Contents

1. Classes and Objects
2. Inheritance
3. Interfaces
4. Exceptions and Error Handling
5. Intro to Concurrency
6. Concurrency in Java
7. Graphics and Animation
8. Applets
Chapter 2: Inheritance

After this chapter you will be able to:

- Extend existing classes using inheritance
- Apply polymorphism
- Understand static and dynamic binding
- Use abstract classes
Inheritance

- Can define a class by *inherit*ing the data attributes and methods of another object
- Inheritance is used to model the Is-A relationship

**class Person**

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
</tr>
<tr>
<td>Location</td>
</tr>
</tbody>
</table>

**class Employee**

<table>
<thead>
<tr>
<th>SS#</th>
</tr>
</thead>
<tbody>
<tr>
<td>work()</td>
</tr>
</tbody>
</table>

- An Employee Is-A Person
Instantiation with Inheritance

• An Employee object has attributes defined in class Employee and Person

<table>
<thead>
<tr>
<th>Name:</th>
<th>Fred</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age:</td>
<td>40</td>
</tr>
<tr>
<td>Location:</td>
<td>New York</td>
</tr>
<tr>
<td>SS#:</td>
<td>321-11-1232</td>
</tr>
</tbody>
</table>
Other Inheritance Examples

Chair

Desk Chair

Lounge Chair

Phone

Rotary Phone

Touch Tone Phone

Adjustable Desk Chair

Cordless Phone
Benefits

• Can use existing classes when defining new (more specialized) classes

We will see later that polymorphism will:
  – facilitate software maintenance
  – allow us to write code that is generic
Other Languages

• Other OO languages with inheritance are:
  – C++
  – Eiffel
  – SmallTalk
  – Ada95
Terms

• More generic class is called the superclass (parent)

• More specialized class is called the subclass (child)

• The set of Lounge chairs is a subset of the set of all Chairs
Inheritance in Java

- **Use keyword** extends

```java
class Account {
    private int balance;
    private String owner;

    public Account (int b, String o) {...}
    public Account () {...}
    void deposit (int d) { ... }
    void withdraw (int d) { ... }
    public int getBalance () {...}
    public String getOwner () {...}
}

class BrokerageAccount extends Account {
    private int numShares;

    public BrokerageAccount () { numShares = 0; }
}
```
Using A Brokerage Account

• Can access the public interface provided by class Account

```java
BrokerageAccount b = new BrokerageAccount();

System.out.println("Balance = " + b.getBalance());
b.deposit(100);
System.out.println("Balance = " + b.getBalance());
```

Output

<table>
<thead>
<tr>
<th>balance:</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>owner:</td>
<td>&quot;nobody&quot;</td>
</tr>
<tr>
<td>numShares:</td>
<td>0</td>
</tr>
</tbody>
</table>

Balance = 0
Balance = 100
Inheritance and Private

• Cannot access the inherited private parts

```java
BrokerageAccount b = new BrokerageAccount();
System.out.println("balance = " + b.balance);
```

• Compile-time error
Inheritance and Constructors

• **Problem:** the constructor needs to initialize private parts of class `Account`

```java
class BrokerageAccount extends Account {
    private int numShares;

    public BrokerageAccount () { numShares = 0; }
    public BrokerageAccount (int balance, String owner) {
        this.balance = balance;
        this.owner = owner;
    }
}
```

• **Solution:** use keyword `super` to invoke the superclass constructor
Calling Superclass Constructor

• Call the superclass constructor which accepts `int` and `String` arguments

```java
class BrokerageAccount extends Account {
    private int numShares;

    public BrokerageAccount () { numShares = 0; }
    public BrokerageAccount (int balance, String owner) {
        super(balance, owner);
    }
}
```

• Calling `super` will search the superclasses in the inheritance hierarchy for a matching constructor
More About Superclass Constructors

• In fact when an object is created some constructor for every class in the related hierarchy must be called.

• The default constructor of superclasses is invoked automatically unless `super` is used to call a non-default constructor.
Question

• Compile-time error: can you see why?

class Account {
    private int balance;
    private String owner;

    public Account (int b, String o) {...}
    void deposit (int d) { ... }
    void withdraw (int d) { ... }
    public int getBalance () {...}
    public String getOwner () {...}
}

class BrokerageAccount extends Account {
    private int numShares;
    private int stockPrice;

    public BrokerageAccount () {
        numShares = 0; stockPrice = 0; }
}
Answer

• No default constructor is provided by class Account

```java
class Account {
    ...
    public Account (int b, String o) {...} ...
    ...
}
class BrokerageAccount extends Account {
    private int numShares;
    private int stockPrice;

    public BrokerageAccount () {
        super(0,"nobody"); /*gets rid of error */
        numShares = 0;
        stockPrice = 0;
    }
}
```
Buying Stock

- Buying involves withdrawing from the cash part of the Account

```java
class BrokerageAccount extends Account {
    private int numShares;
    private int stockPrice;
    ...
    public void buy (int numShares) {
        int transactionAmount = numShares * stockPrice;
        if (getBalance() < transactionAmount) {
            System.out.println("illegal transaction");
            return;
        }
        this.numShares += numShares;
        withdraw(transactionAmount);
    }
}
```
Balance of Brokerage Acct

• Balance is the sum of the cast Account and the value of the stock at a given price

• Can override the definition of getBalance in class Account with a definition in class BrokerageAccount

```java
class BrokerageAccount extends Account {
    private int numShares;
    ...
    public int getBalance () {...

```

• Problem: need to call getBalance in the superclass to get the cash balance
Solution

- Use keyword `super` to call the superclass method

```java
class BrokerageAccount extends Account {
    private int numShares;
    ...
    public int getBalance () {
        int cash = super.getBalance();
        return cash + (numShares * stockPrice);
    }
}
```
Complete Example

class BrokerageAccount extends Account {
    private int numShares;

    public BrokerageAccount () { numShares = 0; }
    public BrokerageAccount (int balance, String owner) {
        super(balance, owner);
    }

    public void buy (int numShares) {
        int transactionAmount = numShares * stockPrice;
        if (getBalance() < transactionAmount) {
            System.out.println("illegal transaction");
            return;
        }
        this.numShares += numShares;
        withdraw(transactionAmount);
    }

    ...
}
Example continued

```java
public void buy (int numShares) {
    int transactionAmount = numShares * stockPrice;
    if (getBalance() < transactionAmount) {
        System.out.println("illegal transaction");
        return;
    }
    this.numShares += numShares;
    withdraw(transactionAmount);
}

public void sell (int numShares) {
    if (this.numShares < numShares) {
        System.out.println("illegal translation");
        return;
    }
    this.numShares -= numShares;
    deposit(numShares * stockPrice);
}...
```
Example continued

```java
public int getBalance () {
    int cash = super.getBalance();
    return cash + (numShares * stockPrice);
}

d public void setStockPrice (int p) {
    stockPrice = p;
}

} /* end class BrokerageAccount */
Calling getBalance

- Can call `getBalance` using `Account` and `BrokerageAccount` objects

```java
Account a = new Account(1000,"fred");
System.out.println("fred’s balance = " + a.getBalance());

BrokerageAccount b = new BrokerageAccount(600,"mary");
b.buy(10, 50);
System.out.println("mary’s balance = " + b.getBalance());
```

**Output**

fred’s balance = 1000  
mary’s balance = 600
Subclass to Superclass Conversion

- A variable of type A can be bound to any object of type A or any of A's subclasses

```java
Account b = new BrokerageAccount(600, "mary");
```

- This rule allows BrokerageAccount objects to be used in contexts where the Account type is specified

b’s type

object b will be bound to at run-time
Illegal Conversion

• The converse is not legal: A variable cannot be bound to any superclasses of it’s type

```
BrokerageAccount b = new Account(600,"mary");
```

• Compile-time error

Soon we’ll see that the legal conversion will allow us to incorporate *polymorphism*
Calling Deposit

• **Method lookup**: performed at run-time starting at the object’s class and then continuing with superclasses (if necessary) until a corresponding method definition is found

```java
Account a = new Account(1000,"fred");
a.deposit(100);  /*  calls Account.deposit() */
Account b = new BrokerageAccount(600,"mary");
b.deposit(100); /*  no definition in BrokerageAccount, calls Account.deposit() */
```
Calling getBalance

- There are two definitions of getBalance

```java
Account a = new Account(1000,"fred");
a.getBalance();  // calls Account.getBalance() */

Account b = new BrokerageAccount(600,"mary");
b.getBalance(100); // calls BrokerageAccount.getBalance()*/
```

- The decision regarding which method to call is made at run-time based on the object (not the variables type)
Static vs Dynamic Binding

• **Static binding**: decides which method to call at *compile-time* based on type information

• **Dynamic binding**: decides which method to call at *run-time* based on the object used

• In Java (and SmallTalk, Eiffel) the default is dynamic binding

• In C++ the default is static binding
The Tradeoff

• Dynamic binding:
  - extra overhead (need to decide which method to call at run-time)
  + flexibility

• Static binding:
  - less flexibility (decision is fixed at compile-time)
  + efficiency
Question

• Which \texttt{getBalance} is going to be called?

```java
void processAccount (Account a) {
    ...  
    balance = a.getBalance();
    ...
}
```
Answer

• Could be either!

• `processAccount` could be called multiple times with `Account` and `BrokerageAccount` objects

```java
void processAccount (Account a) {
    ... 
    balance = a.getBalance();
    ...
}
```

• the type of object will determine which `getBalance` gets called
End of Part I
It’s Exercise Time
Part II
Polymorphism

• English meaning of polymorphism is: pertaining to *multiple forms*

• In the context of programming languages polymorphism means:
  – programs which operate on many types
  – data which have many types

• Two kinds of polymorphism:
  – *OO polymorphism*: constrained by a set of classes related by inheritance
  – *Parametric polymorphism*: type variables are used
Benefits of Polymorphism

- **Genericity**: Can write generic code that can be applied to different kinds of objects

```java
void processAccount (Account a) {
    ... balance = a.getBalance();
    ...
}
```

- method `processAccount` can be applied to `Account` and `BrokerageAccount` objects
- The appropriate `getBalance` will be called
Benefits of Polymorphism

• **Software maintenance:** can extend the inheritance hierarchy with new classes that can work with existing code

```java
class IRA_Account extends Account {
    public int getBalance () {...}
}
```

```java
processAccount(new IRA_Account(200,"sam"))
```

• Can call `processAccount` with an `IRA_Account` object without recompiling

`processAccount`
Another Example

class Car {
    private String type;
    public Car (String type) { this.type = type; }
    public void start () { .../* generic start */ }
}
class SportsCar extends Car {
    public SportsCar (String type) { super(type); }
    public void start () { .../* specific for SportsCar */ }
}

class Limo extends Car {
    public Limo (String type) { super(type); }
    public void start () { .../* specific for Limo */ }
}

```java
class Person {
    Car myCar = null;
    public Person (Car myCar) { this.myCar = myCar; }
    public void startCar () { myCar.start(); }
}
```
Questions

• Calling `startCar` will indirectly call method `start` of which class?

```java
Person ed = new Person(new Car("chevy"));
ed.startCar();
```

• How about now?

```java
Person sally = new Person(new Limo("lincoln"));
sally.startCar();
```
Answers

```java
Person ed = new Person(new Car("chevy"));
ed.startCar();  // indirectly call Car.start() */
```

```java
Person sally = new Person(new Limo("lincoln"));
sally.startCar();  // indirectly call LimoCar.start() */
```

```java
class Person {
    Car myCar = null;
    public Person (Car myCar) { this.myCar = myCar; }
    public void startCar () { myCar.start(); }
}
```
Adding MiniVans

```java
class MiniVan extends Car {
    public MiniVan (String type) { super(type); }
    public void start () { /* specific for MiniVan */ }
}
```

• Can use MiniVans with class Person

```java
Person sally = new Person(new MiniVan("lincoln"));
sally.startCar(); /* indirectly call MiniVan.start() */
```

• and the **right** start gets called
Summary

• We applied OO polymorphism to:

  – make class Person more generic (a person can have different types of cars)

  – facilitate software maintenance (added a new type of car which still works with class Person)
Inheritance and Access

• Using protected, a class can present a different interface for its subclasses
  - protected: access given to the class and its subclasses
  - public: access unrestricted
  - default (no keyword specified): access given to code inside the same package as the class
Using Protected

- **ID** can be accessed from subclasses of `Account`

```java
class Account {
    private int balance;
    private String owner;
    protected int ID;
    public Account (int b, String o, int ID) {...}
    void deposit (int d) { ... }
    public int getBalance () {...}
    public String getOwner () {...}
}
```
Direct Access From Subclasses

- Can access ID from BrokerageAccount

```java
class BrokerageAccount extends Account {
    private int numShares;
    ...
    public void print () {
        System.out.println("Brokerage Acct: " + ID);
    }
}
```
But Not From Outside

- Cannot Access ID outside of Account and BrokerageAccount

```java
Account a = new Account(1000,"fred",402);
System.out.println("ID = " + a.ID);
```

- Compile-time error
Abstract Classes

- **Abstract classes:**
  - cannot be instantiated
  - may contain the specifications of methods that must be defined by *concrete* subclasses

- **Concrete classes** are subclasses of abstract classes where all abstract method specifications have been defined (or have been defined by intermediate superclasses)
Declaring An Abstract Class

• Use keyword `abstract`

```java
abstract class Thing {
}
```

• Cannot create a Thing object

```java
new Thing();
```

• Compile-time error
Abstract Methods

• Use abstract in the method specification

```
abstract class Thing {
    String name;
    public Thing (String name) { this.name = name; }
    public String getName () { return name; }
    public abstract void print (); /* no body */
}
```

• An abstract class may contain both abstract and concrete methods
Concrete Class

• Can make a concrete class by:
  – extending Thing
  – implementing the print specification

```java
class Person extends Thing {
    String address;
    public Person (String name, String address) {
        super(name);
        this.address = address;
    }
    public void print () {
        System.out.println("Person: " + getName());
        System.out.println("Address: " + address);
    }
}
```
Error

• Not defining print would produce a compile-time error

```java
class Person extends Thing {
    String address;
    public Person (String name, String address) {
        super(name);
        this.address = address;
    }
    /* no definition of print */
}
```

• Must either define print or define Person as abstract
Another Concrete Class

• Can have different concrete classes with different implementations of print

class Car extends Thing {
    String model;

    public Car (String name, String model) {
        super(name);
        this.model = model;
    }

    public void print () {
        System.out.println("Car: " + getName());
        System.out.println("Model: " + model);
    }
}

Reference

• "On Understanding Types, Data Abstraction and Polymorphism" by Luca Cardelli (Microsoft Research) available on the web at:
  – www.luca.demon.co.uk
    • click on "Papers" link and then look for title
It’s Exercise Time