gencat
www.gencat.net

Cicles Formatius de Grau Superior
Desenvolupament d'Aplicacions Informàtiques

D06
PROGRAMMING with JAVA

Ch13 – Inheritance

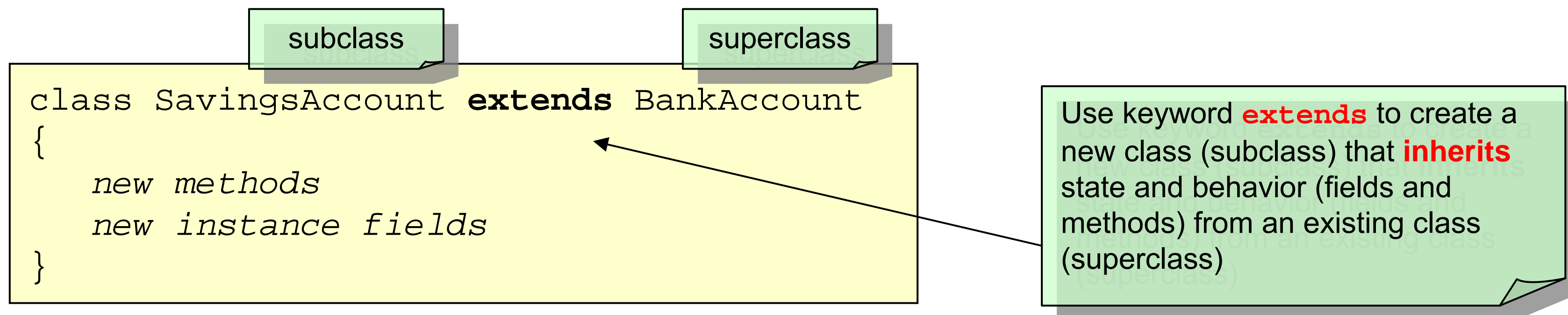
PowerPoint presentation, created by Angel A. Juan - ajuanp(@)gmail.com,
for accompanying the book "Big Java", by Cay S. Horstmann (2006)

Chapter Goals

- To learn about inheritance
- To understand how to inherit and override superclass methods
- To be able to invoke superclass constructors
- To learn about `package` access control
- To understand the common superclass `Object` and to override its `toString()` and `equals()` methods

An Introduction to Inheritance

- Inheritance is a mechanism for enhancing existing classes. If you need to implement a new class and a class representing a more general concept is already available, then the new class can inherit from the existing class
- Example: Savings account = bank account with interest:



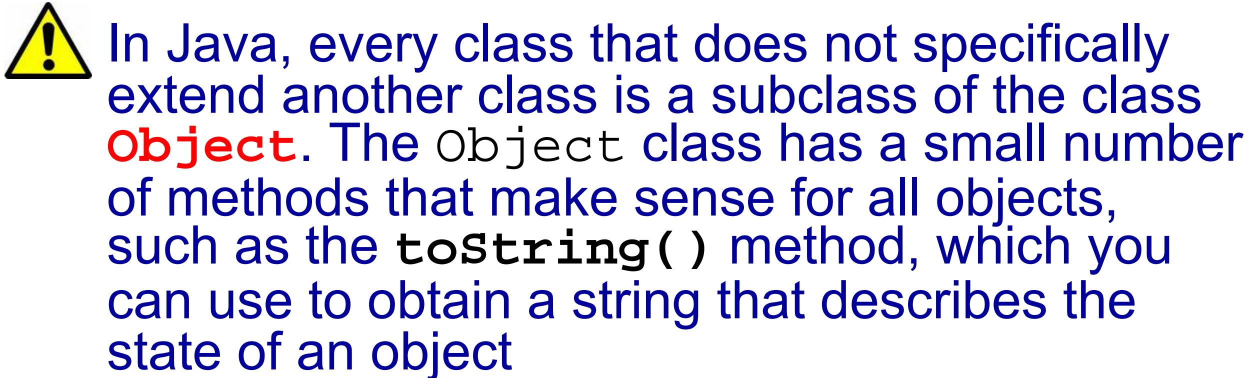
The `SavingsAccount` class automatically inherits all methods and instance fields of the `BankAccount` class (e.g., the `deposit` method automatically applies to savings accounts):

```
SavingsAccount collegeFund = new SavingsAccount(10);
// Savings account with 10% interest
collegeFund.deposit(500);
// OK to use BankAccount method with SavingsAccount object
```

Continued…

An Introduction to Inheritance

- The more general class that forms the basis for inheritance (extended class) is called the superclass. The more specialized class that inherits from the superclass (extending class) is called the subclass



In Java, every class that does not specifically extend another class is a subclass of the class `Object`. The `Object` class has a small number of methods that make sense for all objects, such as the **`toString()`** method, which you can use to obtain a string that describes the state of an object

- In a class diagram, you denote inheritance by a solid arrow with a “hollow triangle” tip that points to the superclass

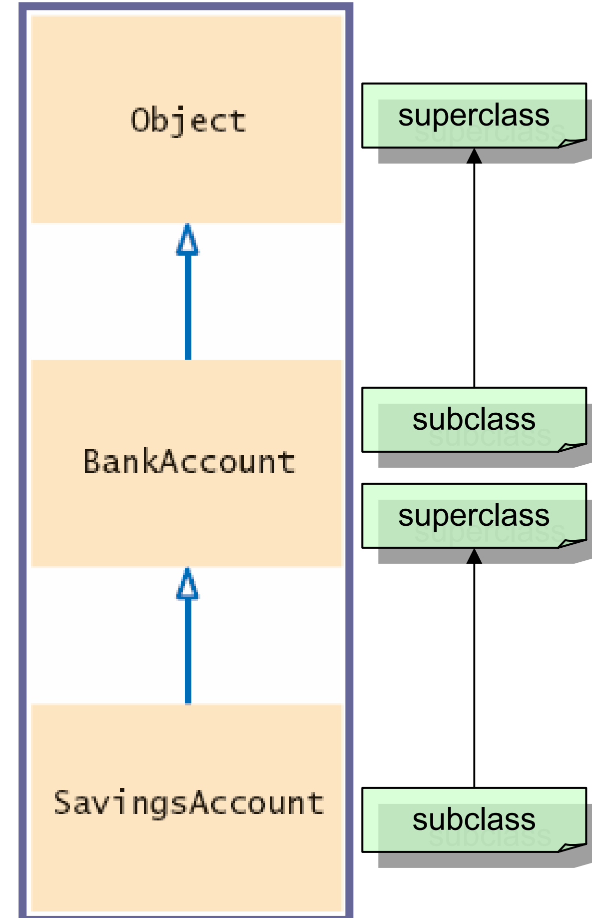


Figure 1:
An Inheritance Diagram

Continued…

An Introduction to Inheritance

Inheriting from class ≠ implementing interface: An interface is not a class. It has no state and no behavior. It merely tells you which methods you should implement. A superclass has state and behavior, and the subclasses inherit them

- One important reason for inheritance is code reuse. By inheriting an existing class, you do not have to replicate the effort that went into designing and perfecting that class
- In the subclass, specify added instance fields, added methods, and changed or overridden methods:

```
public class SavingsAccount extends BankAccount
{
   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      deposit(interest);
   }

   private double interestRate;
}
```

Continued…

An Introduction to Inheritance

- Note the following:
  1. The `addInterest()` method calls the `getBalance()` and `deposit()` methods rather than directly updating the `balance` field of the superclass. This is a consequence of **encapsulation**: the `balance` field was defined as `private` in the `BankAccount` class. The `addInterest()` method is defined in the `SavingsAccount` class → it does not have the right to access a private field of another class
  2. The `addInterest()` method calls the `getBalance()` and `deposit()` methods of the superclass without specifying an implicit parameter. This means that the calls apply to the same object, i.e., the implicit parameter of the `addInterest()` method
  3. A `SavingsAccount` object inherits the `balance` instance field from the `BankAccount` superclass, and gains one additional instance field, `interestRate`:

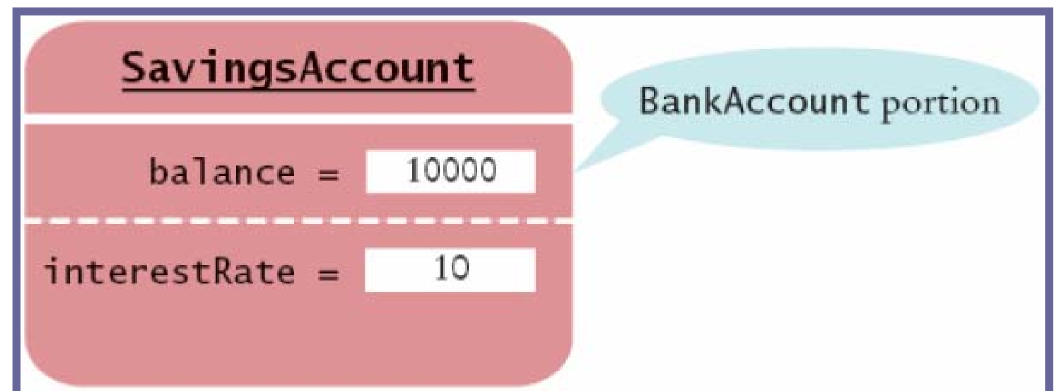


Figure 2:
Layout of a Subclass Object

Syntax 13.1: Inheritance

```
class SubclassName extends SuperclassName
{
   methods
   instance fields
}
```

Example:

```
public class SavingsAccount extends BankAccount
{
   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }

   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      deposit(interest);
   }
   private double interestRate;
}
```

Purpose:

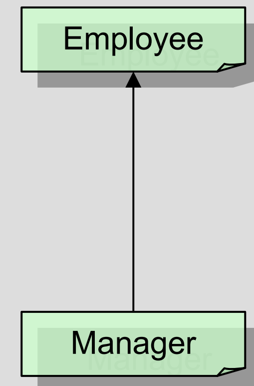
To define a new class that inherits from an existing class, and define the methods and instance fields that are added in the new class.

Self Check

1. Which instance fields does an object of class `SavingsAccount` have?
2. Name four methods that you can apply to `SavingsAccount` objects
3. If the class `Manager` extends the class `Employee`, which class is the superclass and which is the subclass?

Answers

1. Two instance fields: `balance` and `interestRate`
2. `deposit()`, `withdraw()`, `getBalance()`, and `addInterest()`
3. `Manager` is the subclass; `Employee` is the superclass.



Inheritance Hierarchies

- In Java it is common to group classes in complex inheritance hierarchies. The classes representing the most general concepts are near the root, more specialized classes towards the branches
- Example: Figure 4 shows part of the hierarchy of `Swing` user interface components in Java

When designing a hierarchy of classes, you ask yourself which features and behaviors are common to all the classes that you are designing. Those **common properties** are collected in a **superclass**. More specialized properties can be found in subclasses

The **javax.swing** package provides a set of graphical components such as push buttons and text fields that work the same on all platforms

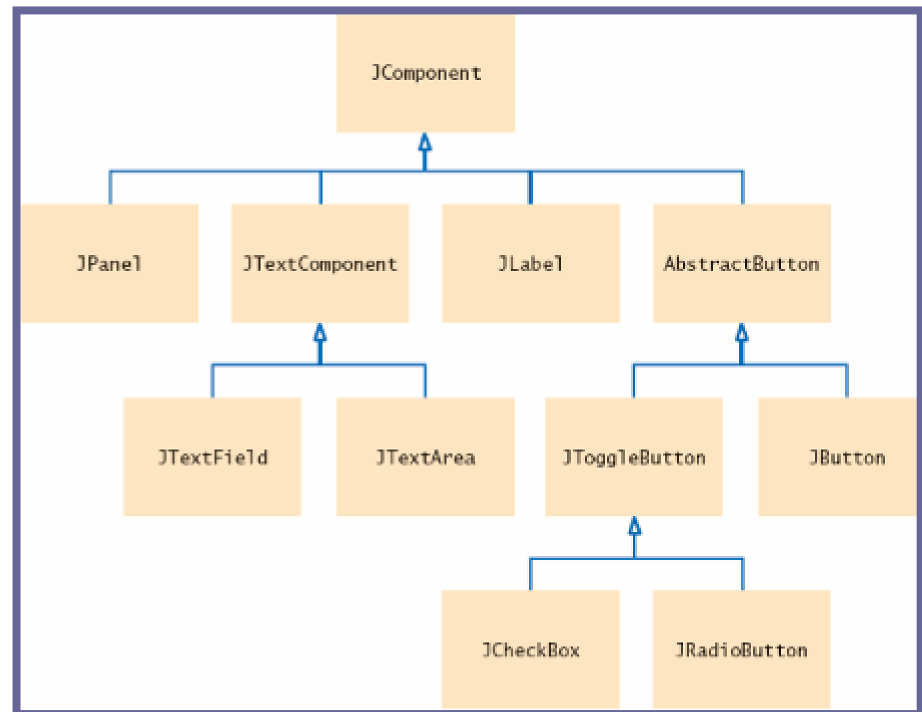


Figure 4:
A Part of the Hierarchy of Swing User Interface Components

A Simpler Hierarchy: Hierarchy of Bank Accounts

- Consider a bank that offers its customers the following account types:
  - Checking account: no interest; small number of free transactions per month, additional transactions are charged a small fee
  - Savings account: earns interest that compounds monthly
- All bank accounts support the `getBalance()` method. They also support the `deposit()` and `withdraw()` methods, although the details of the implementation differ
- The checking account needs a method `deductFees()` to deduct the monthly fees and to reset the transaction counter. The `deposit()` and `withdraw()` methods must be redefined to count the transactions. The savings account needs a method `addInterest()` to add interest

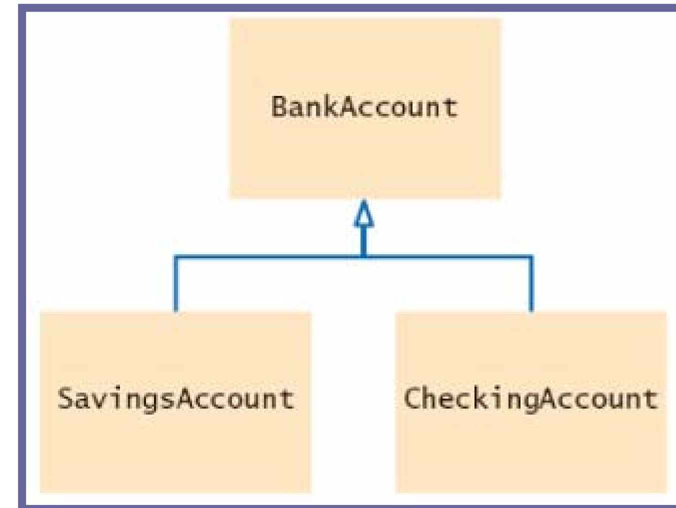


Figure 5:
Inheritance Hierarchy for Bank Account Classes

The subclasses support all methods from the superclass, but their implementation may be **modified** to match the specialized purposes of the subclasses. In addition, subclasses are free to introduce **additional** methods.

Self Check

4. What is the purpose of the `JTextComponent` class in Figure 4?
5. Which instance field will we need to add to the `CheckingAccount` class?

Answers

4. To express the common behavior of text fields and text components.
5. We need a counter that counts the number of withdrawals and deposits.

Inheriting Methods

- When you form a subclass of a given class, you can specify additional instance fields and methods. When defining the methods for a subclass, there are three possibilities:
   1. You can **override methods** from the superclass: If you specify a method with the same signature (same name and parameter types), it overrides the method of the same name in the superclass. Whenever the method is applied to an object of the subclass type, the overriding method, and not the original method, is executed
   2. You can **inherit methods** from the superclass: If you do not explicitly override a superclass method, you automatically inherit it. The superclass method can be applied to the subclass objects
   3. You can **define new methods**: If you define a method that did not exist in the superclass, then the new method can be applied only to subclass objects

Inheriting Instance Fields

The situation for instance fields is quite different. You can never override instance fields. For fields in a subclass, there are only two cases:

1. The subclass **inherits all fields** from the superclass: All instance fields from the superclass are automatically inherited
2. Any **new instance fields** that you define in the subclass are present only in subclass objects

- What happens if you define a new field with the same name as a superclass field? This is legal but extremely undesirable

Implementing the `CheckingAccount` Class

- Overrides `deposit()` and `withdraw()` to increment the transaction count:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount) {. . .}
   public void withdraw(double amount) {. . .}
   public void deductFees() {. . .} // new method
   private int transactionCount;    // new instance field
}
```

- Each `CheckingAccount` object has two instance fields:
  - `balance` (inherited from `BankAccount`)
  - `transactionCount` (new to `CheckingAccount`)
- You can apply four methods to `CheckingAccount` objects:
  - `getBalance()` (inherited from `BankAccount`)
  - `deposit(double amount)` (overrides `BankAccount` method)
  - `withdraw(double amount)` (overrides `BankAccount` method)
  - `deductFees()` (new to `CheckingAccount`)

Inherited Fields Are Private

- Consider the `deposit()` method of the `CheckingAccount` class:

```
public void deposit(double amount)
{
   transactionCount++;
   // now add amount to balance
   . . .
}
```

- Now we have a problem. We can't just add `amount` to `balance` since `balance` is a private field of the superclass

Subclass methods have no more access rights to the private data of the superclass than any other methods. If you want to modify a private superclass field, you must use a public method of the superclass

Invoking a Superclass Method

- To invoke another method on the implicit parameter, you don't specify the parameter but simply write the method name:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Now add amount to balance
      deposit(amount); // Wrong! use super.deposit(amount)
   }
}
```

- But this won't work since the compiler interprets `deposit(amount)` as `this.deposit(amount)` (i.e., the `deposit()` method in the subclass) → it calls the same method again and again (infinite recursion)

Instead, we must be specific that we want to invoke only the superclass's `deposit()` method using the special keyword **`super`**:
`super``.deposit(amount);` (now it calls the `deposit()` method of the `BankAccount` class)

Syntax 13.2: Calling a Superclass Method

```
super.methodName(parameters)
```

Example:

```
 public void deposit(double amount)
{
   transactionCount++;
   super.deposit(amount);
}
```

Purpose:

To call a method of the superclass instead of the method of the current class

Implementing Remaining Methods

```
public class CheckingAccount extends BankAccount
{
   . . .
   public void withdraw(double amount)
   {
      transactionCount++;
      // Now subtract amount from balance
      super.withdraw(amount);
   }
```

```
   public void deductFees()
   {
      if (transactionCount > FREE_TRANSACTIONS)
      {
         double fees = TRANSACTION_FEE
            * (transactionCount - FREE_TRANSACTIONS);
         super.withdraw(fees);
      }
      transactionCount = 0;
   }
   . . .
   private static final int FREE_TRANSACTIONS = 3;
   private static final double TRANSACTION_FEE = 2.0;
}
```

Self Check

6. Why does the `withdraw()` method of the `CheckingAccount` class call `super.withdraw()`?
7. Why does the `deductFees()` method set the transaction count to zero?

Answers

6. It needs to reduce the balance, and it cannot access the `balance` field directly.
7. So that the count can reflect the number of transactions for the following month.

Common Error: Shadowing Instance Fields

- A subclass has no access to the private instance fields of the superclass
- Beginner's error: to "solve" this problem by adding another instance field with the same name:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      balance = balance + amount;
   }
   . . .
   private double balance; // Don't do this!
}
```

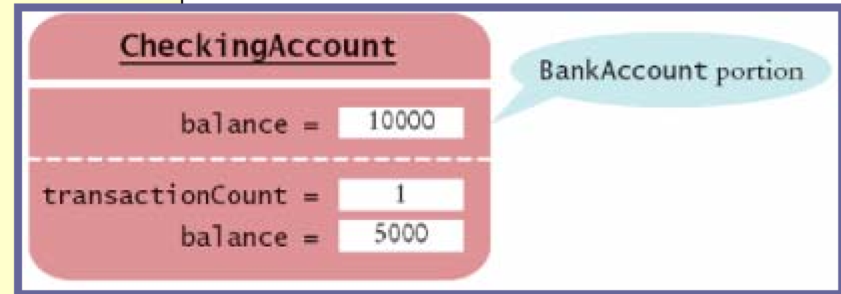


Figure 6:
Shadowing Instance Fields

- Now the `deposit()` method compiles, but it doesn't update the correct balance! Such a `CheckingAccount` object has two instance fields, both named `balance`. The `getBalance()` method of the superclass retrieves one of them, and the `deposit()` method of the subclass updates the other

Common Error: Failing to Invoke the Superclass Method

A common error in extending the functionality of a superclass method is to forget the **`super`** qualifier:

```
public class CheckingAccount extends BankAccount
{
   public void withdraw(double amount)
   {
      transactionCount++;
      withdraw(amount);
      // Error-should be super.withdraw(amount);
   }
   . . .
}
```

- Here `withdraw(amount)` refers to the `withdraw()` method applied to the implicit parameter of the method. The implicit parameter is of type `CheckingAccount`, and the `CheckingAccount` class has a `withdraw()` method, so that method is called. Of course, that calls the current method all over again, which will call itself yet again, over and over, until the program runs out of memory

Subclass Construction

- There is a special instruction to call the superclass constructor from a subclass constructor: use the keyword **super()**, followed by the construction parameters in parentheses:

```
public class CheckingAccount extends BankAccount
{
   public CheckingAccount(double initialBalance)
   {
      // Construct superclass
      super(initialBalance);
      // Initialize transaction count
      transactionCount = 0;
   }
}
```

When the keyword **super()** is followed by a parenthesis, it indicates a call to the superclass constructor. When it is used this way, the constructor call must be the first statement of the subclass constructor

- If **super** is followed by a period and a method name, it indicates a call to a superclass method (such a call can be made anywhere in any subclass method)

Continued…

Subclass Construction

- The dual use of the `super` keyword is analogous to the dual use of the `this` keyword
- If a subclass constructor does not call the superclass constructor, the superclass is constructed with its default constructor. However, if all constructor of the superclass require parameters, then the compiler reports an error
- Most commonly, however, subclass constructors have some parameters that they pass on to the superclass and others that they use to initialize subclass fields

Syntax 13.1: Calling a Superclass Constructor

```
ClassName(parameters)
{
   super(parameters);
   . . .
}
```

Example:

```
public CheckingAccount(double initialBalance)
{
      super(initialBalance);
      transactionCount = 0;
}
```

Purpose:

To invoke a constructor of the superclass. Note that this statement must be the first statement of the subclass constructor.

Self Check

8. Why didn't the `SavingsAccount()` constructor call its superclass constructor?
9. When you invoke a superclass method with the `super` keyword, does the call have to be the first statement of the subclass method?

Answers

8. It was content to use the default constructor of the superclass, which sets the balance to zero.
9. No–this is a requirement only for constructors. For example, the `SavingsAccount.deposit` method first increments the transaction count, then calls the superclass method.

Converting Between Subclass and Superclass Types

- It is often necessary to convert a subclass type to a superclass type. Occasionally, you need to carry out the conversion in the opposite direction

It is OK to convert subclass reference to superclass reference:

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund;
Object anObject = collegeFund;
```

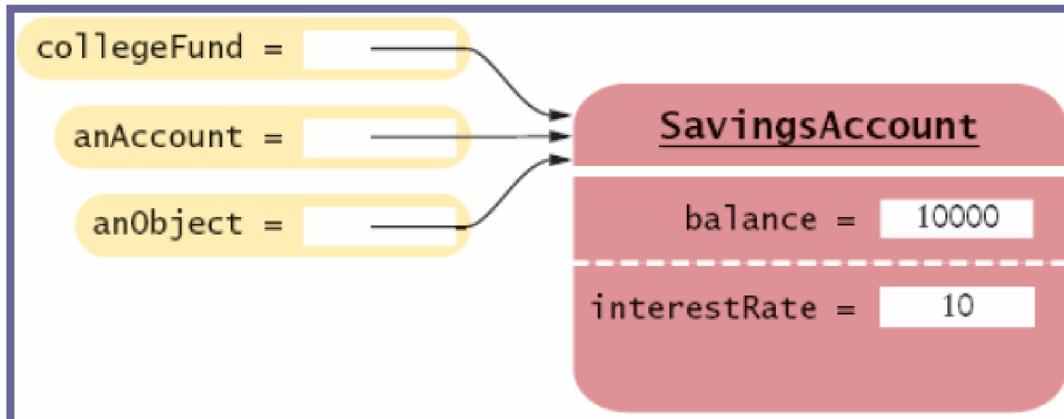


Figure 7:
Variables of Different Types Refer to the Same Object

Continued…

Converting Between Subclass and Superclass Types

Superclass references don't know the full story:

```
anAccount.deposit(1000); // OK
anAccount.addInterest();
// No--not a method of the class to which anAccount belongs
```

- When you convert between a subclass object to its superclass type:
  - The value of the reference stays the same–it is the memory location of the object
  - But, less information is known about the object
- Why would anyone want to know less about an object and store a reference in an object field of a superclass? This can happen if you want to reuse code that knows about the superclass but not the subclass

Converting Between Subclass and Superclass Types

- Very occasionally, you need to convert from a superclass reference to a subclass reference, for example:

```
BankAccount anAccount = (BankAccount) anObject;
```

- However, this cast is somewhat dangerous: if you are wrong, and `anObject` actually refers to an object of an unrelated type, then an exception is thrown

To protect against bad casts, you can use the **`instanceof`** operator. It tests whether an object belongs to a particular type:

```
if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   . . .
}
```

Syntax 13.4: The `InstanceOf` Operator

```
object instanceof TypeName
```

Example:

```
 if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   . . .
}
```

Purpose:

To return `true` if the *object* is an instance of *`TypeName`* (or one of its subtypes), and `false` otherwise

Self Test

10. Why did the second parameter of the `transfer()` method have to be of type `BankAccount` and not, for example, `SavingsAccount`?
11. Why can't we change the second parameter of the `transfer()` method to the type `Object`?

Answers

10. We want to use the method for all kinds of bank accounts. Had we used a parameter of type `SavingsAccount`, we couldn't have called the method with a `CheckingAccount` object.

11. We cannot invoke the `deposit()` method on a variable of type `Object`.

Polymorphism

In Java, the type of a variable does not completely determine the type of the object to which it refers. For example, a variable of type `BankAccount` can hold a reference to an actual `BankAccount` object or a subclass object such as `SavingsAccount` (you already encountered this phenomenon with variables whose type was an interface)

```
BankAccount aBankAccount = new SavingsAccount(1000);
// aBankAccount holds a reference to a SavingsAccount
```

What happens when you invoke a method?

```
BankAccount anAccount = new CheckingAccount();
anAccount.deposit(1000);
// Calls "deposit" from CheckingAccount
```

In Java, method calls are always determined by the type of the actual object, not the type of the object reference. The ability to refer to objects of multiple types with varying behavior is called polymorphism

Continued…

Polymorphism

- If polymorphism is so powerful, why not store all account references in variables of type `Object`?

  This does not work because the compiler needs to check that only legal methods are invoked (the `Object` type does not define a `deposit()` method –the `BankAccount` type (at least) is required to make a call to the `deposit()` method

```
Object anObject = new BankAccount();
anObject.deposit(1000); // Wrong!
```

Polymorphism

- Polymorphism: ability to refer to objects of multiple types with varying behavior
- Polymorphism at work:

```
public void transfer(double amount, BankAccount other)
{
   withdraw(amount); // Shortcut for this.withdraw(amount)
   other.deposit(amount);
}
```

Depending on types of `amount` and `other`, different versions of `withdraw()` and `deposit()` are called

File AccountTester.java

```
01: /**
02:    This program tests the BankAccount class and
03:    its subclasses.
04: */
05: public class AccountTester
06: {
07:    public static void main(String[] args)
08:    {
09:       SavingsAccount momsSavings
10:             = new SavingsAccount(0.5);
11:
12:       CheckingAccount harrysChecking
13:             = new CheckingAccount(100);
14:
15:       momsSavings.deposit(10000);
16:
```

Continued…

File `AccountTester.java`

```
17:         momsSavings.transfer(2000, harrysChecking);
18:         harrysChecking.withdraw(1500);
19:         harrysChecking.withdraw(80);
20:
21:         momsSavings.transfer(1000, harrysChecking);
22:         harrysChecking.withdraw(400);
23:
24:         // Simulate end of month
25:         momsSavings.addInterest();
26:         harrysChecking.deductFees();
27:
28:         System.out.println("Mom's savings balance = $"
29:               + momsSavings.getBalance());
30:
31:         System.out.println("Harry's checking balance = $"
32:               + harrysChecking.getBalance());
33:     }
34: }
```

File BankAccount.java

```
01: /**
02:    A bank account has a balance that can be changed by
03:    deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:    /**
08:       Constructs a bank account with a zero balance.
09:    */
10:    public BankAccount()
11:    {
12:       balance = 0;
13:    }
14:
15:    /**
16:       Constructs a bank account with a given balance.
17:       @param initialBalance the initial balance
18:    */
```

Continued…

File BankAccount.java

```
19:     public BankAccount(double initialBalance)
20:     {
21:        balance = initialBalance;
22:     }
23:
24:     /**
25:        Deposits money into the bank account.
26:        @param amount the amount to deposit
27:     */
28:     public void deposit(double amount)
29:     {
30:        balance = balance + amount;
31:     }
32:
33:     /**
34:        Withdraws money from the bank account.
35:        @param amount the amount to withdraw
36:     */
```

Continued…

File BankAccount.java

```
37:     public void withdraw(double amount)
38:     {
39:        balance = balance - amount;
40:     }
41:
42:     /**
43:        Gets the current balance of the bank account.
44:        @return the current balance
45:     */
46:     public double getBalance()
47:     {
48:        return balance;
49:     }
50:
51:     /**
52:        Transfers money from the bank account to another account
53:        @param amount the amount to transfer
54:        @param other the other account
55:     */
```

Continued…

File BankAccount.java

```
56:     public void transfer(double amount, BankAccount other)
57:     {
58:        withdraw(amount);
59:        other.deposit(amount);
60:     }
61:
62:     private double balance;
63: }
```

File CheckingAccount.java

```
01: /**
02:    A checking account that charges transaction fees.
03: */
04: public class CheckingAccount extends BankAccount
05: {
06:    /**
07:       Constructs a checking account with a given balance.
08:       @param initialBalance the initial balance
09:    */
10:    public CheckingAccount(double initialBalance)
11:    {
12:       // Construct superclass
13:       super(initialBalance);
14:
15:       // Initialize transaction count
16:       transactionCount = 0;
17:    }
18:
```

Continued…

File CheckingAccount.java

```
19:     public void deposit(double amount)
20:     {
21:        transactionCount++;
22:        // Now add amount to balance
23:        super.deposit(amount);
24:     }
25: 
26:     public void withdraw(double amount)
27:     {
28:        transactionCount++;
29:        // Now subtract amount from balance
30:        super.withdraw(amount);
31:     }
32: 
33:     /**
34:        Deducts the accumulated fees and resets the
35:        transaction count.
36:     */
```

Continued…

File CheckingAccount.java

```
37:    public void deductFees()
38:    {
39:       if (transactionCount > FREE_TRANSACTIONS)
40:       {
41:          double fees = TRANSACTION_FEE *
42:                (transactionCount - FREE_TRANSACTIONS);
43:          super.withdraw(fees);
44:       }
45:       transactionCount = 0;
46:    }
47:
48:    private int transactionCount;
49:
50:    private static final int FREE_TRANSACTIONS = 3;
51:    private static final double TRANSACTION_FEE = 2.0;
52: }
```

File SavingsAccount.java

```
01: /**
02:    An account that earns interest at a fixed rate.
03: */
04: public class SavingsAccount extends BankAccount
05: {
06:    /**
07:       Constructs a bank account with a given interest rate.
08:       @param rate the interest rate
09:    */
10:    public SavingsAccount(double rate)
11:    {
12:       interestRate = rate;
13:    }
14:
15:    /**
16:       Adds the earned interest to the account balance.
17:    */
```

Continued…

File SavingsAccount.java

```
18:     public void addInterest()
19:     {
20:         double interest = getBalance() * interestRate / 100;
21:         deposit(interest);
22:     }
23:
24:     private double interestRate;
25: }
```

Output:

```
Mom's savings balance = $7035.0
Harry's checking balance = $1116.0
```

Abstract Classes

- When you extend an existing class, you can redefine the methods of the superclass. Sometimes, it is desirable to force programmers to redefine a method (for instance, when there is no good default for the superclass, and only the subclass programmer can know how to implement the method properly). You can declare an abstract method with the keyword **`abstract`**:

```
public abstract class BankAccount
{
   public abstract void deductFeeees() {...}
}
```

- An abstract method has no implementation. This forces the implementers of subclasses to specify concrete implementation of this method

You cannot construct objects of classes with abstract methods. A class for which you cannot create objects is called an abstract class (as opposite to concrete class). In Java, you must declare all abstract classes with the keyword **`abstract`**

Continued…

Abstract Classes

- A class that defines an abstract method, or that inherits an abstract method without overriding it, must be declared as an abstract class
- You can also declare classes with no abstract methods as abstract. Doing so prevents programmers from creating instances of that class but allows them to create their own subclasses

Note that you cannot construct an object of an abstract class, but you can still have an object reference whose type is an abstract class. Of course, the actual object to which it refers must be an instance of a concrete subclass

- The reason for using abstract classes is to force programmers to create subclasses. By specifying certain methods as abstract, you avoid the trouble of coming up with useless default methods that others might inherit by accident

Abstract classes differ from interfaces in an important way –they can have instance fields, and they can have concrete methods and constructors

Final methods and Classes

- Occasionally, you may want to prevent other programmers from creating subclasses or from overriding certain methods. In these situations, you use the **`final`** keyword
- Example: the `String` class in the standard Java library has been declared as:

```
public final class String {...}
```

That means that nobody can extend the `String` class

The `String` class is meant to be immutable –string objects can't be modified by any of their methods. Nobody can create subclasses of `String`; therefore, you know that all `String` references can be copied without the risk of mutation

You can also declare individual methods as `final`. This way, nobody can override them

```
public final boolean checkPassword(String pass)
```

Self Check

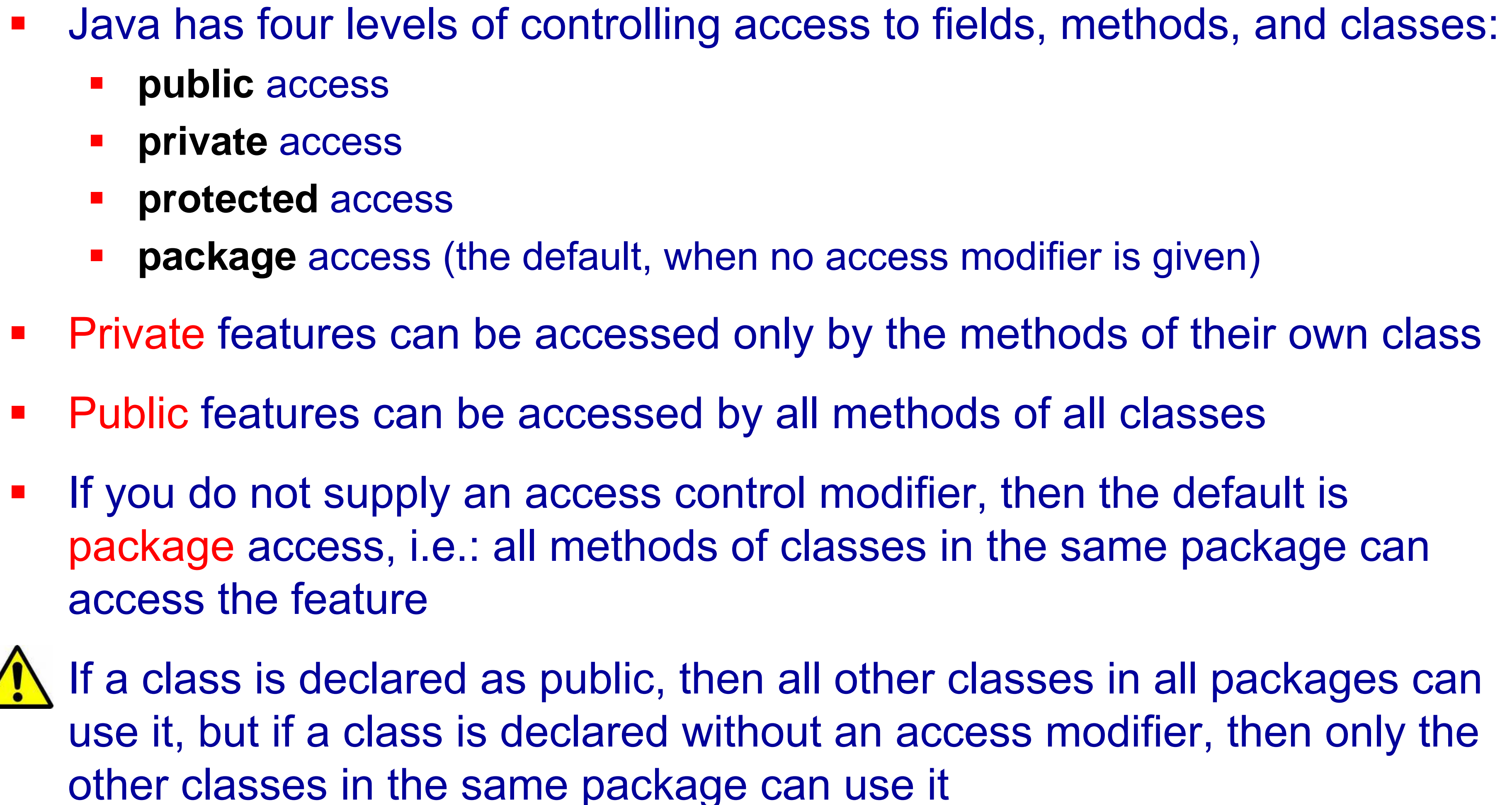
12. If a is a variable of type `BankAccount` that holds a non-null reference, what do you know about the object to which a refers?
13. If a refers to a checking account, what is the effect of calling `a.transfer(1000, a)`?

Answers

12. The object is an instance of `BankAccount` or one of its subclasses.
13. The balance of a is unchanged, and the transaction count is incremented twice.

Access Control

- Java has four levels of controlling access to fields, methods, and classes:
  - **public** access
  - **private** access
  - **protected** access
  - **package** access (the default, when no access modifier is given)
- Private features can be accessed only by the methods of their own class
- Public features can be accessed by all methods of all classes
- If you do not supply an access control modifier, then the default is package access, i.e.: all methods of classes in the same package can access the feature



If a class is declared as public, then all other classes in all packages can use it, but if a class is declared without an access modifier, then only the other classes in the same package can use it

Continued…

Access Control

- Package access is a good default for classes, but it is extremely unfortunate for fields. Instance and static fields of classes should always be **`private`**. There are a few exceptions:
  - Public constants (`public static final` fields) are useful and safe
  - Some objects, such as `System.out`, need to be accessible to all programs and therefore should be public
  - Very occasionally, several classes in a package must collaborate very closely. In that case, it may make sense to give some fields package access. But inner classes are usually a better solution

Package access for fields is rarely useful, and most fields are given package access by accident because the programmer simply forgot the `private` keyword, thereby opening up a potential security hole

Continued…

Access Control

- Methods should generally be **`public`** or **`private`**. If a superclass declares a method to be publicly accessible, you cannot override it to be more private
- Classes and interfaces can have `public` or package access. Classes that are generally useful should have **`public`** access. Classes that are used for implementation reasons should have package access (you can hide them even better by turning them into inner classes)

Self Check

14. What is a common reason for defining package-visible instance fields?
15. If a class with a public constructor has package access, who can construct objects of it?

Answers

14. Accidentally forgetting the `private` modifier.
15. Any methods of classes in the same package.

Object: The Cosmic Superclass

- In Java, every class that is defined without an explicit `extends` clause automatically extends the class **`Object`**, i.e.: the class `Object` is the direct or indirect superclass of every class in Java

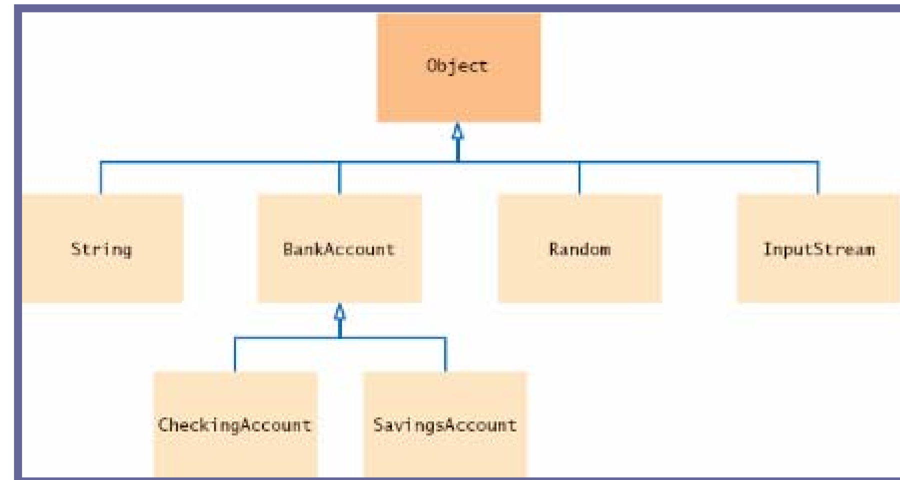


Figure 8:
The `Object` Class is the Superclass of Every Java Class

- The methods of the `Object` class are very general, e.g.:
  - `String` **`toString()`** → **Returns a string representation of the object**
  - `boolean` **`equals(`**`Object otherObject`**`)`** → **Tests whether the object equals another object**
  - `Object` **`clone()`** → **Makes a full copy of an object**
- It is a good idea to override these methods in your classes

Overriding the `toString` Method

- The **`toString()`** method returns a string representation for each object. It is used for debugging:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
// Sets s to "java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

- In fact, this `toString` method is called whenever you concatenate a string with an object:

```
"box=" + box;
// Result: "box=java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

- `Object.toString` prints class name and the hash code of the object (the hash code can be used to tell objects apart –different objects are likely to have different hash codes):

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to something like "BankAccount@d24606bf"
```

Continued…

Overriding the `toString` Method

- The `toString()` method ob the `Object` class provides the hash code because it does not know what is inside the `BankAccount` class. To provide a nicer representation of an object, override `toString()`:

```
public class BankAccount
{
   public String toString()
   {
      return "BankAccount[balance=" + balance + "]";
   }
}
```

- This works better:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to "BankAccount[balance=5000]"
```

Overriding the `equals` Method

- The `equals()` method is called whenever you want to compare whether two objects have the same contents

This is different from the test with the `==` operator, which tests whether the two references are to the same object

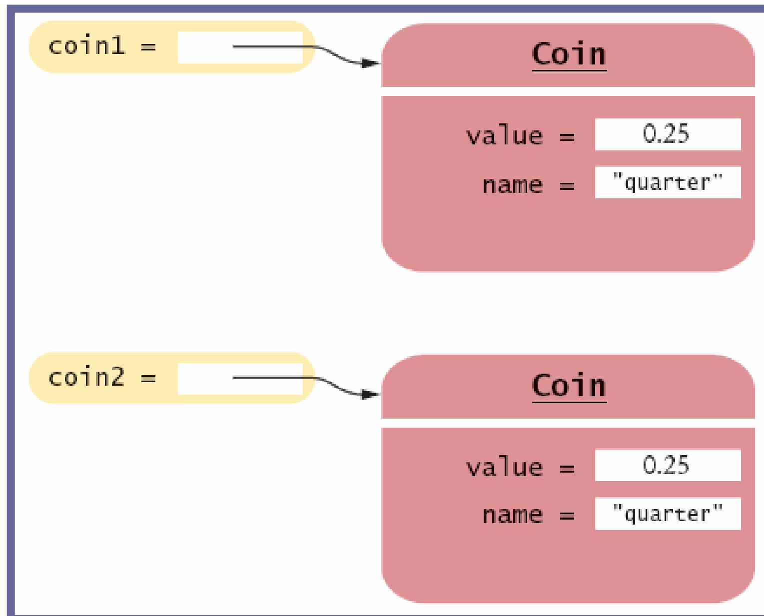


Figure 9:
Two References to Equal Objects

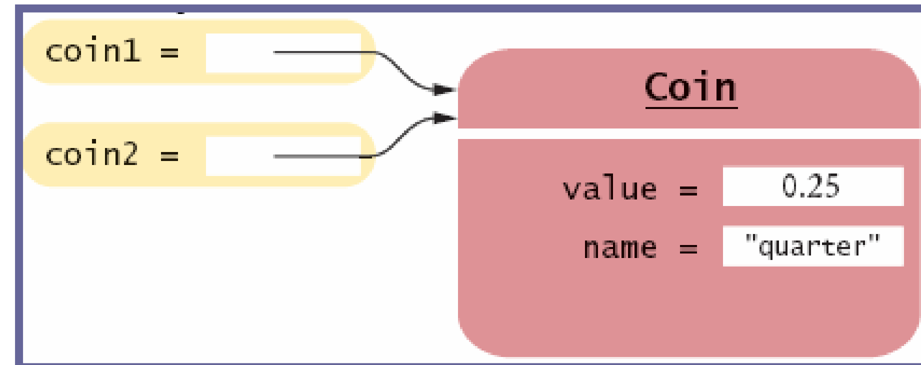


Figure 10:
Two References to the Same Object

Continued…

Overriding the `equals` Method

- Define the `equals()` method to test whether two objects have equal state

When redefining `equals()` method, you cannot change object signature; use a cast instead:

```
public class Coin
{
    . . .
    public boolean equals(Object otherObject)
    {
        Coin other = (Coin) otherObject;
        return name.equals(other.name) && value == other.value;
    }
    . . .
}
```

- You should also override the **`hashCode()`** method so that equal objects have the same hash code

Self Check

16. Should the call `x.equals(x)` always return `true`?
17. Can you implement `equals` in terms of `toString`? Should you?

Answers

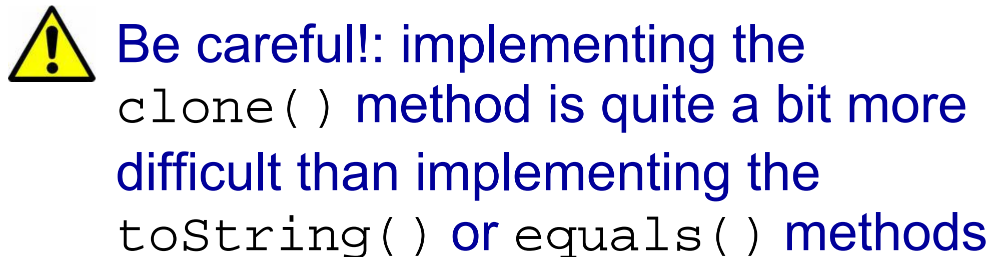
16. It certainly should–unless, of course, `x` is `null`.
17. If `toString` returns a string that describes all instance fields, you can simply call `toString` on the implicit and explicit parameters, and compare the results. However, comparing the fields is more efficient than converting them into strings.

Overriding the `clone` Method

- Copying an object reference gives you two references to the same object:

```
BankAccount account2 = account;
```

- Sometimes, what you need is to make a copy of the object → that is the purpose of the **`clone()`** method
- The `clone()` method must return a new object that has an identical state to the existing object



Be careful!: implementing the `clone()` method is quite a bit more difficult than implementing the `toString()` or `equals()` methods

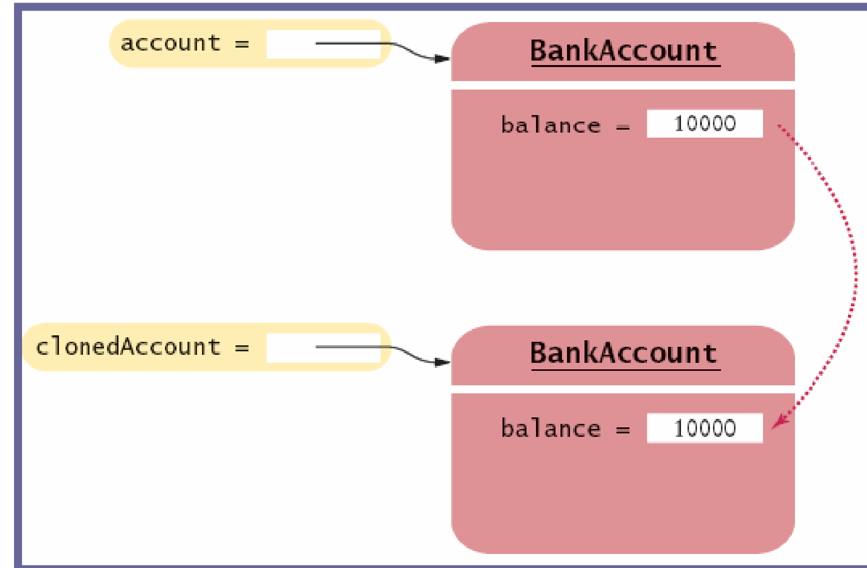


Figure 11: Cloning Objects

Chapter Summary

- Inheritance is a mechanism for extending classes by adding methods and fields
- The more general class is called a superclass. The more specialized class that inherits from the superclass is called the subclass
- Every class extends the `Object` class either directly or indirectly
- Inheriting from a class differs from implementing an interface: The subclass inherits behavior and state from the superclass
- One advantage of inheritance is code reuse
- When defining a subclass, you specify added instance fields, added methods, and changed or overridden methods
- Sets of classes can form complex inheritance hierarchies
- A subclass has no access to private fields of its superclass

Continued

Chapter Summary

- Use the `super` keyword to call a method of the superclass
- To call the superclass constructor, use the `super` keyword in the first statement of the subclass constructor
- Subclass references can be converted to superclass references
- The `instanceof` operator tests whether an object belongs to a particular type
- An abstract method is a method whose implementation is not specified
- An abstract class is a class that cannot be instantiated
- A field or method that is not declared as `public`, `private`, or `protected` can be accessed by all classes in the same package
- Protected features can be accessed by all subclasses and all classes in the same package

Continued

Chapter Summary

- Define the `toString()` method to yield a string that describes the object state
- Define the `equals()` method to test whether two objects have equal state
- The `clone()` method makes a new object with the same state as an existing object