
Inheritance



This is the first of two inheritance-related chapters, where we show how to define a hierarchy of classes that differ from each other incrementally. This chapter focuses on inheritance itself, and the following chapter looks at the related topic of interfaces.

Inheritance and interfaces are essential ingredients in object-oriented programming. It's imperative that you have a good understanding of these topics.

Contents

1. Overview of inheritance
2. Defining superclasses and subclasses
3. Polymorphism
4. Additional techniques



Demo project: DemoInheritance

2

Section 1 provides a quick introduction to the concepts and terminology of inheritance. We'll explain what inheritance is, and why it matters.

Section 2 shows the Java syntax for defining superclasses and subclasses. There are several important language features and keywords that come into play here.

Section 3 introduces the wonderful world of polymorphism, whereby you can treat objects of a class hierarchy in a consistent manner without needing to know exactly what type of object you're actually dealing with.

Section 4 finishes off our discussion of inheritance by looking at some important additional techniques, such as abstract classes, abstract methods, and so on.

The demos for chapter are located in the DemoInheritance project.

1. Overview of Inheritance

- Inheritance and OO
- Superclasses and subclasses
- Inheritance in Java
- The Object class



We kick off with a brief overview of the role and importance of inheritance in object-oriented applications. We'll also discuss some important philosophical issues about the way Java approaches inheritance.

Inheritance and OO

- Inheritance is a very important part of object-oriented development
 - Allows you to define a new class based on an existing class
 - You just specify how the new class differs from the existing class
- Terminology:
 - For the "existing class": Base class, superclass, parent class
 - For the "new class": Derived class, subclass, child class
- Potential benefits of inheritance:
 - Improved OO model
 - Faster development
 - Smaller code base

4

Inheritance allows you define a new class based on an existing class. The existing class is known as the superclass, and the new class is known as the subclass. Note that some other OO programming languages use alternative terminology here; for example, C++ developers tend to talk about base classes and derived classes. No matter.

One of the main benefits of inheritance is code reuse. You can develop a new class more quickly, simply by specifying how it differs from an existing class. Another key benefit of inheritance is that it makes sense; it results in an object model that resembles the real world.

Superclasses and Subclasses

- The subclass inherits everything from the superclass (except constructors)
 - You can define additional variables and methods
 - You can override existing methods from the superclass
 - You typically have to define constructors too
 - Note: You can't cherry pick or "blank off" superclass members

5

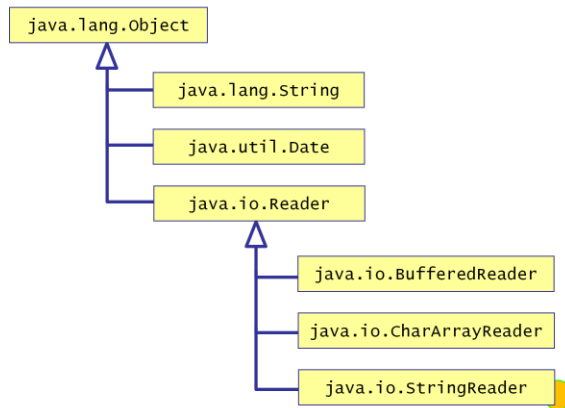
When you define a subclass, i.e. when you define a class that inherits from an existing class, the subclass inherits everything from the superclass (except for constructors, which we'll address later).

The subclass can add extra data members and methods, as relevant for that class. The subclass can also potentially redefine methods defined in the superclass, so that they behave differently in the subclass; this is known as method overriding, and it lies at the heart of polymorphism.

Inheritance in Java

- Java supports single inheritance
 - All classes have one direct superclass (i.e. Java does not support multiple inheritance of classes)
 - Everything ultimately inherits from `java.lang.Object`

- Here's a tiny snippet of the standard Java inheritance hierarchy



Java supports single inheritance of implementation. In other words, a class can have only one direct superclass. This is in stark contrast to C++, which supports multiple inheritance and allows a class to have any number of direct superclasses.

All the classes in Java are arranged into a single inheritance hierarchy. At the top of this hierarchy is the `Object` class, defined in the `java.lang` package. Thus all classes ultimately inherit from `Object`.

The Java SE library contains thousands of classes. All of these classes fit in somewhere in the grand scheme of Java's inheritance hierarchy. Some classes are many levels removed from `Object`, and inherit everything from each class above them in the inheritance chain back to `Object`.

The Object Class

- The Object class defines methods that are available to all types in Java
 - `clone()`
 - Creates and returns a copy of object (object must implement `Cloneable`)
 - `equals()`
 - Indicates whether some other object is "equal to" this one
 - `finalize()`
 - Called automatically on object when it is being garbage collected
 - `getClass()`
 - Returns a `Class` object that provides run-time information about this object
 - `hashCode()`
 - Returns a hash code value for object
 - `notify()`, `notifyAll()`, `wait()`
 - Associated with multithreading (see later in the course)
 - `toString()`
 - Returns string representation of object (default returns `classname@hashCode`)

7

The Object class provides a small number of general-purpose methods, as listed in the slide above.

We've encountered some of these methods already, such as `toString()`, `equals()`, and `finalize()`. We'll discuss some of the other methods later in the course, e.g. `notify()`, `notifyAll()`, and `wait()`.

Every Java class inherits all these methods.

2. Defining Subclasses & Superclasses

- Sample hierarchy
- Superclass considerations
- Access modifiers
- Defining a superclass
- Defining a subclass
- Adding new members
- Defining constructors
- Overriding methods

8

Now that we've discussed the concepts and terminology of inheritance, it's time to look at the syntax. This section describes how to define superclasses and subclasses, and shows how to implement common inheritance features using Java programming language constructs.

During OOA/OOD (or even during coding ☺) you decide how to organize classes in your inheritance hierarchy. One approach is generalization:

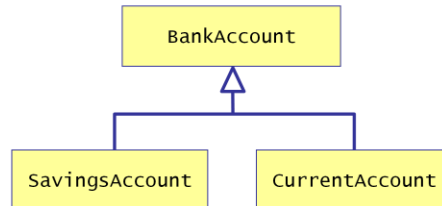
- This is when you notice semantic similarities between classes in your design.
- Factor the similarities into a common superclass.
- Define subclasses that extend the superclass in appropriate ways.

Another approach is specialization:

- This often occurs quite late during development, or if you are using a pre-provided class library or framework.
- You notice a class that is almost what you want, but not quite.
- Define a new class that extends the existing class as appropriate.

Sample Hierarchy

- We'll see how to implement the following hierarchy in this chapter:



- Note:
 - BankAccount defines common state and behaviour that is relevant for all kinds of account
 - SavingsAccount "is a kind of" BankAccount that earns interest
 - CurrentAccount "is a kind of" BankAccount that has cheques

9

For the purposes of our discussions, we'll show how to implement the simple inheritance hierarchy depicted in the slide. This diagram uses simple UML notation.

Note the following points:

- BankAccount is the superclass, and defines common members that are relevant for all kinds of bank account.
- SavingsAccount and CurrentAccount are two subclasses of BankAccount. They inherit everything from BankAccount (except constructors), and can add new members and override existing methods.

Superclass Considerations

- To define a superclass, just define a regular class ☺
 - Although there are some issues you need to consider...
- You can allow subclasses special access to your members
 - Designate members as "protected" (see later in this section)
- You can stop clients from creating instances of your class
 - Designate the class as "abstract" (see later in this chapter)
- You can defer a method implementation to subclasses
 - Designate the method as "abstract" (see later in this chapter)
- You can stop anyone from extending your class, or from overriding methods
 - Designate the class or methods as "final" (see later in this chapter)

10

When you define a class, you may or may not know whether you expect to define any subclasses in future. For example, sometimes you develop two separate classes in parallel, without necessarily realizing they are going to be very similar to each other. After a while, your understanding improves and you realize the potential for defining a common superclass from which the two classes can inherit. This is an example of generalization, which we mentioned a bit earlier.

If you do know in advance that a class will be inherited from, you can implement the class in such a way that it lends itself to subsequent inheritance. The slide lists some specific measures you can take in this respect.

Access Modifiers

- Java supports 4 access levels for members in a class...
- `public`
 - Accessible by anyone
 - Methods and constants are often `public`
- `private`
 - Accessible only by class itself
 - Data and helper methods are usually `private`
- `protected`
 - Accessible by class itself, subclasses, and classes in same package
 - Allow access to members that are hidden from general client code
- (No access modifier)
 - Accessible by class itself, and classes in same package



We mentioned these access modifiers earlier in the class, but it's worth reiterating the rules now in the context of our discussion about inheritance. In particular, note the `protected` access modifier, which means members can be accessed by a class or by any subclasses (or, bizarrely, any classes in the same package).

Defining a Superclass

- Here are the interesting parts of the BankAccount class
 - This is a simplification for now...

```
public class BankAccount {  
    // Instance data.  
    private String accountHolder;  
    private int id;  
    protected double balance = 0.0;  
    private Date creationTimestamp = new Date();  
  
    // Class data ...  
  
    // Constructors ...  
  
    // Instance methods ...  
  
    // Class methods ...  
}
```

BankAccount.java

- Note:
 - balance is protected, so it can be accessed by subclasses
 - All instance methods in Java are inherently "overridable" (unless marked as final – see later in this chapter)

This slide shows an embryonic definition of the BankAccount class. At this stage, the only telltale sign that suggests this class will be inherited from is the presence of the protected instance variable named balance.

The balance instance variable will be accessible to all the members in the BankAccount class, plus the methods in classes that subclass BankAccount, plus the methods in classes in the same package as BankAccount.

Note that all the other instance variables in the BankAccount class are declared private. In Java, private really does mean private; these instance variables are only visible to the BankAccount class itself. Subclasses will inherit these instance variables but they won't be able to access them directly; this preserves the internal encapsulation of the BankAccount class within its own inheritance hierarchy. Important stuff!

Defining a Subclass

- To define a subclass, use the **extends** keyword
 - Remember, a Java class can only inherit from one direct superclass

```
public class SavingsAccount extends BankAccount {  
    // Additional data and methods ...  
    // Constructor(s) ...  
    // Overrides for superclass methods, if necessary ...  
}
```

SavingsAccount.java

13

This slide shows how to define a subclass in Java. When you define a class, use the **extends** keyword and specify the name of the class you want to inherit from.

Note the following points:

- You can only extend a single class in Java.
- If you omit the **extends** clause, your class inherits directly from **Object** by default.

Adding New Members

- The subclass inherits everything from the superclass
 - (Except for constructors)
 - The subclass can define additional members if it needs to ...

```
public class SavingsAccount extends BankAccount {  
  
    private boolean premium;  
    private boolean goneOverdrawn;  
  
    private static final double BASIC_INTEREST_RATE = 0.015;  
    private static final double PREMIUM_INTEREST_RATE = 0.030;  
    private static final double GUARANTEED_LIMIT = 85000;  
  
    public void applyInterest() {  
        if (balance < 0) {  
            // Sorry mate, no interest if you're overdrawn.  
        }  
        else if (premium && !goneOverdrawn) {  
            balance *= (1 + PREMIUM_INTEREST_RATE);  
        }  
        else {  
            balance *= (1 + BASIC_INTEREST_RATE);  
        }  
        goneOverdrawn = false;  
    }  
    ...  
}
```

Additional instance vars

Additional class vars

Additional methods
(instance / class methods)

SavingsAccount.java

When you define a subclass, you automatically inherit all the members from the superclass (except constructors).

You can add whatever extra data members and methods you need in your subclass. This includes instance variables, instance methods, static variables, and static methods.

The code in the slide shows how the `SavingsAccount` extends `BankAccount` and adds various data members and methods. The `applyInterest()` method is allowed to access the `balance` variable, because it was declared as protected in the superclass.

Defining Constructors

- The subclass doesn't inherit constructors from superclass
 - So, define constructor(s) in subclass, to initialize subclass data
- The subclass constructor must invoke the superclass constructor, to initialize superclass data
 - To do this, call `super(params)` as the first statement in the subclass constructor

```
public class SavingsAccount extends BankAccount {  
    ...  
    public SavingsAccount(String accountHolder, boolean premium) {  
        super(accountHolder);  
        this.premium = premium;  
    }  
    ...  
}
```

Call superclass ctor
(otherwise, the compiler
will attempt to call
`super()` with no args)

SavingsAccount.java

15

The one thing you don't inherit from the superclass is the constructors. You must redefine constructors in your subclass, taking the full set of parameters that you deem necessary to fully initialize all the members in your subclass plus the superclass.

When the client creates a subclass object, Java calls the constructor defined in your subclass. The first thing the subclass constructor must do is to invoke a superclass constructor, to initialize the base part of the object. You do this via a call to `super()`, passing whatever parameters you need into the superclass constructor.

Note the following additional points:

- The call to `super()` must be the first statement in your subclass constructor.
- If you omit the call to `super()`, Java will attempt to call a no-argument constructor in the superclass on your behalf. If the superclass doesn't have a no-argument constructor, a compiler error occurs.

Overriding Methods (1 of 2)

- The subclass can override superclass instance methods
 - To provide a different (or supplementary) implementation
 - No obligation ☺
- If you do decide to override a method in a subclass:
 - The signature must match the superclass method signature
 - The return type must be the same (or a subclass – this is called a "covariant" return)
 - The access level must be the same (or less restrictive)
 - The list of checked exceptions must be narrower or fewer (it can't be broader or new checked exceptions)
- An override can call the original superclass method, to leverage existing functionality
 - Via `super.methodName(params)`

16

By default, all methods in a superclass are overridable. This means subclasses can re-implement methods with alternative application logic if appropriate.

Method overriding is a very common practice, especially when you're using a third-party framework. The framework will typically define a large number of classes in a predefined arrangement, to implement core functionality for a simple application. To use the framework, you typically define subclasses for some of the existing classes, and override some of the methods with your own application-specific functionality.

When you override a method from a superclass, you must follow some specific rules as described in the slide above.

Overriding Methods (2 of 2)

■ Examples of overriding methods:

```
public class SavingsAccount extends BankAccount {
    ...
    @Override ← Indicates we're overriding a superclass method
    public double withdraw(double amount) {
        super.withdraw(amount); ← Invoke superclass method
        if (balance < 0) {
            goneOverdrawn = true;
        }
        return balance;
    }

    @Override ← Indicates we're overriding a superclass method
    public String toString() {
        String str = String.format("%s [%s, %s]",
            super.toString(), ← Invoke superclass method
            premium ? "Premium" : "Normal",
            goneOverdrawn ? "gone overdrawn" : "not gone overdrawn");
        return str;
    }
    ...
}
```

SavingsAccount.java

17

This example shows how the `SavingsAccount` class can override some of the methods defined in the `BankAccount` class. Note the following points:

- The `withdraw()` method first calls the superclass version of `withdraw()`, to perform the normal processing as defined by the superclass. Afterwards, it checks whether the balance is now negative, and if so sets the `goneOverdrawn` flag.
- The `toString()` method calls the superclass version of `toString()` to get a textual representation of superclass state, and then appends some additional verbiage relating to specific state defined in the subclass.

3. Polymorphism

- What is polymorphism?
- The principle of substitutability
- Polymorphism in action
- Accessing subclass-specific members
- Polymorphic collections

18

Polymorphism basically relies on method overriding. When you have an inheritance hierarchy where classes implement some methods in distinct ways, polymorphism allows you to pass objects around in a general fashion, without ever needing to know exactly what type of object you're dealing with. Polymorphism ensures that the appropriate version of methods will always be called, based on the type of object you're dealing with.

What is Polymorphism?

- Greek for "many forms"
- In an OO context:
 - You can have many different "kinds" of object (e.g. many different kinds of bank accounts)
 - Your application can treat them all in the same way
 - Your application doesn't need to know which particular kind of object it's using at any given moment

19

The main goal of polymorphism is to simplify client code in the face of large and complex inheritance hierarchies.


In a nutshell, your client code can define methods that receive superclass parameter types, and you can then pass in objects of that type or any subclass type. The magic of polymorphism ensures that the system always knows what kind of object you've actually passed in, and thereby ensures the correct version of methods are called upon that object.

The Principle of Substitutability

- Polymorphism is facilitated by the principle of substitutability
 - A superclass reference can refer to any kind of subclass object

```
public static void demoPolymorphism() {  
    BankAccount sa = new SavingsAccount("Mickey", true);  
    BankAccount ca = new CurrentAccount("Donald", 50);  
  
    processAccount(sa);  
    processAccount(ca);  
}  
  
public static void processAccount(BankAccount account) {  
    ...  
}
```

UseAccounts.java



20

Polymorphism relies on the principle of substitutability, as illustrated in the slide above. Note the following points:

- The processAccount() method receives a superclass-type parameter, i.e. a BankAccount reference. Client code can pass in a BankAccount object, or anything that inherits from BankAccount.
- Within processAccount(), the compiler knows the account variable refers to a BankAccount object or some subclass, but it doesn't know which particular subclass. Therefore the compiler only lets you access members defined in BankAccount.
- The account variable doesn't allow any access to SavingsAccount-specific or CreditAccount-specific members. This is good – your code is more general purpose, so you won't have to modify your code if someone defines a new kind of BankAccount subclass in the future.

Polymorphism in Action

- Question:
 - What happens when you invoke an instance method via a superclass reference?
- Answer:
 - The "correct" version of the method is invoked, depending on the actual type of object currently pointed to
- Specifically, this is what polymorphism boils down to:
 - If the reference actually points to a subclass object...
 - And that subclass has overridden the method...
 - The subclass's version of the method is called

```
public static void processAccount(BankAccount account) {  
    account.withdraw(200);  
    account.deposit(300);  
    System.out.println(account.toString());  
}
```

UseAccounts.java

21

This slide describes exactly how polymorphism works in practice.

The crux of the matter is that the decision about which version of a method to call is deferred to run time, whereupon the JVM knows the actual type of object passed into a method on this occasion, and can therefore invoke the method defined in the appropriate subclass.

Accessing Subclass-Specific Members

- Polymorphism is great ☺
 - You can write general-purpose code that doesn't care what actual subclass objects it's dealing with
 - The downside is that you can't actually access subclass-specific members via a superclass reference
- If you really need to access subclass-specific members:
 - Use `instanceof` to determine if the reference points to a particular subclass object type
 - Then cast the reference to that subclass type

```
public static void processAccount(BankAccount account) {  
    ...  
    if (account instanceof SavingsAccount) {  
        SavingsAccount temp = (SavingsAccount)account;  
        temp.applyInterest();  
    }  
    ...  
}
```

UseAccounts.java

- Note: If the object reference is `null`, `instanceof` returns `false`

22

Java has an `instanceof` operator that enables you to determine whether an object reference points to a particular type of object. For example, the code snippet in the slide shows how the `processAccount()` method can determine whether it has been passed in a `SavingsAccount` object (as opposed to any other kind of object in the `BankAccount` hierarchy).

You are advised to use this technique very sparingly, because it results in fragile code that could easily break as new classes are added to the inheritance hierarchy during the lifetime of your development project.

Polymorphic Collections

- A polymorphic collection (or array) can hold different types of subclass objects
 - Achieved by specifying the superclass as the generic type parameter (or as the array type)
- Allows you to insert any type of subclass object into the collection/array
 - When you access items in the collection/array, you are working with superclass references

```
List<BankAccount> accounts = new ArrayList<BankAccount>();  
accounts.add(new SavingsAccount("Pluto", true));  
accounts.add(new CurrentAccount("Goofy", 50));  
...  
for (BankAccount account: accounts) {  
    processAccount(account);  
}
```

UseAccounts.java

23

Java (and other OO languages) supports the concept of polymorphic collections. To define a polymorphic collection or array, use the superclass in your declaration. For example, the `accounts` collection in the slide is a list of `BankAccount`. According to the principle of substitutability, you can insert any kind of bank account into this collection (i.e. a `BankAccount` object or anything that inherits from `BankAccount`).

You can then iterate through the items in the collection. On each iteration you'll obtain a `BankAccount` object reference, and you can invoke any `BankAccount` methods upon the object reference. According to the principle of polymorphism, this will result in the appropriate subclass implementation of the method being executed at run time.

4. Additional Techniques

- abstract classes
- abstract methods
- final classes
- final methods

24

To finish this chapter, we take a look at abstract classes/methods and final classes/methods. `abstract` and `final` are keywords in Java.

abstract Classes

- When you define an inheritance hierarchy, you often find that the superclass is "incomplete"
 - It contains common members for its subclasses, but it doesn't have enough know-how to represent a real object
- In such cases, declare the superclass as `abstract`
 - The compiler will prevent any instances of the superclass ever being created

```
public abstract class BankAccount {  
    ...  
}
```

BankAccount.java

25

If you declare a class as `abstract`, it prevents instances of that class from ever being created. This is a useful technique for superclasses, which often exist purely to house the common set of members required by all subclasses.

Note that an `abstract` class can look and feel just like any other class. The only difference is that you can't instantiate objects.

abstract Methods

- When you define an inheritance hierarchy, the superclass must list all the methods that will be available to the client
 - The problem is, the superclass might not know how to actually implement some of these methods
 - For example, the implementation details might vary completely across all the subclasses
- In such cases, declare the method as **abstract**
 - The superclass does not provide a method body
 - Instead, each subclass is obliged to implement the method

```
public abstract class BankAccount {  
    ...  
    public abstract String getTermsAndConditions();  
    public abstract double getGuaranteedLimit();  
}  
  
public class SavingsAccount extends BankAccount {  
    ...  
    public String getTermsAndConditions() { ... }  
    public double getGuaranteedLimit() { ... }  
}
```

If you have an abstract class, you can define abstract methods in the class. An abstract method doesn't have a method body - you just terminate it immediately with a semicolon. Note that you can only define abstract methods in abstract classes.

The purpose of abstract methods is to define placeholders for functionality that all subclasses are obliged to implement. If a subclass doesn't implement an abstract method, then you'll get a compiler error (unless you declare the subclass as abstract too).

final Classes

- If you want to prevent any further classes from extending your class...
 - Define it as a `final` class
 - This is very bold step...

```
public final class SavingsAccount extends BankAccount {  
    ...  
}
```

27

If you define a class as `final`, it means you can never define any subclasses that inherit from this class. This is quite a strong statement of intent, but it is relevant for some special classes such as `String`, `Console`, etc.

final Methods

- If you want to prevent any subclass from overriding a particular method...
 - Define it as a `final` method
 - Disables polymorphism on that method
 - Why might this be a useful thing to do...?

```
public abstract class BankAccount {  
    ...  
    public final int getId() {  
        return id;  
    }  
  
    public final double getBalance() {  
        return balance;  
    }  
  
    public final Date getCreationTimestamp() {  
        return creationTimestamp;  
    }  
    ...  
}
```

BankAccount.java

28

A `final` method is one that cannot be overridden by subclasses. This means the method is not polymorphic.

Defining a method as `final` is an optimization mechanism. The compiler knows the method will never be overridden, so it can resolve the method call at compile time rather than having to defer to run time to see what kind of object is involved.

Any Questions?



29