Week 7
Inheritance

(Chapter 9)

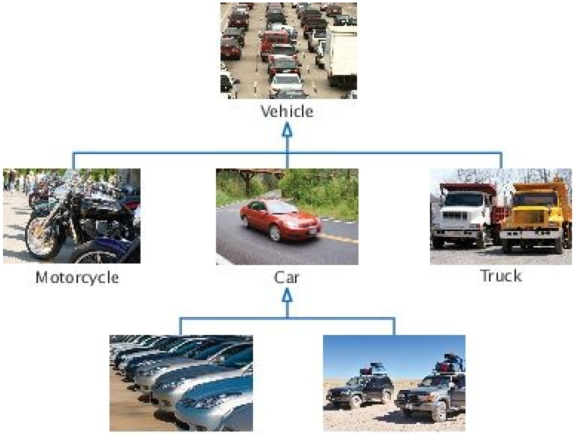
Chapter Goals

© Jason Hosking/Photodisc/Getty Images, Inc.

- To learn about inheritance
- To implement subclasses that inherit and override superclass methods
- To understand the concept of polymorphism
- To be familiar with the common superclass `Object` and its methods

Inheritance Hierarchies

- Inheritance: the relationship between a more general class (superclass) and a more specialized class (subclass).
- The subclass inherits data and behavior from the superclass.
- Cars share the common traits of all vehicles
  - Example: the ability to transport people from one place to another



© Richard Stouffer/iStockphoto (vehicle); © Ed Hidden/iStockphoto (motorcycle); © YinYang/iStockphoto (car);
© Robert Pernell/iStockphoto (truck); Media Bakery (sedan); Cezary Wojtkowski/Age Fotostock America (SUV).

Figure 1 An Inheritance Hierarchy of Vehicle Classes

Inheritance Hierarchies

- The class `Car` inherits from the class `Vehicle`
- The `Vehicle` class is the superclass
- The `Car` class is the subclass

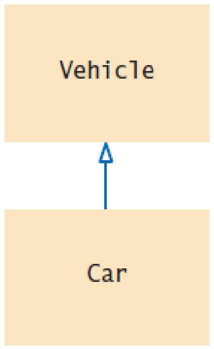


Figure 2 Inheritance Diagram

Inheritance Hierarchies

- Inheritance lets you can reuse code instead of duplicating it.
- Two types of reuse
  - A subclass inherits the methods of the superclass
  - Because a car is a special kind of vehicle, we can use a `Car` object in algorithms that manipulate `Vehicle` objects
- The substitution principle:
  - You can always use a subclass object when a superclass object is expected.
- A method that processes `Vehicle` objects can handle any kind of vehicle

Inheritance Hierarchies

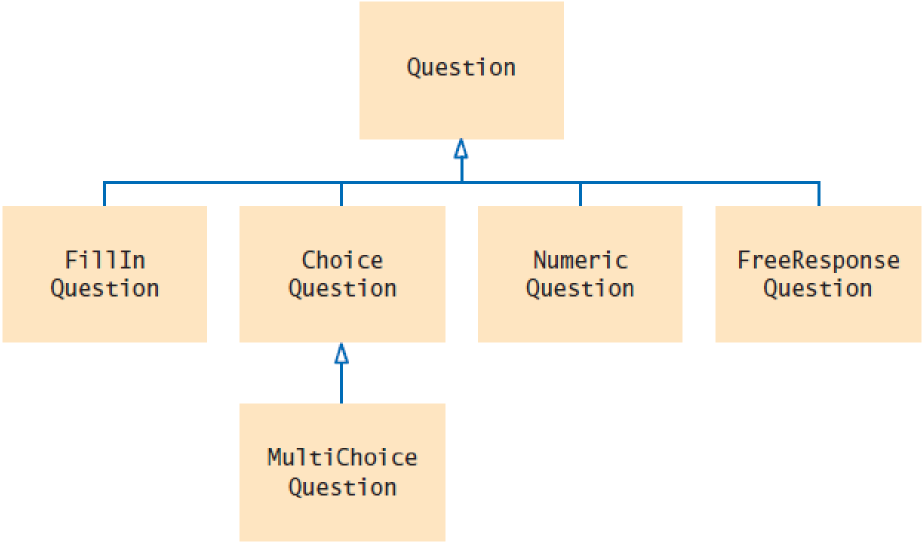


Figure 3 Inheritance Hierarchy of Question Types

Example: Computer-graded quiz

- There are different kinds of questions
- A question can display its text, and it can check whether a given response is a correct answer.
- You can form subclasses of the `Question` class.

section_1/Question.java

```java
/**
   A question with a text and an answer.
*/
public class Question
{
   private String text;
   private String answer;

   /**
      Constructs a question with empty question and answer.
   */
   public Question()
   {
      text = "";  answer = "";
   }


   /**
      Sets the question text.
      @param questionText the text of this question
   */
   public void setText(String questionText)
   {
      text = questionText;
   }

   /**
      Sets the answer for this question.
      @param correctResponse the answer
   */
   public void setAnswer(String correctResponse)
   {
```

section_1/QuestionDemo1.java

```
import java.util.Scanner;

/**
   This program shows a simple quiz with one question.
*/
public class QuestionDemo1
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      Question q = new Question();
      q.setText("Who was the inventor of Java?");
      q.setAnswer("James Gosling");

      q.display();  System.out.print("Your answer: ");
      String response = in.nextLine();
      System.out.println(q.checkAnswer(response));
   }
}
```

Program Run:

```
Who was the inventor of Java?
Your answer: James Gosling
true
```

Implementing Subclasses

Media Bakery.

Like the manufacturer of a stretch limo, who starts with a regular car and modifies it, a programmer makes a subclass by modifying another class.

- To get a `ChoiceQuestion` class, implement it as a subclass of `Question`

  Specify what makes the subclass different from its superclass.

  - Subclass objects automatically have the ***instance variables*** that are declared in the superclass.
  - Subclass objects only declare instance variables that are not part of the superclass objects.

- A subclass inherits all methods that it does not override.

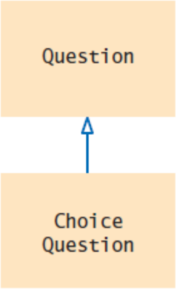


Figure 4 The `ChoiceQuestion` Class is a Subclass of the `Question` Class.

Implementing Subclasses

- The subclass inherits all public methods from the superclass.
- You declare any methods that are new to the subclass.
- You change the implementation of inherited methods if the inherited behavior is not appropriate.
- **Override a method**: supply a new implementation for an inherited method

Implementing Subclasses

A `ChoiceQuestion` object differs from a `Question` object in three ways:

- Its objects store the various choices for the answer.
- There is a method for adding answer choices.
- The display method of the `ChoiceQuestion` class shows these choices so that the respondent can choose one of them.

Implementing Subclasses

- The `ChoiceQuestion` class needs to spell out the three differences:

```
public class ChoiceQuestion extends Question
{
  // This instance variable is added to the

  subclass  private ArrayList<String> choices;


  // This method is added to the subclass
  public void addChoice(String choice, boolean correct) { . . . }

  // This method overrides a method from the

  superclass  public void display() { . . . }
}
```

- The `extends` reserved word indicates that a class inherits from a superclass.

Implementing Subclasses

- UML of `ChoiceQuestion` and `Question`

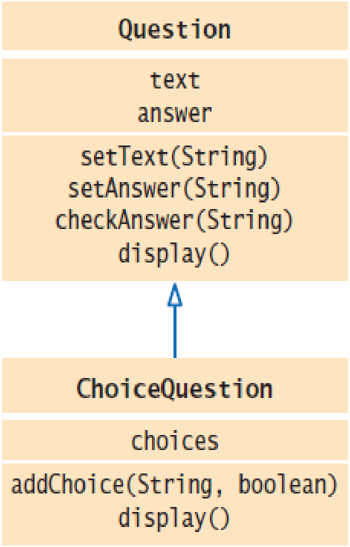
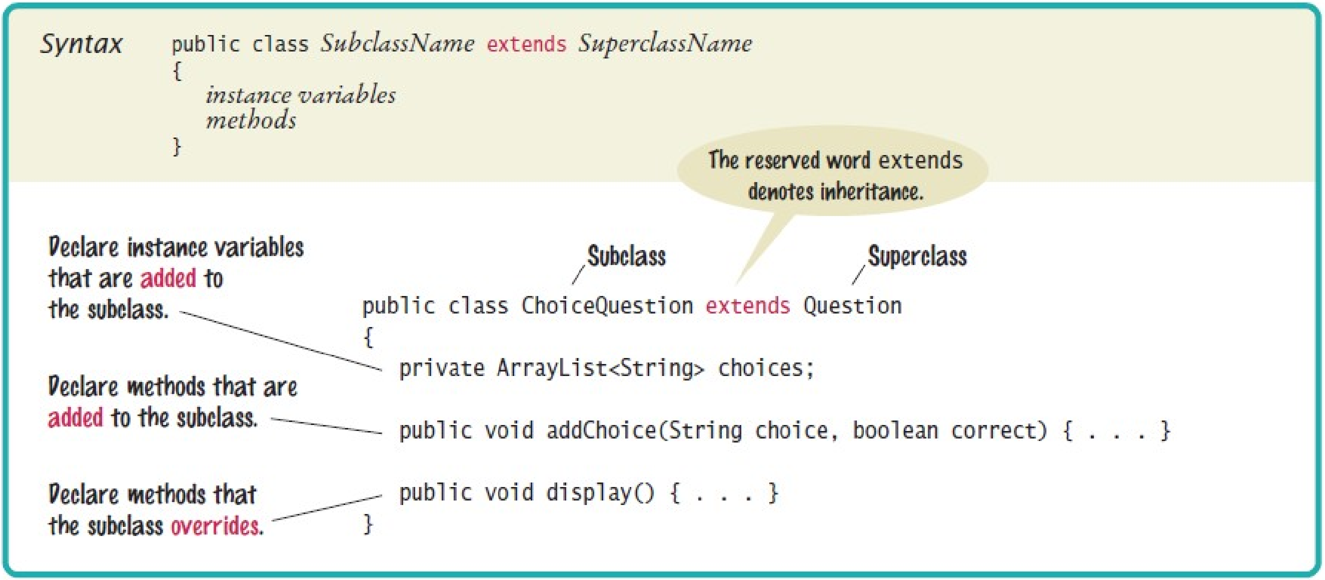


Figure 5 The `ChoiceQuestion` Class Adds an Instance Variable and a Method, and Overrides a Method

Syntax 9.1 Subclass Declaration



Syntax

```
public class SubclassName extends SuperclassName
{
   instance variables
   methods
}
```

```
public class ChoiceQuestion extends Question
{
   private ArrayList<String> choices;

   public void addChoice(String choice, boolean correct) { . . . }

   public void display() { . . . }
}
```

The reserved word extends denotes inheritance.

Subclass

Superclass

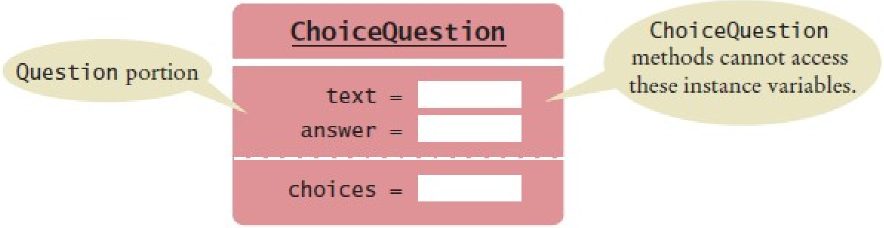
Declare instance variables that are added to the subclass.

Declare methods that are added to the subclass.

Declare methods that the subclass overrides.

Implementing Subclasses

- A ChoiceQuestion object


ChoiceQuestion
Question portion
text =
answer =
choices =
ChoiceQuestion methods cannot access these instance variables.

- You can call the inherited methods on a subclass object:

```
choiceQuestion.setAnswer("2");
```

- The *private instance variables* of the superclass are inaccessible.
- The ChoiceQuestion methods cannot directly access the instance variable answer.
- ChoiceQuestion methods must use the public interface of the Question class to access its private data.

Implementing Subclasses

- Adding a new method: `addChoice`

```
public void addChoice(String choice, boolean correct)
{
   choices.add(choice);
   if (correct)
   {
      // Convert choices.size() to string
      String choiceString = "" + choices.size();
      setAnswer(choiceString);
   }
}
```

- `addChoice` method can not just access the `answer` variable in the superclass:
- It must use the `setAnswer` method
- Invoke `setAnswer` on the implicit parameter:

```
setAnswer(choiceString);
```

OR

```
this.setAnswer(choiceString);
```

- `ChoiceQuestion` class represents a union of features implemented in `Question` as well as `ChocieQuestion` classes!

Common Error: Replicating Instance Variables from the Superclass

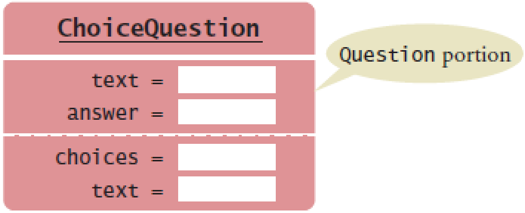
- A subclass has no access to the private instance variables of the superclass:

```
public ChoiceQuestion(String questionText)
{
   text = questionText; // Error—tries to access private superclass variable
}
```

- Beginner's error: "solve" this problem by adding another instance variable with same name
- Error!

```
public class ChoiceQuestion extends Question
{
   private ArrayList<String> choices;
   private String text; // Don’t!
   . . .
}
```

- The constructor compiles, but it doesn’t set the correct



- The `ChoiceQuestion` constructor should call the `setText` method of the `Question` class.

Overriding Methods

- If you are not satisfied with the behavior of an inherited method,
  - you override it by specifying a new implementation in the subclass.
- An overriding method can extend or replace the functionality of the superclass method.
- The display method of the `ChoiceQuestion` class needs to:
  - Display the question text.
  - Display the answer choices.

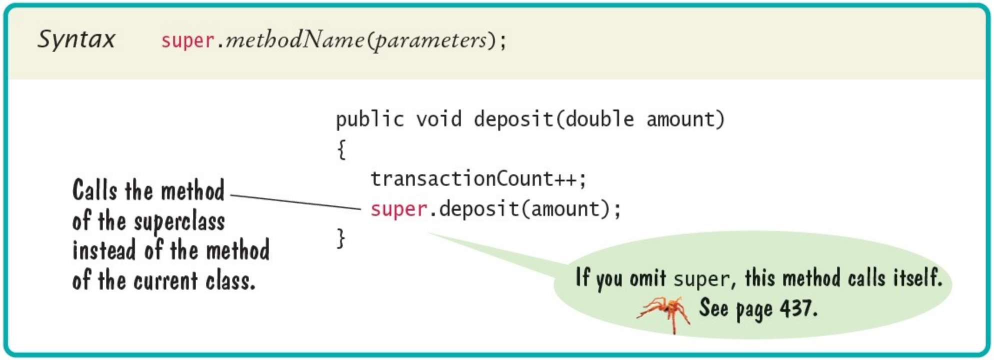
Overriding Methods

- Problem: `ChoiceQuestion`'s `display` method can’t access the `text` variable of the superclass directly because it is private.
- Solution: It can call the `display` method of the superclass, by using the reserved word `super`

```
public void display()
{
   // Display the question text
   super.display(); // OK
   // Display the answer choices
   . . .
}
```

- `super` is a reserved word that forces execution of the superclass method.

Syntax 9.2 Calling a Superclass Method



Syntax super.*methodName*(*parameters*);

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

Calls the method of the superclass instead of the method of the current class.

If you omit super, this method calls itself. See page 437.

section_3/QuestionDemo2.java

```java
import java.util.Scanner;

/**
   This program shows a simple quiz with two choice questions.
*/
public class QuestionDemo2
{
   public static void main(String[] args)
   {
      ChoiceQuestion first = new ChoiceQuestion();
      first.setText("What was the original name of the Java language?");
      first.addChoice("*7", false);
      first.addChoice("Duke", false);
      first.addChoice("Oak", true);
      first.addChoice("Gosling", false);

      ChoiceQuestion second = new ChoiceQuestion();
      second.setText("In which country was the inventor of Java born?");
      second.addChoice("Australia", false);
      second.addChoice("Canada", true);
      second.addChoice("Denmark", false);
      second.addChoice("United States", false);


      presentQuestion(first);  presentQuestion(second);

   }
   /**
      Presents a question to the user and checks the response.
      @param q the question
   */
   public static void presentQuestion(ChoiceQuestion q)
   {
      q.display();
      System.out.print("Your answer: ");
```

section_3/ChoiceQuestion.java

```java
import java.util.ArrayList;

/**
  A question with multiple choices.
*/
public class ChoiceQuestion extends Question
{
  private ArrayList<String> choices;

  /**
    Constructs a choice question with no choices.
  */
  public ChoiceQuestion()
  {
    choices = new ArrayList<String>();
  }

  /**
    Adds an answer choice to this question.
    @param choice the choice to add
    @param correct true if this is the correct choice, false otherwise
  */
  public void addChoice(String choice, boolean correct)
  {
    choices.add(choice);
    if (correct)
    {
    // Convert choices.size() to string

```

Program Run:

```
What was the original name of the Java language?  1: *7
2: Duke
3: Oak
4: Gosling
Your answer: *7  false
In which country was the inventor of Java born?
1: Australia
2: Canada
3: Denmark
4: United States

Your answer: 2  true
```

Common Error: Accidental Overloading

- **Overloading**: when two methods have the same name but different parameter types.
- **Overriding**: when a subclass method provides an implementation of a superclass method whose parameter variables have the same types.
- When **overriding** a method, the types of the parameter variables must match exactly.

Common Error: Forgetting to Use super When Invoking Superclass Method

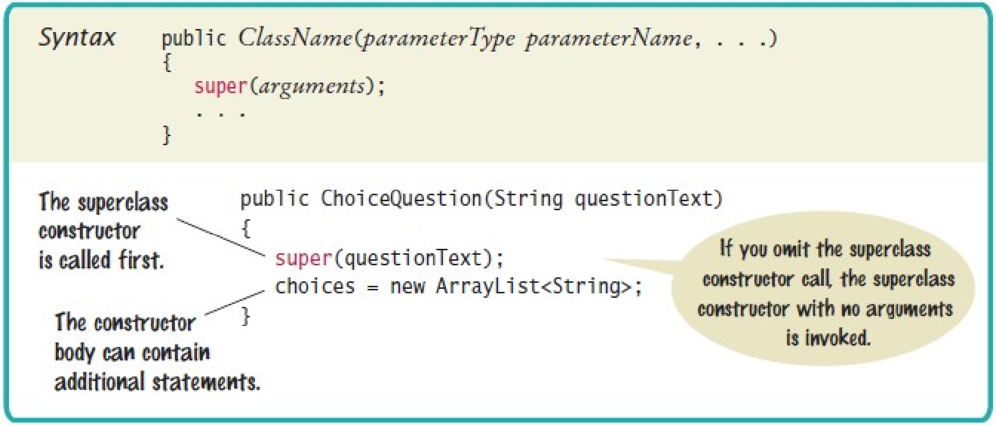
- Use super when extending Employee functionality to Manager class

```
public class Manager {
  . . .
  public double getSalary() {
    double baseSalary = getSalary();
       // Error: should be super.getSalary()
     return baseSalary + bonus;
   }
}
```

Constructor Chaining

- Constructor is called whenever an instance of a class is created.
- Java guarantees that the constructor of **superclass** is also called when an instance of any subclass is created. In order to guarantee this, Java must ensure that every constructor method calls its superclass constructor method.
- If the first statement in a constructor is not an explicit call to a constructor of the superclass with the super keyword, then Java implicitly inserts the call super() -- that is, it calls the superclass constructor with no arguments.
- If the superclass does not have a constructor that takes no arguments, this causes a **compilation error**.
- Suppose a class *C2* is a subclass of *C1*. The constructor of *C2* either explicitly or implicitly calls constructor of *C1,* which in turn calls constructor of *Object*.
  - *Object* constructor runs first, followed by constructor *C1,* followed by constructor *C2.*

Syntax 9.3 Constructor with Superclass Initializer



Syntax

```
public ClassName(parameterType parameterName, . . .)
{
   super(arguments);
   . . .
}
```

```
public ChoiceQuestion(String questionText)
{
   super(questionText);
   choices = new ArrayList<String>;
}
```

The superclass constructor is called first.

The constructor body can contain additional statements.

If you omit the superclass constructor call, the superclass constructor with no arguments is invoked.

Constructor Chaining

```
class Parent{
    public Parent(){ System.out.println("Parent class default constructor "); }
    public Parent(String name){ System.out.println("Parent class constructor 2"+name); }
}

public class Child extends Parent{
    public Child() {
        this(“Jacob"); //Calling its own constructor explicitly
        System.out.println("Child class default constructor");
    }
    public Child(String name){
        super(“Joseph"); //constructor chaining
        System.out.println("Child class constructor 2"+name);
    }
    public static void main(String args[])
    {
        Child c = new Child();
    }
}
```

Program Run:

```
Parent class constructor 2 Joseph
Child class constructor 2 Jacob
Child class default constructor
```

The ***protected*** modifier

- Visibility modifiers affect the way that class members can be used in a child class.
- Variables and methods declared with `private` visibility cannot be referenced in a child class.
- They can be referenced in the child class if they are declared with `public` visibility -- but `public` variables violate the ***principle of encapsulation***.
- There is a third visibility modifier that helps in inheritance situations: ***`protected`***
- The `protected` modifier allows **a child class to reference a variable or method of the parent class**.
- It provides more encapsulation than `public` visibility, but is not as tightly encapsulated as `private` visibility.
- A `protected` variable is also visible to any class in the same package as the parent class.
- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams.

Polymorphism

- Problem: to present both `Question` and `ChoiceQuestion` with the same program.
- We do not need to know the exact type of the question
  - We need to display the question
  - We need to check whether the user supplied the correct answer
- The `Question` superclass has methods for displaying and checking.
- We can simply declare the parameter variable of the `presentQuestion` method to have the type `Question`:

```
public static void presentQuestion(Question q)
{
   q.display();
   System.out.print("Your answer: ");
   Scanner in = new Scanner(System.in);
   String response = in.nextLine();
   System.out.println(q.checkAnswer(response));
}
```

Polymorphism - continued

- We can substitute a subclass object whenever a superclass object is expected:

```
ChoiceQuestion second = new ChoiceQuestion();
. . .
presentQuestion(second); // OK to pass a ChoiceQuestion
```

- When the `presentQuestion` method executes -
  - The object references stored in `second` and q refer to the same object
  - The object is of type `ChoiceQuestion`

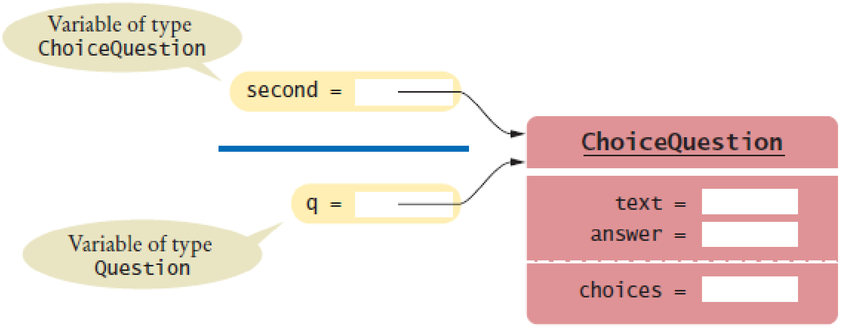


Figure 7 Variables of Different Types Referring to the Same Object

Polymorphism - continued

- The variable q knows less than the full story about the object to which it refers

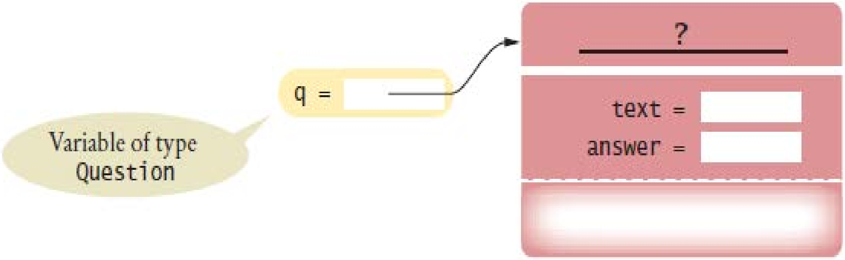


Figure 8 A Question Reference Can Refer to an Object of Any Subclass of Question

- In the same way that vehicles can differ in their method of locomotion, polymorphic objects carry out tasks in different ways.

© Alpophoto/iStockphoto.

Polymorphism - continued

- When the virtual machine calls an instance method -
  - It locates the method of the implicit parameter's class.
  - This is called dynamic method lookup
- Dynamic method lookup allows us to treat objects of different classes in a uniform way.
- This feature is called polymorphism.
- We ask multiple objects to carry out a task, and each object does so in its own way.
- Polymorphism means "having multiple forms"
  - It allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.

The instanceof Operator

- It is *legal to store a subclass reference in a superclass variable*:

```
ChoiceQuestion cq = new ChoiceQuestion();
Question q = cq; // OK

Object obj = cq; // OK
```

- Sometimes you need to convert from a superclass reference to a subclass reference.
- If you know a variable of type `Object` actually holds a `Question` reference, you can cast

```
Question q = (Question) obj
```

- If `obj` refers to an object of an unrelated type, "class cast" exception is thrown.

  The `instanceof` operator tests whether an object belongs to a particular type.

```
obj instanceof Question
```

- Using the `instanceof` operator, a safe cast can be programmed as follows:

```
if (obj instanceof Question)
{
   Question q = (Question) obj;
}
```

section_4/QuestionDemo3.java

```
import java.util.Scanner;

/**
  This program shows a simple quiz with two question types.
*/
public class QuestionDemo3
{
  public static void main(String[] args)
  {
    Question first = new Question();
    first.setText("Who was the inventor of Java?");
    first.setAnswer("James Gosling");

    ChoiceQuestion second = new ChoiceQuestion();
    second.setText("In which country was the inventor of Java born?");
    second.addChoice("Australia", false);
    second.addChoice("Canada", true);
    second.addChoice("Denmark", false);
    second.addChoice("United States", false);

    presentQuestion(first);
    presentQuestion(second);
  }

  /**
    Presents a question to the user and checks the response.
    @param q the question
  */
  public static void presentQuestion(Question q)
  {
    q.display();  System.out.print("Your answer: ");
    Scanner in = new Scanner(System.in);
    String response = in.nextLine();
    System.out.println(q.checkAnswer(response));

```

Program Run:

```
Who was the inventor of Java?
Your answer: Bjarne Stroustrup
false

In which country was the inventor of Java born?
1: Australia
2: Canada
3: Denmark
4: United States
Your answer: 2  true
```

Object: The Cosmic Superclass

- Every class defined without an explicit `extends` clause automatically extend `Object`:
- The class `Object` is the direct or indirect superclass of every class in Java.
- Some methods defined in `Object`:

  `toString` - which yields a string describing the object

  `equals` - which compares objects with each other

  `hashCode` - which yields a numerical code for storing the object in a set

Object: The Cosmic Superclass

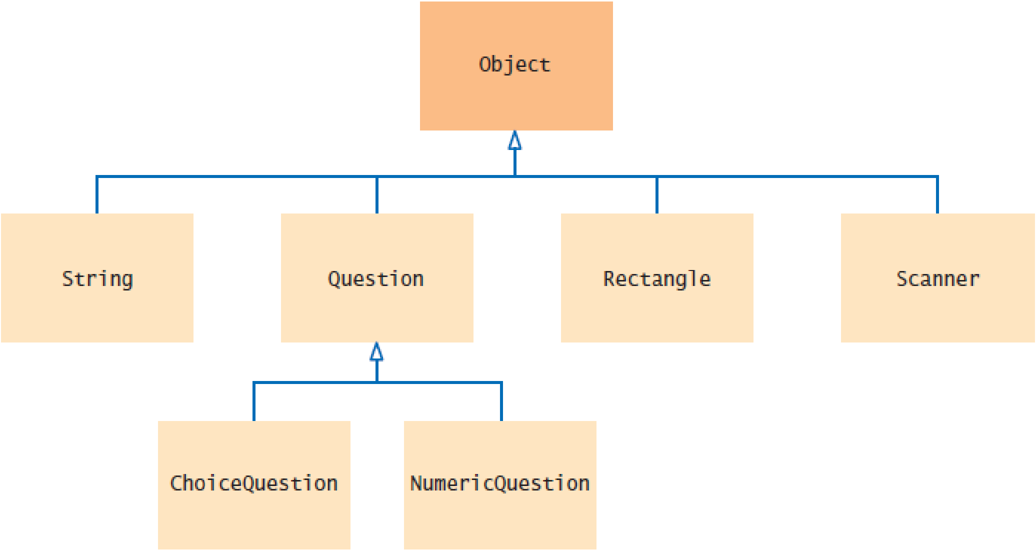


Figure 9 The `Object` Class Is the Superclass of Every Java Class

Overriding 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]"
```

- The compiler can invoke the `toString` method, because it knows that every object has a `toString` method:
    - Every class extends the `Object` class which declares `toString`

Overriding the toString Method

- `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"
```

- Override the `toString` method in your classes to yield a string that describes the object's state.

```
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

- equals method checks whether two objects have the same content:

```
if (stamp1.equals(stamp2)) . . .
   // Contents are the same
```

© Ken Brown/iStockphoto.

- == operator tests whether two references are identical - referring to the same object:

```
if (stamp1 == stamp2) . . .
   // Objects are the same
```

Overriding the equals Method

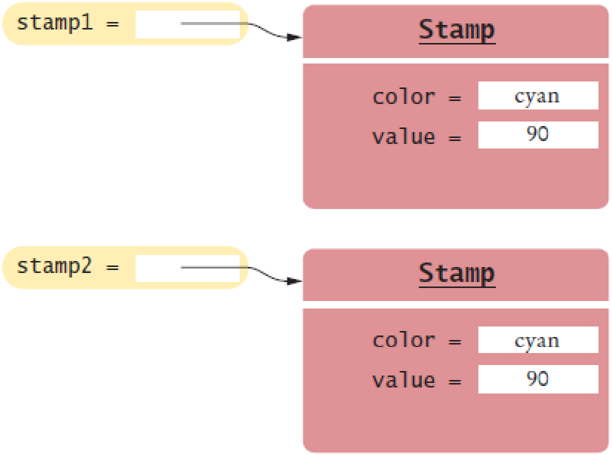


Figure 10 Two References to Equal Objects

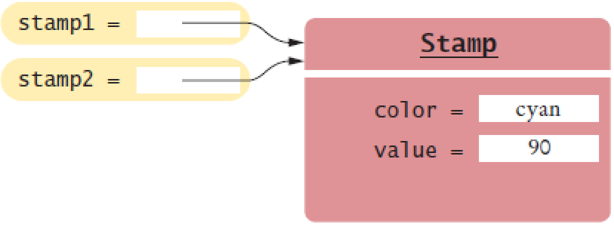


Figure 11 Two References to the Same Object

Overriding the equals Method

- To implement the equals method for a `Stamp` class -
  - Override the `equals` method of the `Object` class:

```
public class Stamp
{
   private String color;
   private int value;
   . . .
   public boolean equals(Object otherObject)
   {
         . . .
   }
   . . .
}
```

- Cannot change parameter type of the `equals` method - it must be Object
- Cast the parameter variable to the class `Stamp` instead:

```
Stamp other = (Stamp) otherObject;
```

Overriding the equals Method

- After casting, you can compare two Stamps

```
public boolean equals(Object otherObject)
{
  Stamp other = (Stamp) otherObject;
  return color.equals(other.color)
        && value == other.value;
}
```

- The `equals` method can access the instance variables of any `Stamp` object.
- The access `other.color` is legal.

The *abstract* class

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept. An abstract class **cannot** be instantiated.
- We use the modifier abstract on the class header to declare a class as `abstract`:

```
public abstract class Product
{
   // class contents
}
```

- An abstract class often contains abstract methods with no definitions (like an interface).
- The `abstract` modifier must be applied to each abstract method.
- Also, an `abstract` class typically contains non-abstract methods with full definitions.

- Unless defined as `final`, the child of an `abstract` class **must override** the `abstract` methods of the parent, or it too will be considered abstract.

- An `abstract` method cannot be defined as final or static.

The *abstract* class

```
abstract class Vehicle {  //Can’t instantiate, can only extend
      abstract void run();  //subclass must implement this method
      void start() { // a non-abstract method.
            System.out.println("I can start …");
      }
    }

class Audi extends Vehicle {
    void run(){
            System.out.println("my car is running smoothly …");
    }

    public static void main(String args[]) {

     Vehicle myAudi = new Audi();
     myAudi.start();
     myAudi.run();

    }
}
```

Program Run:

```
        I can start …
        my car is running smoothly …
```