LECTURE 3

INHERITANCE AND INTERFACES

Ian Stavness, CMPT 270, 2017-18 Term 1

Lecture Goals

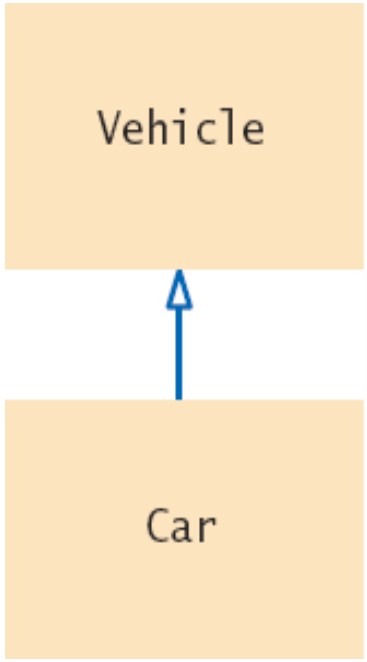
- To learn about inheritance
- To implement subclasses that inherit and override superclass methods
- To understand the concept of polymorphism
- To understand the common superclass `Object` and its methods
- To work with interface types

Contents

- Inheritance Hierarchies
- Implementing Subclasses
- Overriding Methods
- Polymorphism
- `Object`: The Cosmic Superclass
- Interface Types

9.1 Inheritance Hierarchies

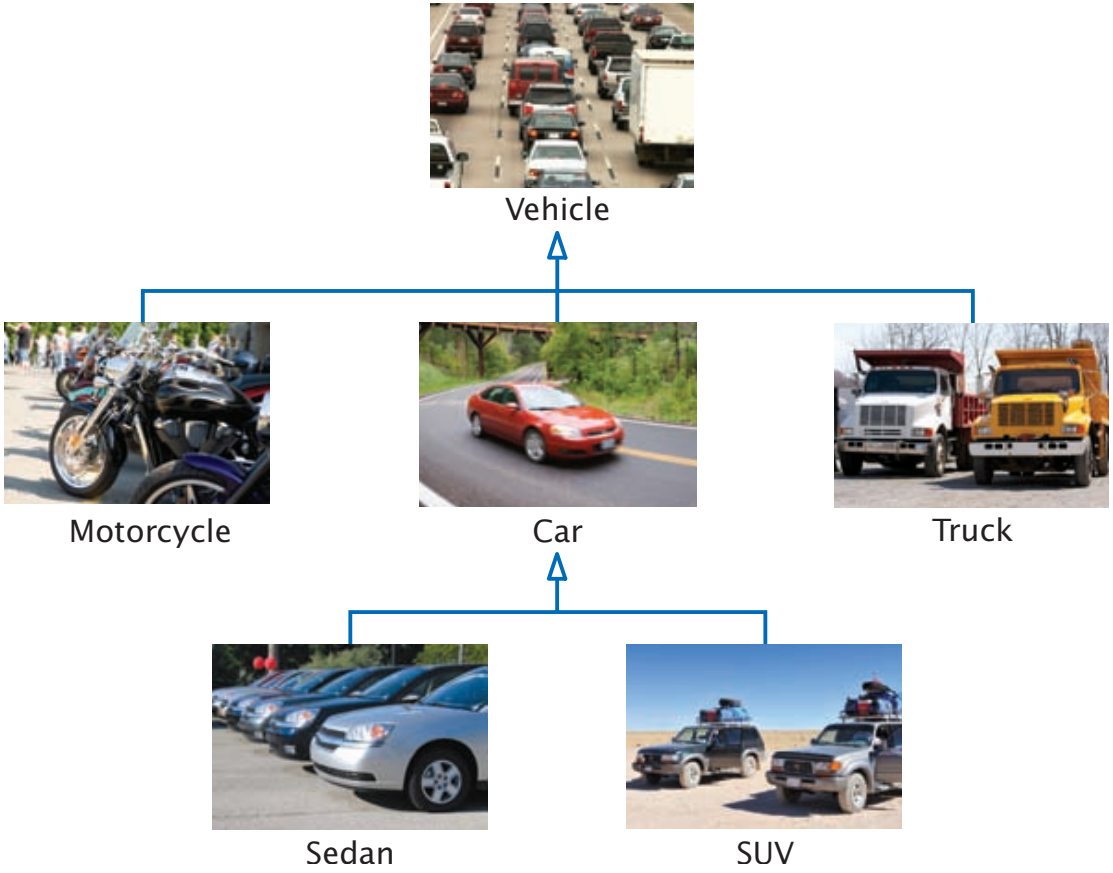
- In object-oriented programming, inheritance is a relationship between:
  - A *superclass*: a more generalized class
  - A *subclass*: a more specialized class



- The subclass 'inherits' data (variables) and behavior (methods) from the superclass

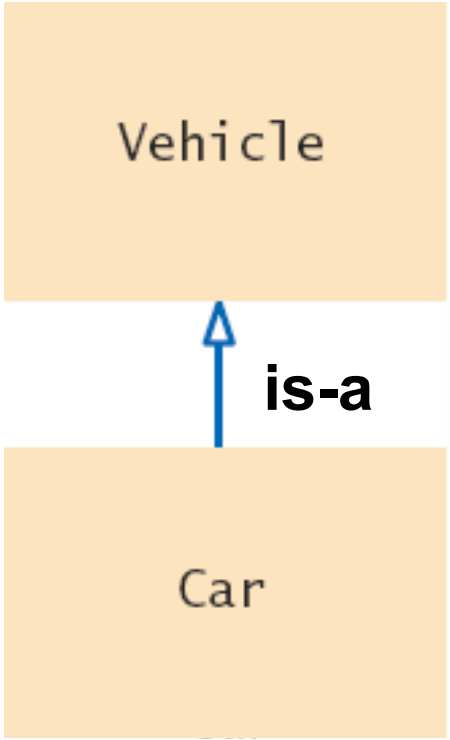
A Vehicle Class Hierarchy

- General
- Specialized
- More Specific



The Substitution Principle

- Since the subclass Car "**is-a**" Vehicle
  - Car shares common traits with Vehicle
  - You can substitute a Car object in an algorithm that expects a Vehicle object



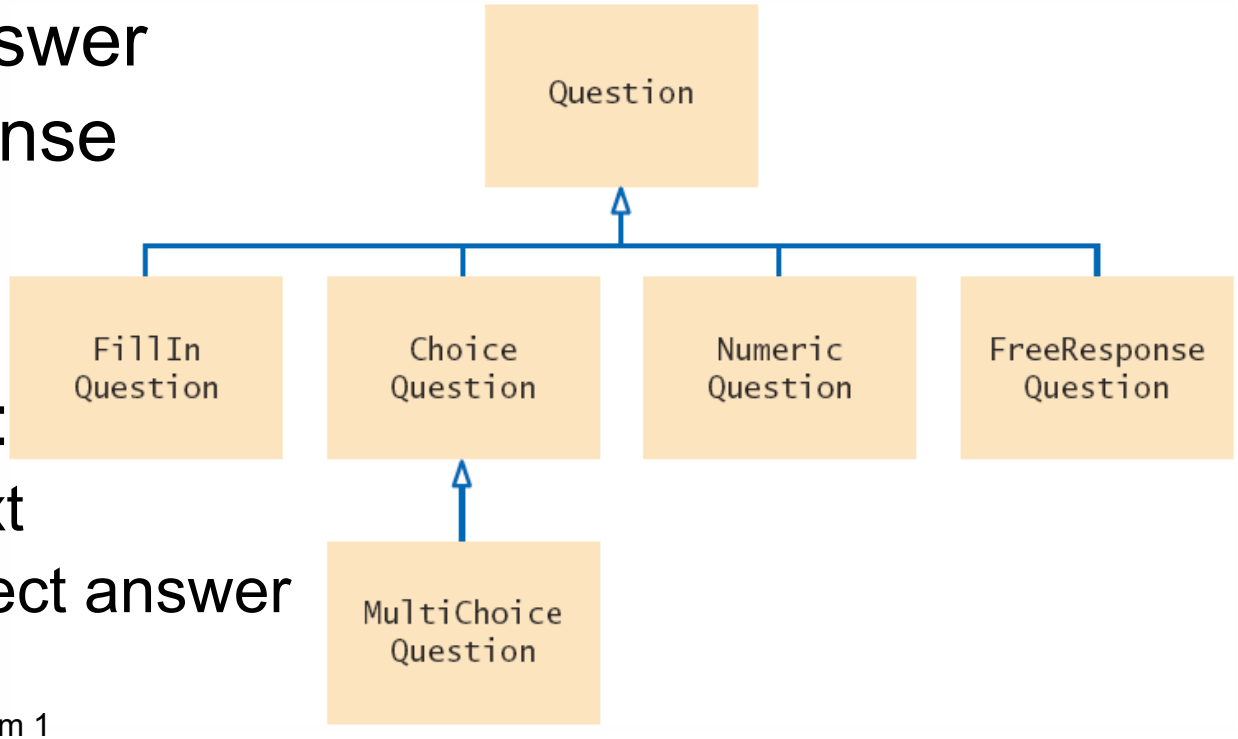

```
Car myCar = new Car(. . .);
processVehicle(myCar);
```

The 'is-a' relationship is represented by an arrow in a class diagram and means that the subclass can behave as an object of the superclass.

Quiz Question Hierarchy

- There are different types of quiz questions:
  1) Fill-in-the-blank
  2) Single answer choice
  3) Multiple answer choice
  4) Numeric answer
  5) Free Response

The 'root' of the hierarchy is shown at the top.



- A question can:
  - Display it's text
  - Check for correct answer

Question.java (1)

```
1  /**
2     A question with a text and an answer.
3  */
4  public class Question
5  {
6     private String text;
7     private String answer;
8
9     /**
10       Constructs a question with empty question and answer.
11    */
12    public Question()
13    {
14       text = "";
15       answer = "";
16    }
17
18    /**
19       Sets the question text.
20       @param questionText the text of this question
21    */
22    public void setText(String questionText)
23    {
24       text = questionText;
25    }
```

The class `Question` is the 'root' of the hierarchy, also known as the superclass

- Only handles Strings
- No support for:
  - Approximate values
  - Multiple answer choice

Question.java (2)

```
27    /**
28       Sets the answer for this question.
29       @param correctResponse the answer
30    */
31    public void setAnswer(String correctResponse)
32    {
33       answer = correctResponse;
34    }
35
36    /**
37       Checks a given response for correctness.
38       @param response the response to check
39       @return true if the response was correct, false otherwise
40    */
41    public boolean checkAnswer(String response)
42    {
43       return response.equals(answer);
44    }
45
46    /**
47       Displays this question.
48    */
49    public void display()
50    {
51       System.out.println(text);
52    }
53 }
```

QuestionDemo1.java

```
1  import java.util.ArrayList;
2  import java.util.Scanner;
3
4  /**
5     This program shows a simple quiz with one question.
6  */
7  public class QuestionDemo1
8  {
9     public static void main(String[] args)
10    {
11       Scanner in = new Scanner(System.in);
12
13       Question q = new Question();
14       q.setText("Who was the inventor of Java?");
15       q.setAnswer("James Gosling");
16
17       q.display();
18       System.out.print("Your answer: ");
19       String response = in.nextLine();
20       System.out.println(q.checkAnswer(response));
21    }
22 }
```

Program Run

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

Creates an object of the `Question` class and uses methods.

Programming Tip 9.1

- Use a Single Class for Variation in Values, Inheritance for Variation in Behavior
  - If two vehicles only vary by fuel efficiency, use an instance variable for the variation, not inheritance

```
// Car instance variable
double milesPerGallon;
```

  - If two vehicles behave differently, use inheritance

Car

Sedan

SUV

Be careful not to over-use inheritance

9.2 Implementing Subclasses

- Consider implementing `ChoiceQuestion` to handle:

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

In this section you will see how to form a subclass and how a subclass automatically inherits from its superclass

- How does `ChoiceQuestion` differ from `Question`?
  - It stores choices (1,2,3 and 4) in addition to the question
  - There must be a method for adding multiple choices
    - The display method will show these choices below the question, numbered appropriately

Inheriting from the Superclass

- Subclasses inherit from the superclass:
  - All public methods that it does not override
  - All instance variables
- The Subclass can
  - Add new instance variables
  - Add new methods
  - Change the implementation of inherited methods

Form a subclass by specifying what is different from the superclass.

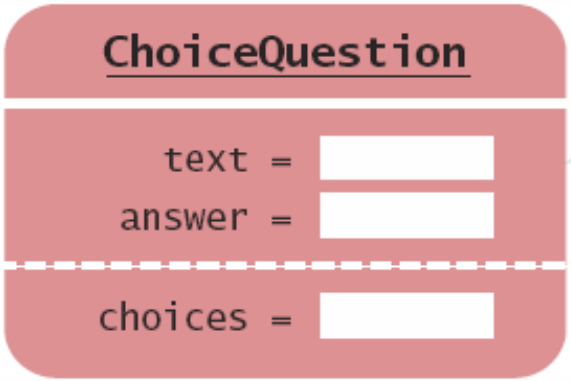
Overriding Superclass Methods

- Can you re-use any methods of the `Question` class?
  - Inherited methods perform exactly the same
  - If you need to change how a method works:
    - Write a new more specialized method in the subclass
    - Use the same method name as the superclass method you want to replace
    - It must take all of the same parameters
  - This will ***override*** the superclass method
- The new method will be invoked with the same method name when it is called on a subclass object

A subclass can override a method of the superclass by providing a new implementation.

Planning the subclass

- Use the reserved word `extends` to inherit from `Question`
  - Inherits `text` and `answer` variables
  - Add new instance variable `choices`



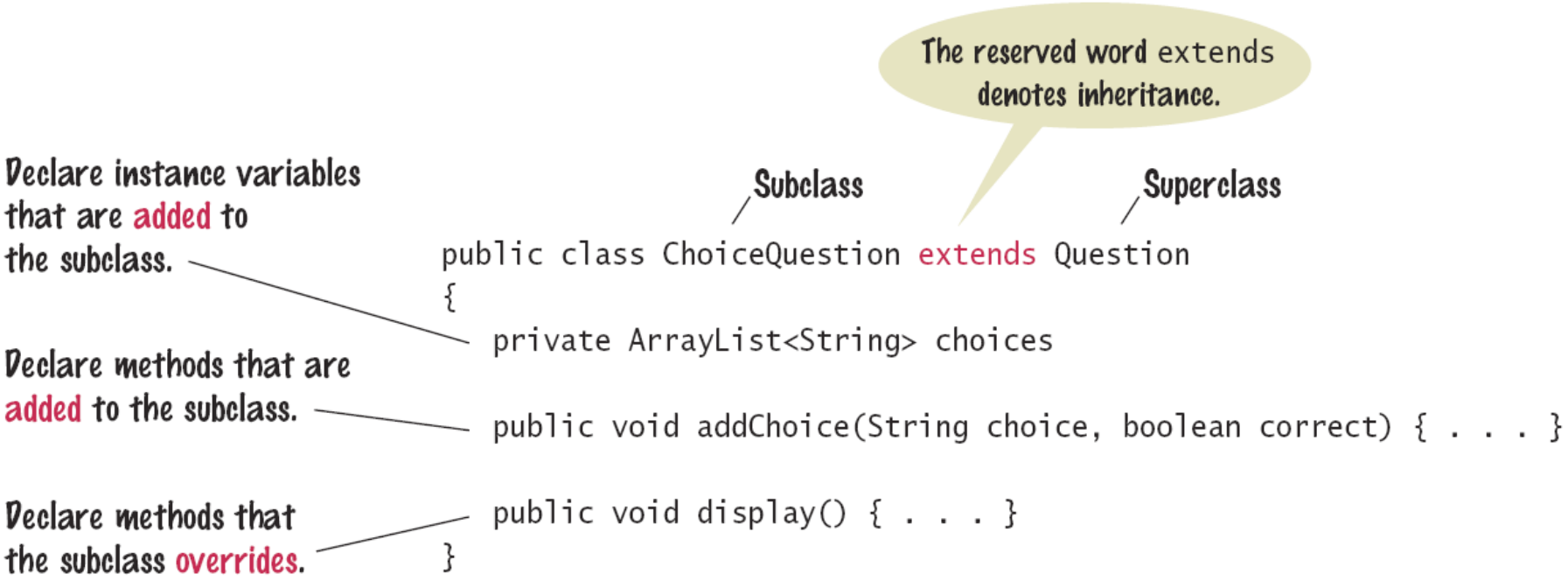

```
public class ChoiceQuestion extends Question
{
   // This instance variable is added to the subclass
   private ArrayList<String> choices;

   // 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() { . . . }
}
```

Syntax 9.1: Subclass Declaration

- The subclass inherits from the superclass and ‘`extends`’ the functionality of the superclass



Implementing addChoice

- The method will receive two parameters
  - The text for the choice
  - A boolean denoting if it is the correct choice or not
- It adds text as a choice, adds choice number to the text and calls the inherited `setAnswer` method

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

`setAnswer()` is the same as calling `this.setAnswer()`

Common Error 9.1

- Replicating Instance Variables from the Superclass
  - A subclass cannot directly access private instance variables of the superclass

```
public class Question
{
   private String text;
   private String answer;
   . . .
```

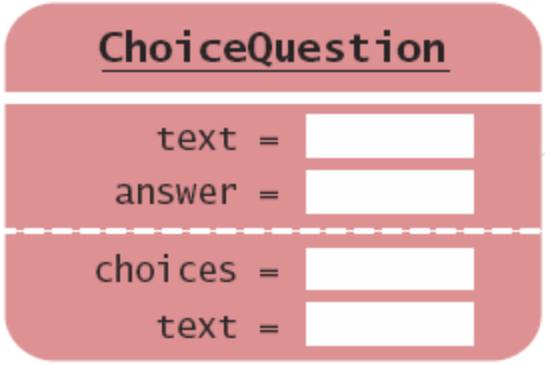
```
public class ChoiceQuestion extends Question
{
   . . .
   text = questionText;    // Complier Error!
```

Common Error 9.1 (2)

- Do not try to fix the compiler error with a new instance variable of the same name

```
public class ChoiceQuestion extends Question
{
   private String text;  // Second copy
```

- The constructor sets one `text` variable
- The display method outputs the other



Common Error 9.2

- Confusing *Super*- and *Sub*classes
  - The use of the terminology super and sub may be confusing
  - The Subclass `ChoiceQuestion` is an 'extended' and more powerful version of `Question`
    - Is it a 'super' version of `Question`?... NO.
- Super and Subclass terminology comes from set theory

  `ChoiceQuestion` is one of a **subset** of all objects that inherit from `Question`
  - The set of `Question` objects is a **superset** of `ChoiceQuestion` objects

9.3 Overriding Methods

- The `ChoiceQuestion` class needs a `display` method that overrides the `display` method of the `Question` class
- They are two different method implementations
- The two methods named `display` are:
  - `Question display`
    - Displays the instance variable text String
  - `ChoiceQuestion display`
    - Overrides `Question display` method
    - Displays the instance variable text String
    - Displays the local list of choices

Calling Superclass Methods

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

- Consider the `display` method of the `ChoiceQuestion` class
  - It needs to display the question AND the list of choices

- `text` is a private instance variable of the superclass
  - How do you get access to it to print the question?
  - Call the `display` method of the superclass `Question`!
    - From a subclass, preface the method name with:
    - `super.`

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

QuestionDemo2.java (1)

```
1  import java.util.Scanner;
2
3  /**
4     This program shows a simple quiz with two choice questions.
5  */
6  public class QuestionDemo2
7  {
8     public static void main(String[] args)
9     {
10       ChoiceQuestion first = new ChoiceQuestion();
11       first.setText("What was the original name of the Java language?");
12       first.addChoice("*7", false);
13       first.addChoice("Duke", false);
14       first.addChoice("Oak", true);
15       first.addChoice("Gosling", false);
16
17       ChoiceQuestion second = new ChoiceQuestion();
18       second.setText("In which country was the inventor of Java born?");
19       second.addChoice("Australia", false);
20       second.addChoice("Canada", true);
21       second.addChoice("Denmark", false);
22       second.addChoice("United States", false);
23
24       presentQuestion(first);
25       presentQuestion(second);
26    }
```

Creates two objects of the `ChoiceQuestion` class, uses new `addChoice` method.

Calls `presentQuestion` (next page)

QuestionDemo2.java (2)

```
28    /**
29       Presents a question to the user and checks the response.
30       @param q the question
31    */
32    public static void presentQuestion(ChoiceQuestion q)
33    {
34       q.display();
35       System.out.print("Your answer: ");
36       Scanner in = new Scanner(System.in);
37       String response = in.nextLine();
38       System.out.println(q.checkAnswer(response));
39    }
40 }
```

Uses ChoiceQuestion (subclass) display method.

ChoiceQuestion.java (1)

```
1  import java.util.ArrayList;
2
3  /**
4     A question with multiple choices.
5  */
6  public class ChoiceQuestion extends Question
7  {
8     private ArrayList<String> choices;
9
10    /**
11       Construct
12    */
13    public Choice
14    {
15       choices =
16    }
```

Inherits from `Question` class.

```
17
18    /**
19       Adds an answer choice to this question.
20       @param choice the choice to add
21       @param correct true if this is the correct choice, false otherwise
22    */
23    public void addChoice(String choice, boolean correct)
24    {
25       choices.add(choice);
26       if (correct)
27       {
28          // Convert choices.size() to string
29          String choiceString = "" + choices.size();
30          setAnswer(choiceString);
31       }
32    }
```

New `addChoice` method.

ChoiceQuestion.java (2)

```
33
34    public void display()
35    {
36       // Display the question text
37       super.display();
38       // Display the answer choices
39       for (int i = 0; i < choices.size(); i++)
40       {
41          int choiceNumber = i + 1;
42          System.out.println(choiceNumber + ": " + choices.get(i));
43       }
44    }
45 }
```

Overridden `display` method.

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
```

Common Error 9.3

- Accidental Overloading

```
println(int x);
println(String s);  // Overloaded
```

  - Remember that **overloading** is when two methods share the same name but have different parameters
  - **Overriding** is where a subclass defines a method with the same name and exactly the same parameters as the superclass method
    - `Question display() method`
    - `ChoiceQuestion display() method`
  - If you intend to **override**, but change parameters, you will be **overloading** the inherited method, not **overriding** it
    - `ChoiceQuestion display(printStream out) method`

Common Error 9.4

- Forgetting to use `super` when invoking a Superclass method
  - Assume that `Manager` inherits from `Employee`
    - `getSalary` is an overridden method of `Employee`
      - `Manager.getSalary` includes an additional bonus

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

Special Topic 9.1

- Calling the Superclass Constructor
  - When a subclass is instantiated, it will call the superclass constructor with no arguments
  - If you prefer to call a more specific constructor, you can invoke it by using replacing the superclass name with the reserved word `super` followed by `()`:

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

  - It must be the first statement in your constructor

Constructor with Superclass

- To initialize private instance variables in the superclass, invoke a specific constructor

The superclass constructor is called first.

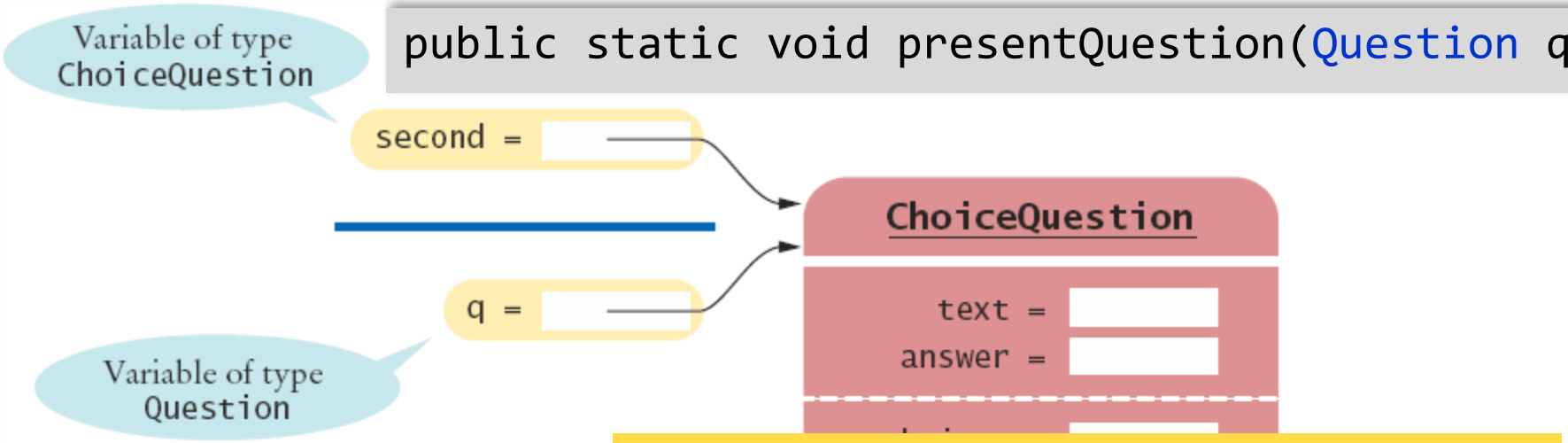
The constructor body can contain additional statements.

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

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

9.4 Polymorphism

- `QuestionDemo2` passed two `ChoiceQuestion` objects to the `presentQuestion` method
  - Can we write a `presentQuestion` method that displays both `Question` and `ChoiceQuestion` types?
  - **How would that work?**



A subclass reference can be used when a superclass reference is expected.

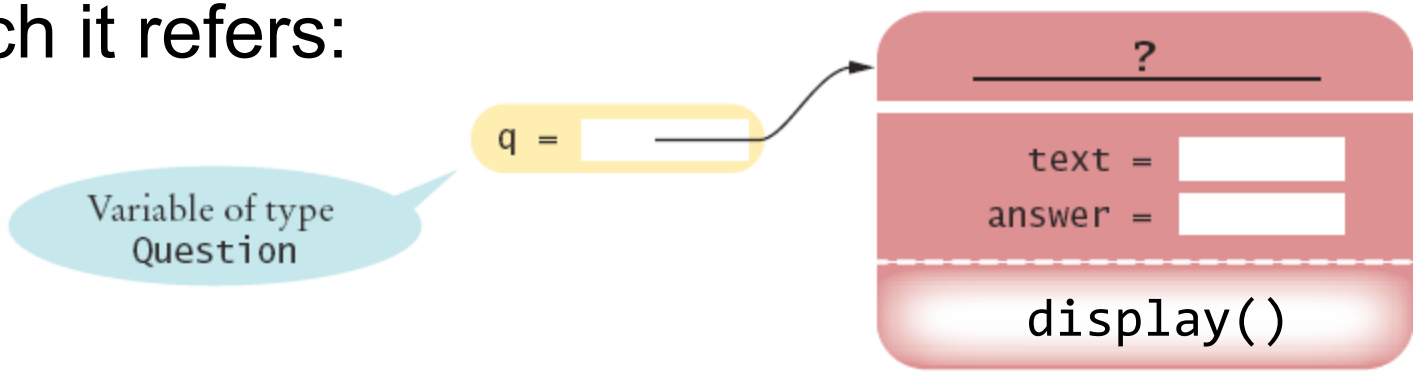
Which `display` method was called?

- `presentQuestion` simply calls the `display` method of whatever type is passed:

```
public static void presentQuestion(Question q)
{
   q.display();
   . . .
```

  - If passed an object of the `Question` class:
    - `Question display`
  - If passed an object of the `ChoiceQuestion` class:
    - `ChoiceQuestion display`

- The variable q does not know the type of object to which it refers:



Polymorphism Benefits

- In Java, method calls *are always determined by the type of the actual object,* **not** the type of the variable containing the object reference
  - 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 makes programs *easily extensible*

QuestionDemo3.java (1)

```
1  import java.util.Scanner;
2
3  /**
4     This program shows a simple quiz with two question types.
5  */
6  public class QuestionDemo3
7  {
8     public static void main(String[] args)
9     {
10       Question first = new Question();
11       first.setText("Who was the inventor of Java?");
12       first.setAnswer("James Gosling");
13
14       ChoiceQuestion second = new ChoiceQuestion();
15       second.setText("In which country was the inventor of Java born?");
16       second.addChoice("Australia", false);
17       second.addChoice("Canada", true);
18       second.addChoice("Denmark", false);
19       second.addChoice("United States", false);
20
21       presentQuestion(first);
22       presentQuestion(second);
23    }
24 }
```

Creates an object of the `Question` class

Creates an object of the `ChoiceQuestion` class, uses new `addChoice` method.

Calls `presentQuestion` (next page) passing both types of objects.

QuestionDemo3.java (2)

```
24
25    /**
26       Presents a question to the user and checks the response.
27       @param q the question
28    */
29    public static void presentQuestion(Question q)
30    {
31       q.display();
32       System.out.print("Your answer: ");
33       Scanner in = new Scanner(System.in);
34       String response = in.nextLine();
35       System.out.println(q.checkAnswer(response));
36    }
37 }
```

Receives a parameter of the super-class type

Uses appropriate display method.

Special Topic 9.2

- Dynamic Method Lookup and the Implicit Parameter
  - Suppose we move the `presentQuestion` method to inside the `Question` class and call it as follows:

```
ChoiceQuestion cq = new ChoiceQuestion();
cq.setText("In which country was the inventor of Java born?");
. . .
cq.presentQuestion();
```

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

  - Which `display` and `checkAnswer` methods will be called?

Dynamic Method Lookup

- Add the Implicit Parameter to the code to find out
  - Because of dynamic method lookup, the `ChoiceQuestion` versions of the `display` and `checkAnswer` methods are called automatically.
  - This happens even though the `presentQuestion` method is declared in the `Question` class, which has no knowledge of the `ChoiceQuestion` class.

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

Special Topic 9.3

- Abstract Classes
  - If it is desirable to ***force*** subclasses to override a method of a base class, you can declare a method as `abstract`.
  - You cannot instantiate an object that has `abstract` methods
    - Therefore the class is considered `abstract`

```
public abstract class Account
{
  public abstract void deductFees();  // no method implementation
. . .
```

```
public class SavingsAccount extends Account // Not abstract
{
  public void deductFees() // Provides an implementation
  {   // method implementation. . .  }
  . . .
}
```

  - If you extend the `abstract` class, you must implement all abstract methods.

Abstract References

- A class that can be instantiated is called `concrete` class
- You cannot instantiate an object that has `abstract` methods
  - But you can declare an object reference whose type is an `abstract` class.
  - The actual object to which it refers must be an instance of a `concrete` subclass

```
Account anAccount;          // OK: Reference to abstract object
anAccount = new Account(); // Error: Account is abstract
anAccount = new SavingsAccount(); // Concrete class is OK
anAccount = null;          // OK
```

  - This allows for polymorphism based on even an `abstract` class!

One reason for using abstract classes is to force programmers to create subclasses.

Special Topic 9.4

- Final Methods and Classes
  - You can also ***prevent*** programmers from creating subclasses and override methods using `final`.
  - The String class in the Java library is an example:

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

  - Example of a method that cannot be overridden:

```
public class SecureAccount extends BankAccount
{
   . . .
   public final boolean checkPassword(String password)
   {
      . . .
   }
}
```

Special Topic 9.5

- `protected` Access
  - When trying to implement the display method of the `ChoiceQuestion` class, the `display` method wanted to access the instance variable `text` of the superclass, but it was `private`.
  - We chose to use a method of the superclass to display the text.
- Java provides a more elegant solution
  - The superclass can declare an instance variable as `protected` instead of `private`
  - `protected` data in an object can be accessed by the methods of the object's class and all its subclasses.
  - But it can also be accessed by all other classes in the same package!

```
public class Question
{
   protected String text;
   . . .
}
```

Steps to Using Inheritance

- As an example, we will consider a bank that offers customers the following account types:
  1) A savings account that earns interest. The interest compounds monthly and is based on the minimum monthly balance.
  2) A checking account that has no interest, gives you three free withdrawals per month, and charges a $1 transaction fee for each additional withdrawal.
- The program will manage a set of accounts of both types
  - It should be structured so that other account types can be added without affecting the main processing loop.
- The menu: `D)eposit W)ithdraw M)onth end Q)uit`
  - For deposits and withdrawals, query the account number and amount. Print the balance of the account after each transaction.
  - In the "Month end" command, accumulate interest or clear the transaction counter, depending on the type of the bank account. Then print the balance of all accounts.

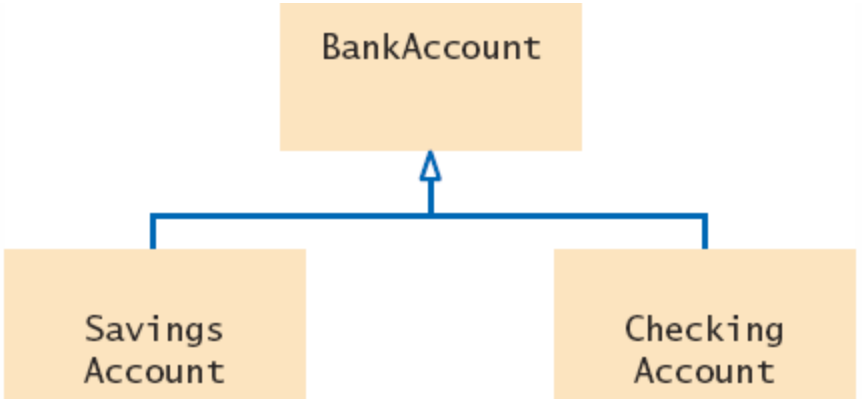
Steps to Using Inheritance

1) List the classes that are part of the hierarchy.

```
SavingsAccount
CheckingAccount
```

2) Organize the classes into an inheritance hierarchy

Base on superclass `BankAccount`



3) Determine the common responsibilities.

a. Write *Pseudocode* for each task

b. Find common tasks

Using Inheritance

```
For each user command
    If it is a deposit or withdrawal
        Deposit or withdraw the amount from the specified account.
        Print the balance.
    If it is month end processing
        For each account
            Call month end processing.
            Print the balance.
```

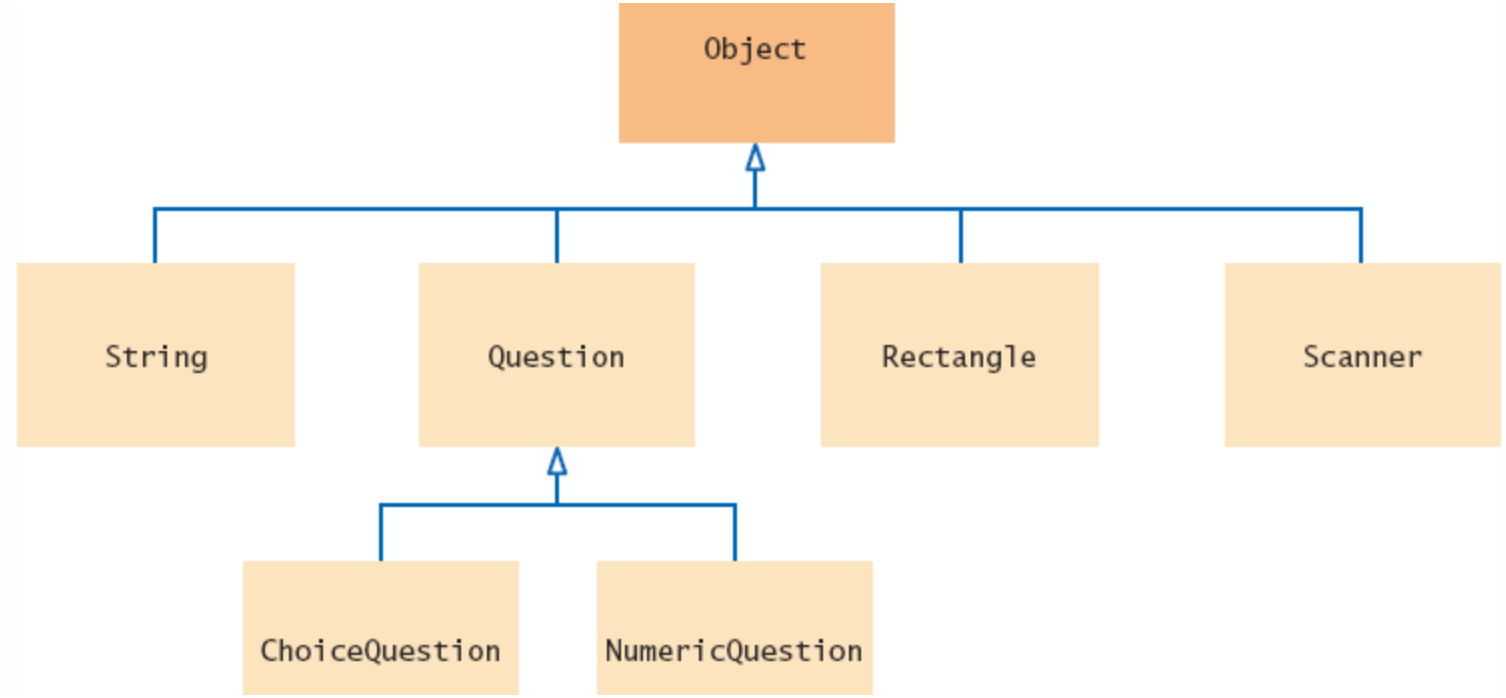
```
Deposit money.
Withdraw money.
Get the balance.
Carry out month end processing.
```

Steps to Using Inheritance

4) Decide which methods are overridden in subclasses.
   - For each subclass and each of the common responsibilities, decide whether the behavior can be inherited or whether it needs to be overridden
5) Declare the public interface of each subclass.
   - Typically, subclasses have responsibilities other than those of the superclass. List those, as well as the methods that need to be overridden.
   - You also need to specify how the objects of the subclasses should be constructed.
6) Identify instance variables.
   - List the instance variables for each class. Place instance variables that are common to all classes in the base of the hierarchy.
7) Implement constructors and methods.
8) Construct objects of different subclasses and process them.

9.5 `Object`: The Cosmic Superclass

- In Java, every class that is declared without an explicit extends clause automatically extends the class `Object`.



The methods of the `Object` class are very general. You will learn to override the `toString` method.

Writing a `toString` method

- The `toString` method returns a `String` representation for each object.
- The `Rectangle` class (java.awt) has a `toString` method
  - You can invoke the `toString` method directly

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

  - The `toString` method can also be invoked implicitly whenever you concatenate a `String` with an object:

```
System.out.println("box=" + box);   // Call toString implicitly
```

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

Overriding the `toString` method

- Example: Override the `toString` method for the BankAccount class

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

  - All that is printed is the name of the class, followed by the hash code which can be used to tell objects (Chapter 10)

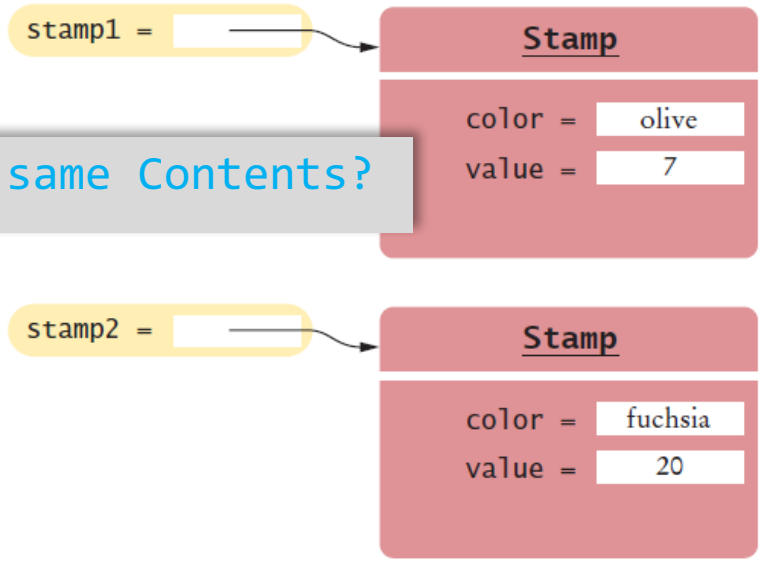
- We want to know what is inside the object

```
public class BankAccount
{
   public String toString()
   {
      // returns "BankAccount[balance=5000]"
      return "BankAccount[balance=" + balance + "]";
   }
}
```

Override the toString method to yield a string that describes the object's state.

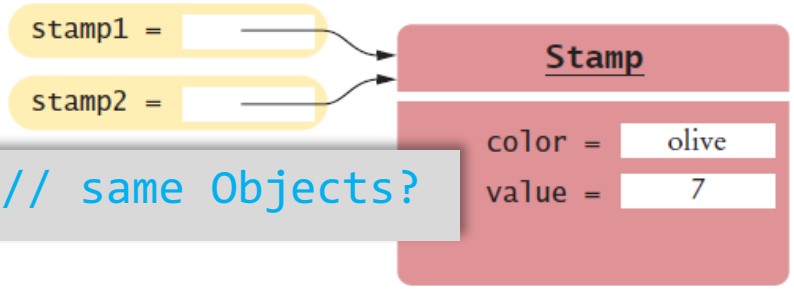
Overriding the `equals` method

- In addition to the `toString` method, the Object class `equals` method checks whether two objects have the same contents:




```
if (stamp1.equals(stamp2)) . . . // same Contents?
```

- This is different from the == operator which compares the two references:




```
if (stamp1 == stamp2) . . . // same Objects?
```

Overriding the `equals` method

- The Object class specifies the type of parameter as `Object`

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

The `Stamp equals` method must declare the same type of parameter as the `Object equals` method to override it.

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

Cast the parameter variable to the class `Stamp`

The `instanceof` operator

- You can store a subclass reference in a variable declared as superclass reference type
- The opposite conversion is also possible:
  - From a superclass reference to a subclass reference
  - If you have a variable of type `Object`, and you know that it actually holds a `Question` reference, you can cast it:

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

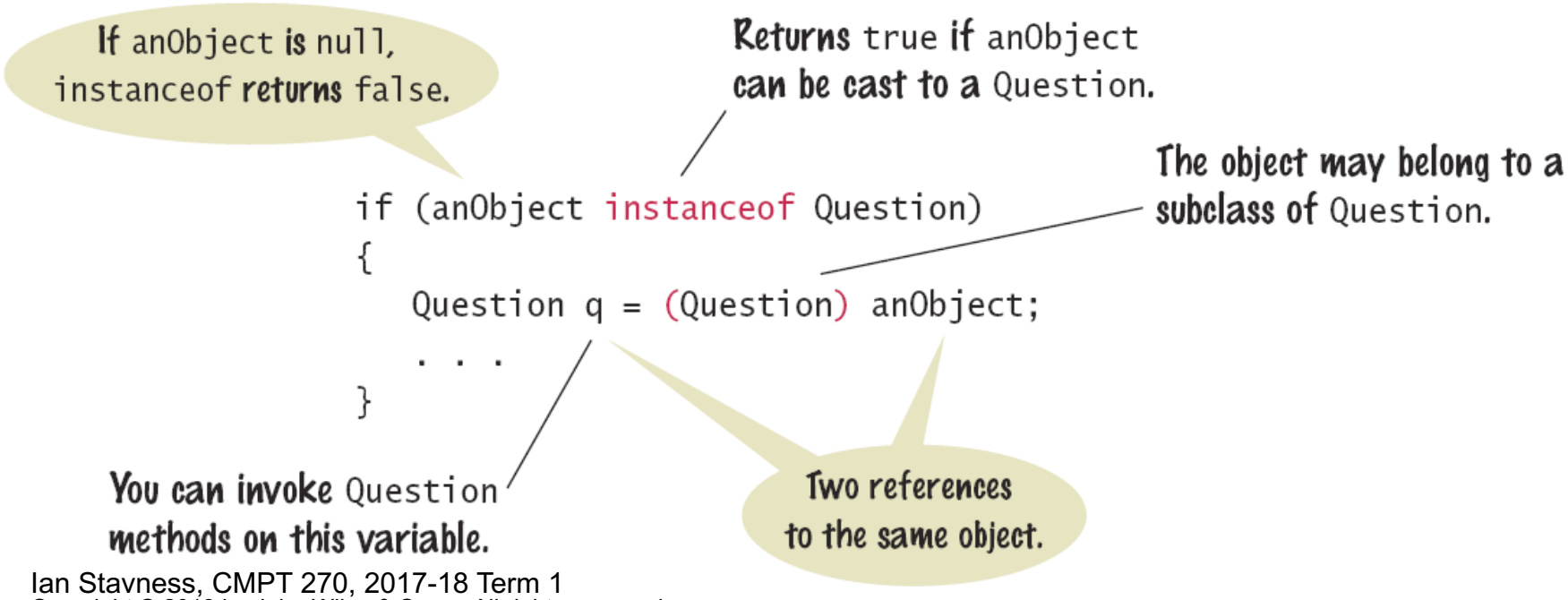
- To make sure it is an object of the `Question` type, you can test it with the `instanceof` operator:

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

`instanceof` returns a boolean

Syntax 9.3: Using `instanceof`

- Using the `instanceOf` operator also involves casting
  - Returns true if you can safely cast one object to another
- Casting allows the use of methods of the new object
  - Most often used to make a reference more specific
    - Cast from an `Object` reference to a more specific class type



Common Error 9.5

- Don't Use Type Tests

```
if (q instanceof ChoiceQuestion)) // Don't do this
{
   // Do the task the ChoiceQuestion way
}
else if (q instanceof Question))
{
   // Do the task the Question way
}
```

  - This is a poor strategy. If a new class is added, then all these queries need to be revised.
    - When you add the class `NumericQuestion`
  - Let polymorphism select the correct method:
    - Declare a method `doTheTask` in the superclass
    - Override it in subclasses

Special Topic 9.6

- Inheritance and the `toString` Method
  - Instead of writing the type of object in a `toString` method
    - Use `getclass` (inherited from object) in the superclass

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

  - Then use inheritance, call the superclass `toString` first

```
public class SavingsAccount extends BankAccount
{
   . . .
   public String toString()
   {
      return super.toString() + "[interestRate=" + intRate + "]";
   }  // returns SavingsAccount[balance= 10000][interestRate= 5]
}
```

This allows the superclass to output private instance variables

Special Topic 9.7

- Inheritance and the `equals` Method
  - What if someone called stamp1.equals(x) where x was not a Stamp object?
    - Using the instanceOf operator, t would be possible for otherObject to belong to some subclass of Stamp.
  - Use the `getClass` method to compare your exact class to the passed object to make sure

Insures comparison of the same types

```
public boolean equals(Object otherObject)
{
  if (otherObject == null) { return false; }
  if (getClass() != otherObject.getClass()) { return false; }
  Stamp other = (Stamp) otherObject;
  return color.equals(other.color) && value == other.value;
}
```

9.6 Interface Types

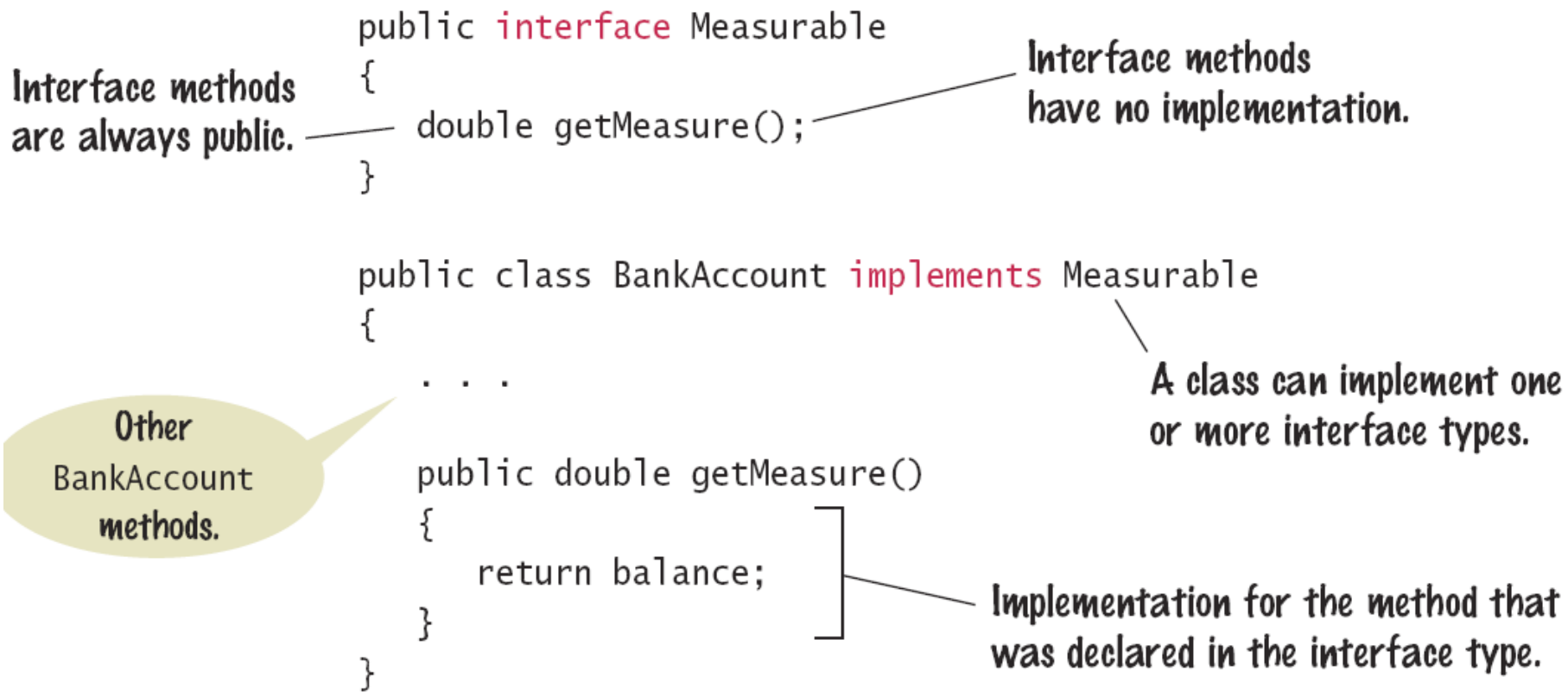
- An interface is a special type of declaration that lists a set of methods and their signatures
  - A class that '*implements*' the interface must implement all of the methods of the interface
  - It is similar to a class, but there are differences:
    - All methods in an interface type are abstract:
      They have a name, parameters, and a return type, but they don't have an implementation
    - All methods in an interface type are automatically public
    - An interface type cannot have instance variables
    - An interface type cannot have static methods

```
public interface Measurable
{
   double getMeasure();
}
```

A Java `interface` type declares a set of methods and their signatures.

Syntax 9.4: Interface Types

- An interface declaration and a class that `implements` the `interface`.



Using Interface Types

- We can use the interface type `Measurable` to implement a "universal" static method for computing averages:

```
public interface Measurable
{
   double getMeasure();
}
```

```
public static double average(Measurable[] objs)
{
   if (objs.length == 0) return 0;
   double sum = 0;
   for (Measurable obj : objs)
   {
      sum = sum + obj.getMeasure();
   }
   return sum / objs.length;
}
```

Implementing an Interface

- A class can be declared to `implement` an interface
  - The class must implement all methods of the interface

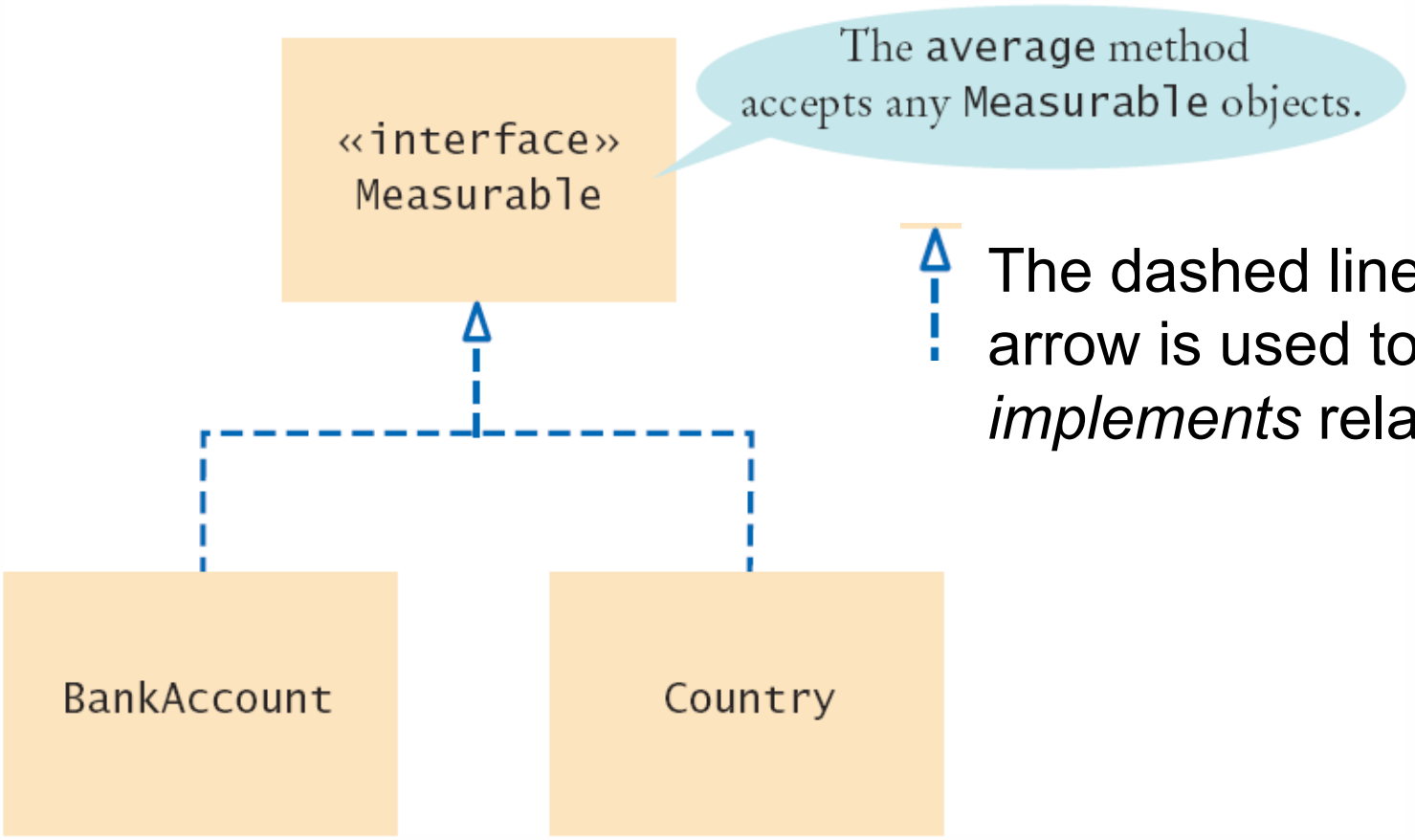
```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
      return balance;
   }
   . . .
}
```

Use the `implements` reserved word in the class declaration.

```
public class Country implements Measurable
{
   public double getMeasure()
   {
      return area;
   }
   . . .
}
```

The methods of the interface must be declared as `public`

An Implementation Diagram



The dashed line with an arrow is used to denote *implements* relationships

MeasureableDemo.java (1)

```
1  /**
2     This program demonstrates the measurable BankAccount and Country classes.
3  */
4  public class MeasurableDemo
5  {
6     public static void main(String[] args)
7     {
8        Measurable[] accounts = new Measurable[3];
9        accounts[0] = new BankAccount(0);
10       accounts[1] = new BankAccount(10000);
11       accounts[2] = new BankAccount(2000);
12
13       System.out.println("Average balance: "
14          + average(accounts));
15
16       Measurable[] countries = new Measurable[3];
17       countries[0] = new Country("Uruguay", 176220);
18       countries[1] = new Country("Thailand", 514000);
19       countries[2] = new Country("Belgium", 30510);
20
21       System.out.println("Average area: "
22          + average(countries));
23    }
```

MeasureableDemo.java (2)

```
25    /**
26       Computes the average of the measures of the given objects.
27       @param objs an array of Measurable objects
28       @return the average of the measures
29    */
30    public static double average(Measurable[] objs)
31    {
32       if (objs.length == 0) { return 0; }
33       double sum = 0;
34       for (Measurable obj : objs)
35       {
36          sum = sum + obj.getMeasure();
37       }
38       return sum / objs.length;
39    }
40 }
```

Program Run

```
Average balance: 4000.0
Average area: 240243.33333333334
```

The Comparable Interface

- The Java library includes a number of important interfaces including Comparable
  - It requires implementing one method: `compareTo()`
  - It is used to compare two objects
  - It is implemented by many objects in the Java API
  - If may want to implement it in your classes to use powerful Java API tools such as sorting
- It is called on one object, and is passed another
  - Called on object a, return values include:
    - Negative: a comes before b
    - Positive: a comes after b
    - 0: a is the same as b

```
a.compareTo(b);
```

The Comparable Type parameter

- The Comparable interface uses a special type of parameter that allows it to work with any type:

```
public interface Comparable<T>
{
  int compareTo(T other);
}
```

  - The type <T> is a placeholder for an actual type of object
  - The class ArrayList class uses the same technique with the type surrounded by angle brackets < >

```
ArrayList<String> names = new ArrayList<String>();
```

Using the type inside angle braces will be covered further in the next chapter.

A Comparable Example

- The BankAccount compareTo method compares bank accounts by their balance.
  - It takes one parameter of it's own class type (BankAccount)

```
public class BankAccount implements Comparable<BankAccount>
{
   . . .
   public int compareTo(BankAccount other)
   {
      if (balance < other.getBalance()) { return -1; }
      if (balance > other.getBalance()) { return 1; }
      return 0;
   }
   . . .
}
```

The methods of the interface must be declared as public

Using compareTo to Sort

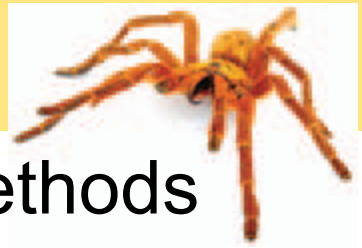
- The `Arrays.sort` method uses the `compareTo` method to sort the elements of the array
  - Once the `BankAccount` class implements the `Comparable` interface, you can sort an array of bank accounts with the `Arrays.sort` method:

```
BankAccount[] accounts = new BankAccount[3];
accounts[0] = new BankAccount(10000);
accounts[1] = new BankAccount(0);
accounts[2] = new BankAccount(2000);
Arrays.sort(accounts);
```

  - The array is now sorted by increasing balance

Implementing Java Library interfaces allows you to use the power of the Java Library with your classes.

Common Error 9.6



- Forgetting to Declare Implementing Methods as Public
  - The methods in an interface are not declared as public, because they are public by default.
  - However, the methods in a class are not public by default.
  - It is a common error to forget the public reserved word when declaring a method from an interface:

```
public class BankAccount implements Measurable
{
   double getMeasure()   // Oops—should be public
   {
      return balance;
   }
   . . .
}
```

Special Topic 9.8

- Interface Constants
  - Interfaces cannot have instance variables, but it is legal to specify constants
  - When declaring a constant in an interface, you can (and should) omit the reserved words `public static final`, because all variables in an interface are automatically `public static final`.

```
public interface SwingConstants
{
   int NORTH = 1;
   int NORTHEAST = 2;
   int EAST = 3;
   . . .
}
```

Special Topic 9.9

- Function Objects
  - Interfaces work well IF all objects that need the service are willing to implement the interface.
  - The sole purpose of a function object is to execute a single method
    - This allows a non-implementing class to use the services of the interface by creating a function object and using it's method
  - First, create a new interface
    - The measure method measures an object and returns its measurement. We use a parameter of type `Object`, the "lowest common denominator" of all classes in Java, because we do not want to restrict which classes can be measured.

```
public interface Measurer
{
   double measure(Object anObject);
}
```

Function Objects(2)

- Then declare a class that implements the new interface

```
public class StringMeasurer implements Measurer
{
  public double measure(Object obj)
  {
    String str = (String) obj; // Cast obj to String type
    return str.length();
  }
}
```

```
public interface Measurer
{
  double measure(Object anObject);
}
```

Function Objects (3)

- Example of Function Object Use
  - Instantiate an object of the Function object class
  - Call your method that accepts an object of this type

```
String[] words = { "Mary", "had", "a", "little", "lamb" };
Measurer strMeas = new StringMeasurer();
double result = average(words, strMeas);
```

```
public static double average(Object[] objs, Measurer meas)
{
  if (objs.length == 0) { return 0; }
  double sum = 0;
  for (Object obj : objs)
  {
    sum = sum + meas.measure(obj);
  }
  return sum / objs.length;
}
```

See ch09/measure2 sample program.

Summary: Inheritance

- A subclass inherits data and behavior from a superclass.
- You can always use a subclass object in place of a superclass object.
- A subclass inherits all methods that it does not override.
- A subclass can override a superclass method by providing a new implementation.

Summary: Overriding Methods

- An overriding method can extend or replace the functionality of the superclass method.
- Use the reserved word `super` to call a superclass method.
- Unless specified otherwise, the subclass constructor calls the superclass constructor with no arguments.
- To call a superclass constructor, use the `super` reserved word in the first statement of the subclass constructor.
- The constructor of a subclass can pass arguments to a superclass constructor, using the reserved word `super`.

Summary: Polymorphism

- A subclass reference can be used when a superclass reference is expected.
- Polymorphism ("having multiple shapes") allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.
- An `abstract` method is a method whose implementation is not specified.
- An `abstract` class is a class that cannot be instantiated.

Summary: `toString` and `instanceOf`

- Override the `toString` method to yield a `String` that describes the object's state.
- The `equals` method checks whether two objects have the same contents.
- If you know that an object belongs to a given class, use a cast to convert the type.
- The `instanceof` operator tests whether an object belongs to a particular type.

Summary: Interfaces

- The Java `interface` type contains the return types, names, and parameter variables of
- Unlike a class, an `interface` type provides no implementation.
- By using an interface type for a parameter variable, a method can accept objects from many classes.
- The `implements` reserved word indicates which interfaces a class implements.
- Implement the `Comparable` interface so that objects of your class can be compared, for example, in a sort method.