Principles of Computer Science II

Prof. Nadeem Abdul Hamid

CSC 121 – Spring 2006

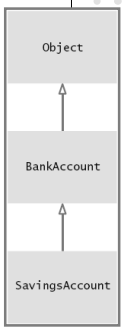
Lecture Unit 4 - Inheritance

Inheritance

- Extending classes by adding methods and fields
  - Extended class = “Superclass”
  - Extending class = “Subclass”
- Inheriting from a class is not the same as implementing an interface
  - Subclass inherits behavior and state
- Advantage of inheritance: Code reuse

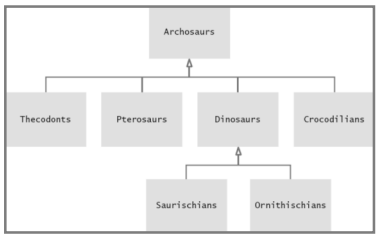
Inheritance Diagram

- Every class in Java extends the Object class either directly or indirectly

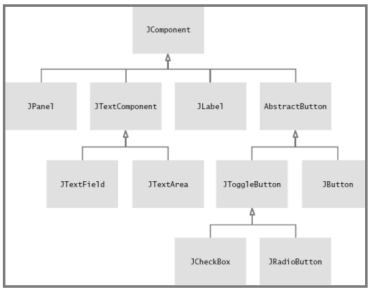


Inheritance Hierarchies

- Sets of classes can form complex inheritance hierarchies



Swing Example



Subclass Methods

- Inherit method:
  - Don’t supply a new implementation of a method that exists in superclass
  - Superclass method can be applied to the subclass objects
- Override method:
  - Supply a different implementation of a method that exists in the superclass
  - Must have same signature (same name and same parameter types)
  - If method is applied to an object of the subclass type, the overriding method is executed
- Add method:
  - Supply a new method that doesn’t exist in the superclass
  - New method can be applied only to subclass objects

Instance Fields

- Can't override fields
- Can:
  - Inherit a field: All fields from the superclass are automatically inherited
  - Add a field: Supply a new field that doesn't exist in the superclass
- What if you define a new field with the same name as a superclass field?
  - Each object would have two instance fields of the same name
  - Fields can hold different values
  - Legal but extremely undesirable

Instance Fields

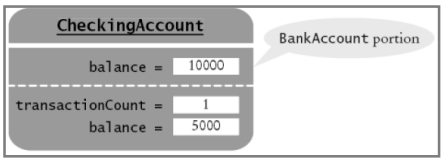
```
// COMPILE ERROR ******

class A {
    private int x;
        //...
}

class B extends A {
        //...
    public void methodB() {
        x = 2;
    }
}
```


Common Error: Shadowing Fields

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



Invoking Superclass Methods

```
class A {
    public void method() {
        System.out.println( "Method A" );
    }
}

class B {
    public void method() {
        method();    // infinite call to itself
        System.out.println( "Method B" );
    }
}
```

```
class A {
    public void method() {
        System.out.println( "Method A" );
    }
}

class B {
    public void method() {
        super.method();
        System.out.println( "Method B" );
    }
}
```

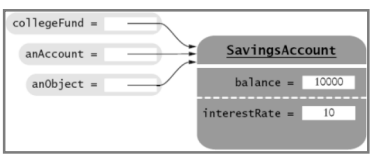

Subclass Constructors

- `super` followed by a parenthesis indicates a call to the superclass constructor
- Must be the *first* statement in subclass constructor
- If subclass constructor doesn't call superclass constructor, default superclass constructor is used
  - Default constructor: constructor with no parameters
  - If all constructors of the superclass require parameters, then the compiler reports an error

Converting Types

- Ok to convert subclass reference to superclass reference

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



Converting Between 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?
  - Reuse code that knows about the superclass but not the subclass:

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

  - Can be used to transfer money from any type of BankAccount

Casting Object Types

- Occasionally you need to convert from a superclass reference to a subclass reference

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

- This cast is dangerous: if you are wrong, an exception is thrown
- *Solution*: use the **`instanceof`** operator
  - **`instanceof`**: tests whether an object belongs to a particular type

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


Polymorphism

- In Java, type of a variable doesn't completely determine type of object to which it refers

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

- Method calls are determined by type of actual object, not type of object reference

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

- Compiler needs to check that only legal methods are invoked

```
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);
   other.deposit(amount);
}
```

- Depending on types of amount and other, different versions of **`withdraw`** and **`deposit`** are called

Access Control

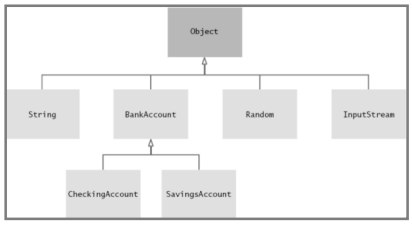
- Java has four levels of controlling access to fields, methods, and classes:
  - **public** - Can be accessed by methods of all classes
  - **private** - Can be accessed only by the methods of their own class
  - package access (default) - Can be accessed by all classes in the same package (folder)
  - **protected** - Can be accessed by all subclasses and by all classes in the same package

Recommended Access Levels

- **Instance and static fields**: Always private. Exceptions:
  - public static final constants are useful and safe
  - Some objects, such as System.out, need to be accessible to all programs (public)
  - Occasionally, classes in a package must collaborate very closely (give some fields package access); inner classes are usually better
- **Methods**: public or private
- **Classes** and **interfaces**: public or package
  - Better alternative to package access: inner classes
  - In general, inner classes should not be public (some exceptions exist, e.g., Ellipse2D.Double)
- Beware of accidental package access (forgetting public or private)

Object: The Cosmic Superclass

- All classes defined without an explicit **extends** clause automatically extend **Object**



19

Object Class Methods

- Most useful methods:
  - String toString()
  - boolean equals(Object otherObject)
  - Object clone()
- Good idea to override these methods in your classes

20

The toString() method

- Returns a string representation of the object
- Useful 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]"
```

- toString 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

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

21

Overriding the toString() method

- To provide a nicer representation of an object, override toString:

```
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]"
```

22

The equals() method

- Tests for equal *contents*:

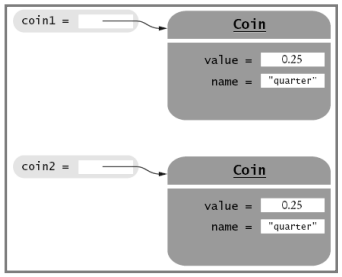


Figure 9:
Two References to Equal Objects

23

The equals() method

- == tests for equal *location*

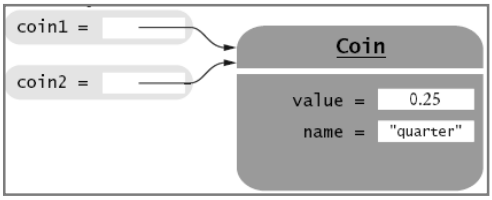


Figure 10:
Two References to the Same Object

24

Overriding equals()

- 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)

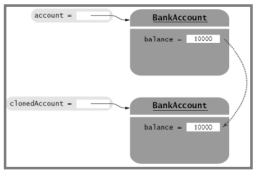
25

The clone() method

- Copying an object reference gives two references to same object

```
BankAccount account2 = account;
```

- Sometimes, you need to make a *copy* of the object



26

The clone() method

- Define **clone** method to make new object (see Advanced Topic 13.6)
- Use **clone**:

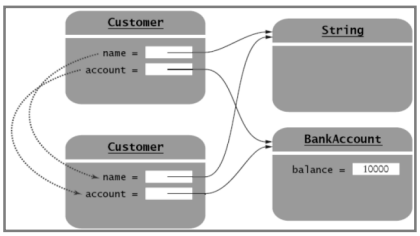
```
BankAccount clonedAccount = (BankAccount) account.clone();
```

- Must cast return value because return type is **Object**

27

The Object.clone() method

- Creates *shallow* copies



28

Object.clone()…

- Does not systematically clone all subobjects
- Must be used with caution
- It is declared as **protected**; prevents from accidentally calling **x.clone()** if the class to which **x** belongs hasn't redefined **clone** to be **public**
- You should override the **clone** method with care (see Advanced Topic 13.6)

29

Scripting Languages

- Integrated with software for purpose of automating repetitive tasks
- Script: Very high-level, often short, program, written in a high-level scripting language
- Scripting languages: Unix shells, Tcl, Perl, Python, Ruby, Scheme, Rexx, JavaScript, VisualBasic, ...

30

Characteristics of a script

- Glue other programs together
- Extensive text processing
- File and directory manipulation
- Often special-purpose code
- Many small interacting scripts may yield a big system
- Perhaps a special-purpose GUI on top
- Portable across Unix, Windows, Mac
- Interpreted program (no compilation+linking)

31

Why Scripts?

- Features of Perl and Python compared with Java, C/C++ and Fortran:
  - shorter, more high-level programs
  - much faster software development
  - more convenient programming
  - you feel more productive
- Reasons:
  - no variable declarations, but lots of consistency checks at run time
  - lots of standardized libraries and tools

32