Software and Programming I

Inheritance and Interfaces

Roman Kontchakov

Birkbeck, University of London
Outline

- Inheritance
- Polymorphism
- Class Object
- Interfaces
- slides are available at
  www.dcs.bbk.ac.uk/~roman/sp1
Generalization

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

Addison-Wesley, 1994
Inheritance Hierarchies

**BankAccount**
- name: String
- balance: double
- deposit(double)
- withdraw(double)
- monthEnd()
- getBalance(): double
- print()

**CurrentAccount**
- print()

**SavingsAccount**
- minBalance: double
- interestRate: double
- setInterestRate(double)
- withdraw(double)
- monthEnd()
- print()
public class BankAccount {
    private String name;
    private double balance;
    public BankAccount(String name) { // constructor
        this.name = name;
        this.balance = 0;
    }
    public void deposit(double amount) {
        balance += amount;
    }
    public void withdraw(double amount) {
        balance -= amount;
    }
}
a subclass inherits all methods that it does not override
a subclass can override a superclass method by providing a new implementation
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)
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;
    }
}
Implementing Subclasses (5)

```java
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)

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

// 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)

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

Which methods are called?

```java
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

```java
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.

```java
public static double sum(BankAccount[] accounts) {
    double sum = 0;
    for (BankAccount a: accounts) { // enhanced
        sum += a.getBalance(); // for loop
    }
    return sum;
}

public static double sum(BankAccount[] accounts) {
    double sum = 0;
    for (int i = 0; i < accounts.length; i++) { // more verbose
        BankAccount a = accounts[i]; // for loop
        sum += a.getBalance();
    }
    return sum;
}
```
The Object Class

every class declared without the explicit `extends` clause automatically extends the class `Object`.

Class `Object` has method `String toString()`, which one can override.

```java
public class BankAccount {
    public String toString() {
        return "Account " + name + ", balance " + balance;
    }
}
```
Using Object.toString()

the method toString() is called, e.g., in System.out.println

```java
public static void printAll(BankAccount[] accounts) {
    for (BankAccount a: accounts)
        System.out.println(a);
}
```
Methods of Object

- `boolean equals(Object obj)`
  indicates whether some other object is “equal to” this one

- `int hashCode()`
  returns a hash code value for the object
  (used in collections)

- `Class getClass()`
  returns the runtime class of an object
Overloading

Methods and constructors can have the same name (provided their signature (i.e., name and types of parameters) are different)

```java
class BankAccount {
    // ... 
    public BankAccount(String name) {
        this.name = name;
        this.balance = 0;
    }
    public BankAccount(BankAccount a) {
        this.name = "copy of " + a.name;
        this.balance = a.balance;
    }
}
```
Overloading (2)

1 // constructor 1: BankAccount(String)
2 BankAccount a = new BankAccount("A");
3 a.deposit(100);

   a

   BankAccount
   name = "A"
   balance = 100.0

4 // constructor 2: BankAccount(BankAccount)
5 BankAccount b = new BankAccount(a);

   b

   BankAccount
   name = "copy of A"
   balance = 100.0

NB: type of arguments determines which constructor is called
Java does not support multiple inheritance

an interface type declares

a set of methods and their signatures

unlike a class, an interface type provides no implementation

```java
public interface Measurable {
    double getMeasure(); // interface methods are always public
}
```
Implementing an Interface

```java
public class BankAccount implements Measurable {
    ...
    public double getMeasure() {
        return getBalance();
    }
}

public class Country implements Measurable {
    ...
    public double getMeasure() {
        return getArea();
    }
}
```
The Interface Comparable

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class’s natural ordering.

```java
public interface Comparable<T> { // T is a type name
    int compareTo(T other);
}

public class BankAccount
    implements Comparable<BankAccount> {
    ...  
    public int compareTo(BankAccount other) {
        return this.getBalance() - other.getBalance();
    }
}
```
BankAccount[] accounts = new BankAccount[3];

java.util.Arrays.sort(accounts);

**NB:** String and many other standard classes implement this interface
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.
- Every class automatically extends the class `Object`.
- An interface type declares a set of methods and their signatures.