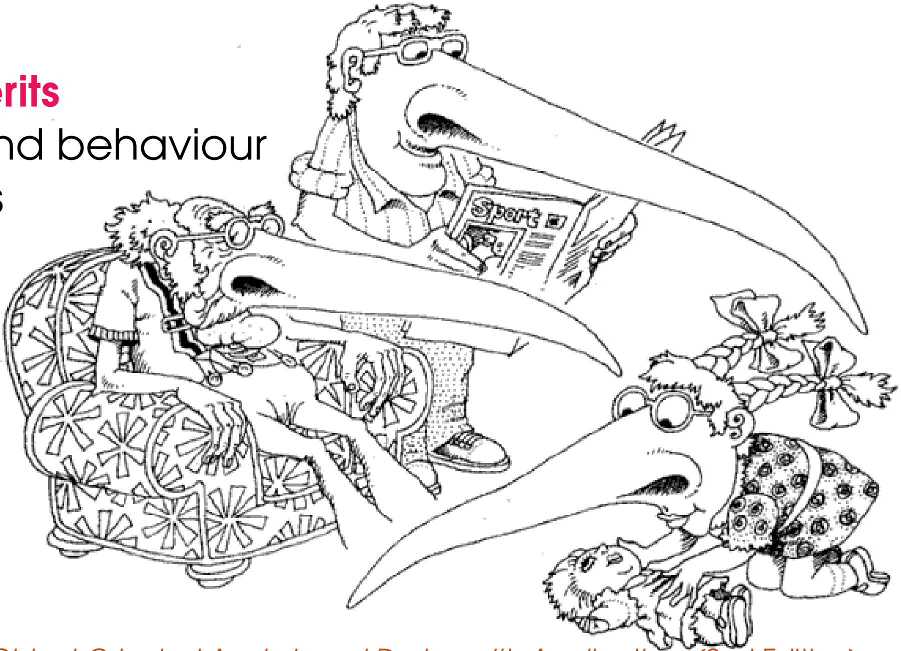
  as the first statement in the source file
- otherwise, the class is in the **default package**, which has no name
- use a **domain name in reverse** to construct an unambiguous package name: e.g., `uk.ac.bbk.dcs.sp1`
- the path of a class file must match its package name: e.g.,

  `uk.ac.bbk.dcs.sp1` is looked up at

  `uk/ac/bbk/dcs/sp1` in the `CLASSPATH` directories

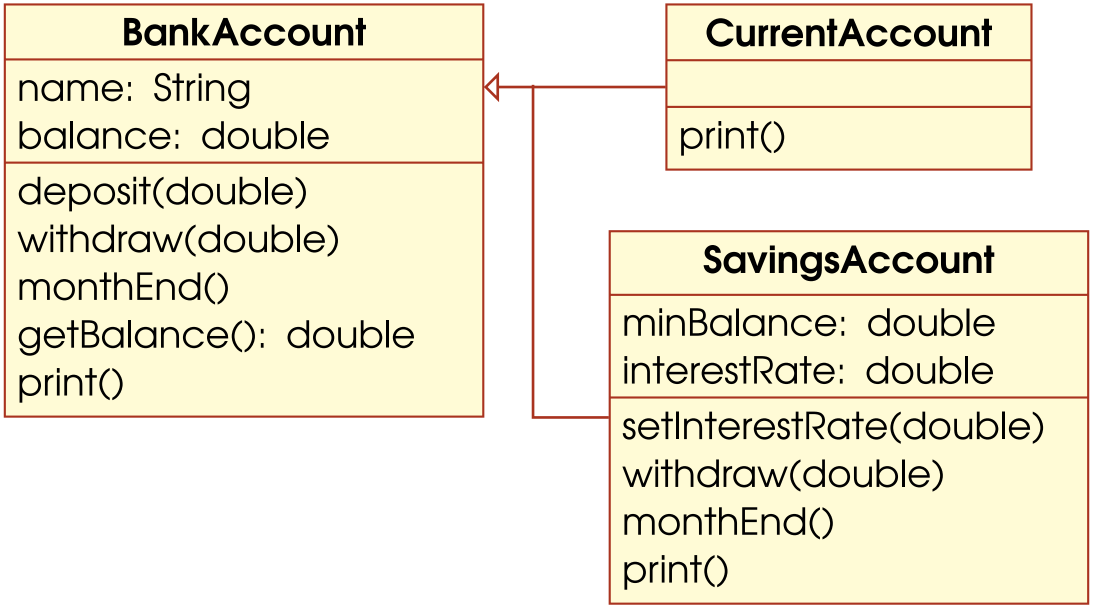
Generalisation

A subclass **inherits** the structure and behaviour of its superclass



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

Inheritance Hierarchies



Implementing Subclasses (1)

```
public class BankAccount {
    private String name;
    private double balance;
    public BankAccount(String name) { // constructor
        this.name = name; // shadowing: see slide 10
        this.balance = 0;
    }
    public void deposit(double amount) {
        balance += amount;
    }
    public void withdraw(double amount) {
        balance -= amount;
    }
```

see next slide

Implementing Subclasses (2)

```
    public void monthEnd() { /* do nothing */ }
    public double getBalance() { return balance; }
    public void print() {
        System.out.print("Account " + name +
                                  ", balance " + balance);
    }
}
```

a subclass **inherits** all methods that it does not **override**

a subclass can override a superclass method by providing a new **implementation**

see slide 11

Shadowing

A declaration of a parameter named `variable` **shadows**, throughout the scope of the declaration, the declarations of any other variables named `variable`.

```java
public class BankAccount {
    private String name;
    public BankAccount(String name) {
    // parameter name shadows instance variable name
        this.name = name;
        this.balance = 0;
    }
}
```

NB: the reserved word `this` can be used to access a shadowed instance/class variable x, using `this.x`

Implementing Subclasses (3)

```java
public class CurrentAccount extends BankAccount {
    public CurrentAccount(String name) {
        super(name); // calls the constructor
                     // of BankAccount
    }
    public void print() {
        System.out.print("Current ");
        super.print(); // calls the implementation
                       // of print() in BankAccount
    }
}
```

NB: what would happen without `super.`?

the reserved word `super` is used to call a superclass method

(or a superclass constructor)

Subclasses and Constructors

Invocation of a superclass constructor with `super` must be the first line in the subclass constructor.

If a constructor does not explicitly invoke a superclass constructor, then the compiler automatically inserts a call to the **no-argument constructor** of the superclass.

NB: a compile-time error if there is no no-argument constructor in the superclass

If a class has **no constructors declared**, then the compiler automatically provides a no-argument **default constructor**

NB: The subclass default constructor will call the no-argument constructor of the superclass.
In this situation, the compiler will complain
if the superclass does not have a no-argument constructor.

Implementing Subclasses (4)

```
public class SavingsAccount extends BankAccount {
    private double interestRate;
    private double minBalance;
    public SavingsAccount(String name,
                          double interestRate) {
        super(name);
        this.interestRate = interestRate;
        this.minBalance = 0;
    }
    public void setInterestRate(double interestRate) {
        this.interestRate = interestRate;
    }
```

see next slide

Implementing Subclasses (5)

```
    public void monthEnd() {
        deposit(minBalance * interestRate / 100);
        minBalance = getBalance();
    }
    public void withdraw(double amount) {
        super.withdraw(amount);
        if (getBalance() < minBalance)
            minBalance = getBalance();
    }
```

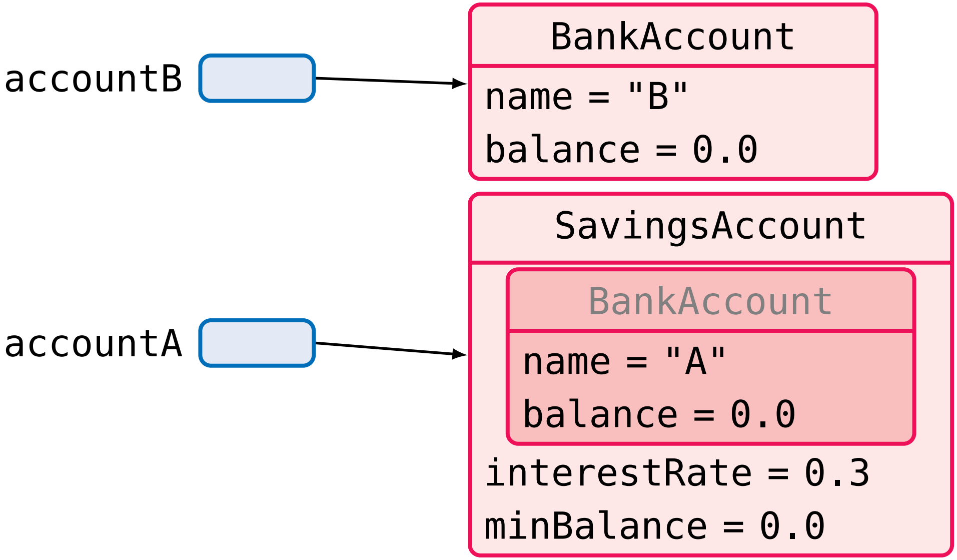
see next slide

Implementing Subclasses (6)

```
    public void print() {
        System.out.print("Savings ");
        super.print();
        System.out.print(", interest rate = " +
                                                    interestRate);
    }
}
```

Subclass Extends Its Superclass

```
BankAccount accountB = new BankAccount("B");
SavingsAccount accountA = new SavingsAccount("A",0.3);
```



Type Casting

```java
SavingsAccount accountA = new SavingsAccount("A",0.3);
BankAccount accountB = new BankAccount("B");
```

Liskov's Substitution Principle: a **subclass** reference can be used when a **superclass** reference is expected

```java
// OK: every SavingsAccount is also a BankAccount
BankAccount accountD = accountA;
// compile-time ERROR!
// not every BankAccount is a SavingsAccount (same for
SavingsAccount accountE = accountB; //... = accountD;)
// ok for compile-time
// BUT could be run-time error if not a SavingsAccount
SavingsAccount accountF = (SavingsAccount) accountD;
```

Polymorphism

polymorphism allows us to manipulate objects that share a set of tasks (even though the tasks are executed in different ways)

```
1 SavingsAccount accountA = new SavingsAccount("A",0.3);
2 BankAccount accountD = accountA;
```

Which methods are called?

```
3 accountA.setInterestRate(2);  //SavingsAccount method
4 accountA.deposit(200);     //BankAccount method
                             //SavingsAccount inherits it
5 accountA.withdraw(50);     //SavingsAccount method
6 accountD.withdraw(10);     //SavingsAccount method
```

polymorphism: the actual class of object is relevant

```
7 accountD.setInterestRate(2);  //compile-time ERROR
                                //BankAccount has no such method
```

Enhanced For Loop

in many ways classes are just like other types in Java,
e.g., one can have an array of `BankAccounts`

```
public static void monthEnd(BankAccount[] accounts) {
    for (BankAccount a: accounts)   // enhanced
        a.monthEnd();               //  for loop
}
```

```
public static void monthEnd(BankAccount[] accounts) {
    for (int i = 0; i < accounts.length; i++) { // more verbose
        BankAccount a = accounts[i];            //  for loop
        a.monthEnd();
    }
}
```

NB: `for (SavingsAccount a: accounts)` will result in a compile-time error

NB: although `monthEnd` has empty implementation in `BankAccount`, one would **not** be able to use `a.monthEnd()` without it

Overriding, Inheritance and Polymorphism: Summary

a subclass **inherits** all methods that it does not **override**

a subclass method overrides a public method from a superclass
if both methods have the **same signature**

a subclass can override a superclass method by providing
a new **implementation**

polymorphism:

- the type of the **reference** determines
  which **method signatures** we may call
  (checked at compile-time)
- the type of the **actual object** determines
  which **method implementation** is invoked (at run-time)

Overriding, Inheritance and Polymorphism: Example (1)

```java
public class A {
    public int f() { return 1; }
    public int g() { return 2; }
}
public class B extends A { // g() is inherited from A
    public int f() { return 3; } // f() is overridden
    public int h() { return 4; } // h() is new
}
```

```java
A a = new A();
System.out.println(a.f() + " " + a.g()); // prints 1 2
// a.h() is a compile-time error
```

Overriding, Inheritance and Polymorphism: Example (2)

```java
public class A {
    public int f() { return 1; }
    public int g() { return 2; }
}
public class B extends A { // g() is inherited from A
    public int f() { return 3; } // f() is overridden
    public int h() { return 4; } // h() is new
}
```

```java
B b = new B();
          // prints 3 2 4
System.out.println(b.f() + " " + b.g() + " " + b.h());
```

Overriding, Inheritance and Polymorphism: Example (3)

```
public class A {
    public int f() { return 1; }
    public int g() { return 2; }
}
public class B extends A { // g() is inherited from A
    public int f() { return 3; } // f() is overridden
    public int h() { return 4; } // h() is new
}
```

```
A c = new B();// c provides A’s methods, uses B’s code
System.out.println(c.f() + " " + c.g()); // prints 3 2
// c.h() is a compile-time error
```

Take Home Messages

- a subclass inherits all methods that it does not override
- a subclass can override a superclass method by providing a new implementation
- polymorphism allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways