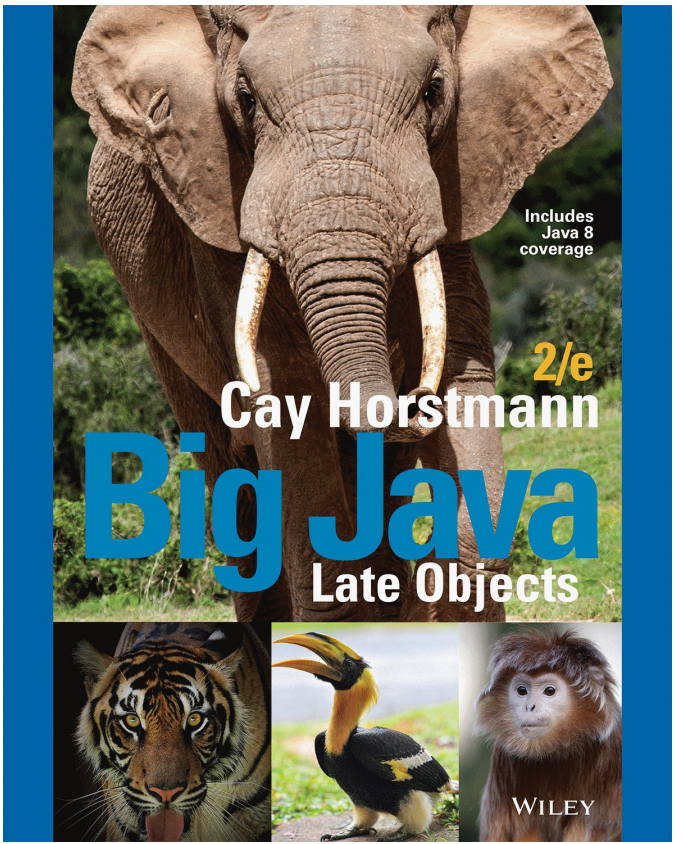
Chapter 9 - Inheritance and Interfaces

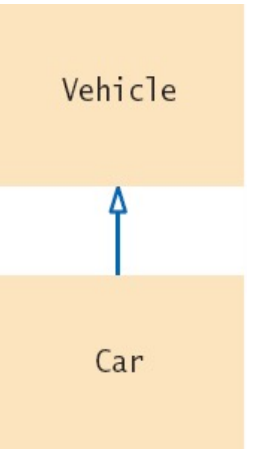


Chapter Goals

- 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

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

Vehicle

Motorcycle

Car

Truck

Sedan

SUV

The Substitution Principle

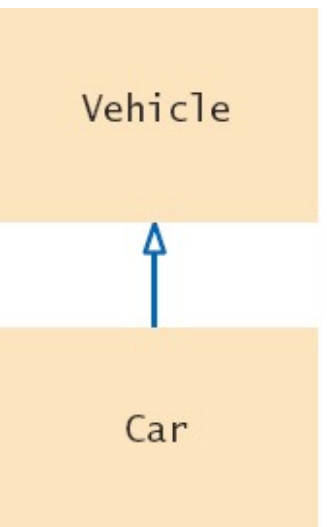
The **substitution principle** states that you can always use a *subclass* object when a *superclass* object is expected

- 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

```
Void processVehicle(Vehicle v)
```

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

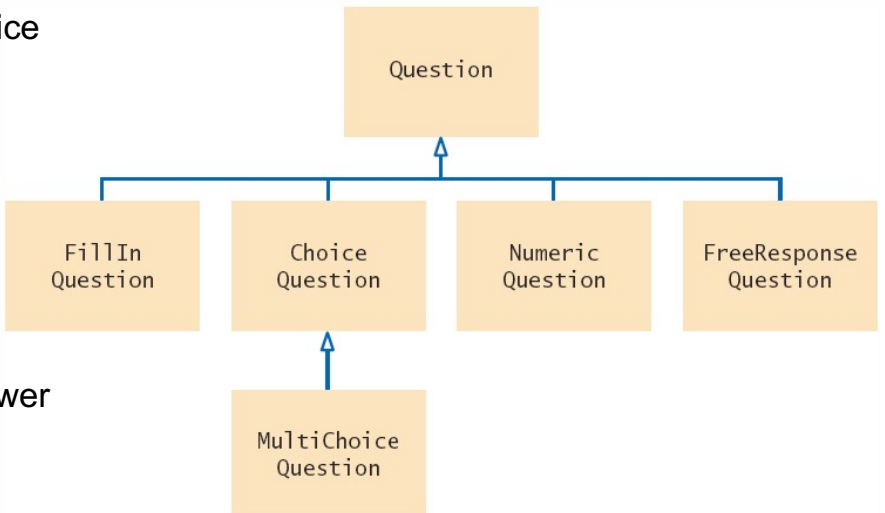
The 'is-a' relationship is represented by an arrow in a class diagram and means that the subclass can behave as an object of the superclass.



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)

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

```
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.

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.

Self Check 9.1

Consider classes `Manager` and `Employee`. Which should be the superclass and which should be the subclass?

Answer: Because every manager is an employee but not the other way around, the `Manager` class is more specialized. It is the subclass, and `Employee` is the superclass.

Programming Tip

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

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

  - If two vehicles behave differently,
  - use inheritance

Be careful not to over-use inheritance

Car

Sedan

SUV

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
```

- 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

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

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

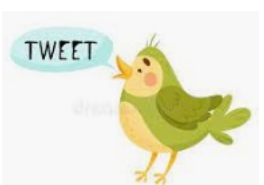
Form a subclass by specifying what is different from the superclass.

You **declare** any methods that are new to the subclass and **change** the implementation of inherited methods if the inherited behavior is not appropriate.

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

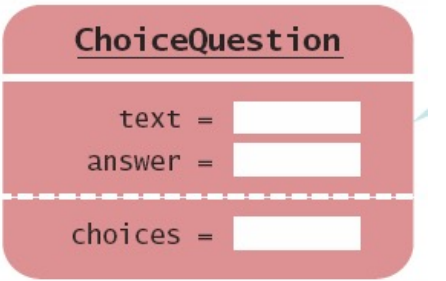
How do animals talk?



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

Syntax

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

The reserved word extends denotes inheritance.

Subclass

Superclass

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

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

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

Declare instance variables that are added to the subclass.

Declare methods that are added to the subclass.

Declare methods that the subclass overrides.

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()`

```
question.addChoice("Canada", true);
```

if choices.size() is 2, then answer is set to the string "2".

Self Check 9.6

Suppose q is an object of the class `Question` and `cq` an object of the class `ChoiceQuestion`. Which of the following calls are legal?

a. `q.setAnswer(response)`

b. `cq.setAnswer(response)`

c. `q.addChoice(choice, true)`

d. `cq.addChoice(choice, true)`

Answer: a, b, d

Self Check 9.7

Suppose the class `Employee` is declared as follows:

```
public class Employee
{
   private String name;
   private double baseSalary;

   public void setName(String newName) { . . . }
   public void setBaseSalary(double newSalary) { . . . }
   public String getName() { . . . }
   public double getSalary() { . . . }
}
```

Declare a class `Manager` that inherits from the class `Employee` and adds an instance variable `bonus` for storing a salary bonus. Omit constructors and methods.

Answer:

```
public class Manager extends Employee
{
   private double bonus;
   // Constructors and methods omitted
}
```

Common Error

- A subclass has **no access** to the private instance variables of the superclass.
- A subclass cannot directly access private instance variables of the superclass

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

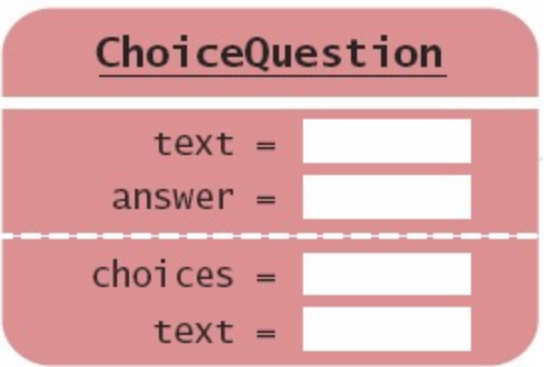
```
public class ChoiceQuestion extends Question
{
   . . .
   text = questionText;    // Complier Error!
```

Common Error (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



9.3 Overriding Methods

- The subclass inherits the methods from the superclass
- 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

- Consider the display method of the `ChoiceQuestion` class
  - It needs to display the question **AND** the list of choices
- The second task is easy
  - We have an instance variable

```
public class ChoiceQuestion
{
   . . .
   public void display()
   {
      // Display the question text
      . . .
      // Display the answer choices
      for (int i = 0; i < choices.size(); i++)
      {
         int choiceNumber = i + 1;
         System.out.println(choiceNumber + ": " + choices.get(i));
      }
   }
}
```

Calling Superclass Methods

- Displaying the question is tricky
- `text` is a **private** instance variable of the superclass
  - You *can't access the text variable of the super- class directly* because it is private
  - 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`**
- Can we call `this.display()`?

Error

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

```
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 Array
9
10    /**
11       Construct
12    */
13    public Choic
14    {
15       choices =
16    }
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    }
```

Inherits from `Question` class.

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

```
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
```

Self Check 9.11

What is wrong with the following implementation of the display method?

```
public class ChoiceQuestion
{
   . . .
   public void display()
   {
      System.out.println(text);
      for (int i = 0; i < choices.size(); i++)
      {
         int choiceNumber = i + 1;
         System.out.println(choiceNumber + ": " + choices.get(i));
      }
   }
}
```

Answer: The method is not allowed to access the instance variable `text` from the superclass.

Self Check 9.12

What is wrong with the following implementation of the display method?

```
public class ChoiceQuestion
{
   . . .
   public void display()
   {
      this.display();
      for (int i = 0; i < choices.size(); i++)
      {
         int choiceNumber = i + 1;
         System.out.println(choiceNumber + ": " + choices.get(i));
      }
   }
}
```

Answer: The type of the `this` reference is `ChoiceQuestion`. Therefore, the `display` method of `ChoiceQuestion` is selected, and the method calls itself.

Common Error

- 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

- 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;
   }
}
```

recursive call, which will never stop
Implicit ***this***

Special Topic

- Calling the Superclass Constructor
  - The **superclass** instance variables also need to be **initialized**
  - 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**
  - Remember, these are different:
    - `super(questionText)`
    - `super.display()`

Syntax 9.2 Constructor with Superclass Initializer

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

Syntax

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

The superclass constructor is called first.

The constructor body can contain additional statements.

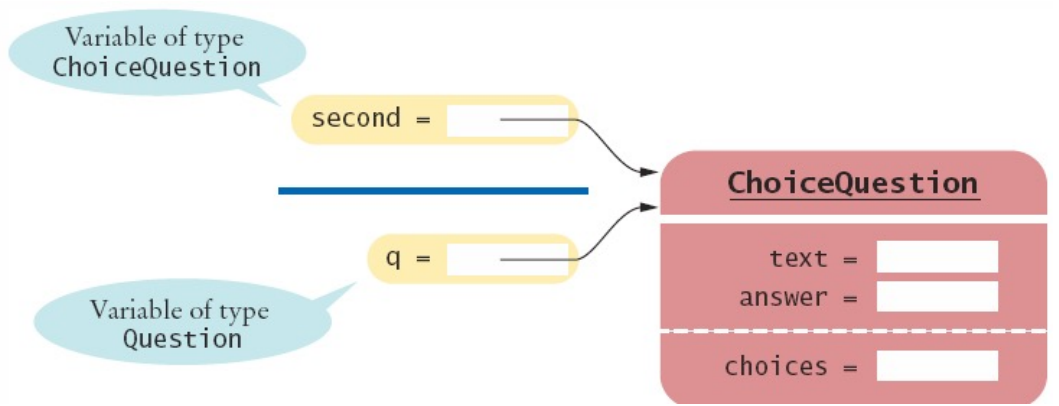
```
public ChoiceQuestion(String questionText)
{
   super(questionText);
   choices = new ArrayList<String>;
}
```

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?**

```
public static void presentQuestion(Question q)
```



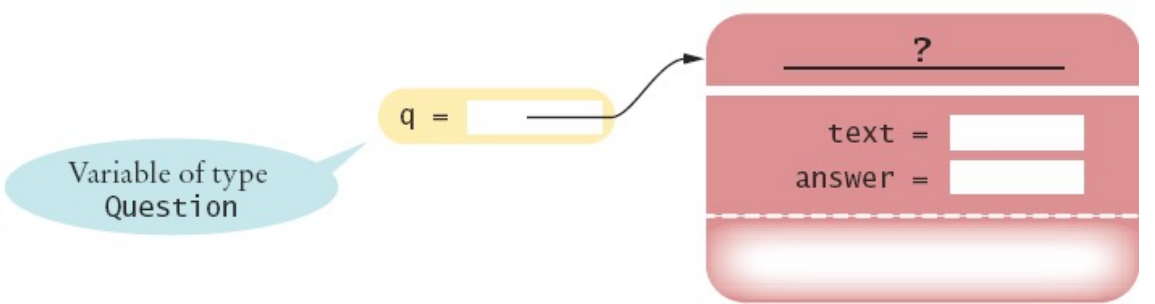
A subclass reference can be used when a superclass reference is expected.

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:



Which `display` Method Was Called?

```
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));
}
```

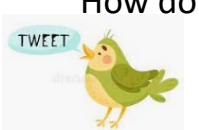
```
ChoiceQuestion second = new ChoiceQuestion();
. . .
presentQuestion(second); // OK to pass a ChoiceQuestion
```

You cannot call the **`addChoice`** method, though—it is not a method of the `Question` superclass.

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*
  - A new kind of question for calculations (i.e., accept an approximate answer)
  - Declare a new class `NumericQuestion` that extends Question
    - With its own `checkAnswer()`
  - Call the same `presentQuestion()`, **no changes needed!**

How do animals talk?



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.

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
```

Self Check 9.16

Assuming `SavingsAccount` is a subclass of `BankAccount`, which of the following code fragments are valid in Java?

a. `BankAccount account = new SavingsAccount();`

b. `SavingsAccount account2 = new BankAccount();`

c. `BankAccount account = null;`

d. `SavingsAccount account2 = account;`

Answer: a and c.

Self Check 9.18

Declare an array quiz that can hold a mixture of `Question` and `ChoiceQuestion` objects.

Answer: `Question[] quiz = new Question[SIZE];`

Special Topic

- When you extend an existing class, you have the **choice** whether or not to override the methods of the superclass
- 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
. . .
```

  - An abstract method has **no 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

- 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 { . . . }
```

    - Nobody can extend the String class.
  - Example of a method that cannot be overridden:

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

Special Topic

- `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;
   . . .
}
```

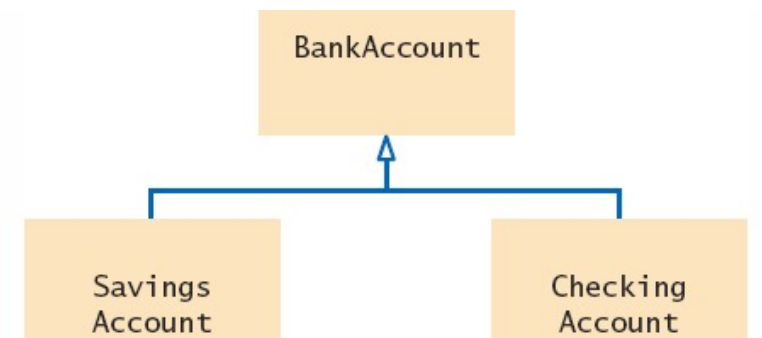
If you want to grant access to the data to subclass methods only, consider making the **accessor method protected.**

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



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.

9.6 Interface Types

Sometimes, the algorithm for computing the result is the same, but the details of measurement differ.
We need to focus on the essential operations that an algorithm needs.

Avg balance

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

Avg of areas

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

How can we write a single method that computes the averages of both bank accounts and countries?

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.

An interface type is used to specify **required** operations.

If a class implements an interface and **does not provide method bodies** for all functions specified in the interface, then the class must be declared **abstract**.

Syntax 9.4 Interface Types

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

Syntax

Declaring:

```
public interface InterfaceName
{
   method declarations
}
```

Implementing:

```
public class ClassName implements InterfaceName, InterfaceName, . . .
{
   instance variables
   methods
}
```

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

public class BankAccount implements Measurable
{
   . . .

   public double getMeasure()
   {
      return balance;
   }
}
```

Interface methods are automatically public.

Abstract methods have no implementation.

A class can implement one or more interface types.

Other BankAccount methods.

Implementation for the abstract method that was declared in the interface type.

- A class can implement more than one 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;
}
```

The interface declaration lists all methods that the interface type requires.

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;
   }
   . . .
}
```

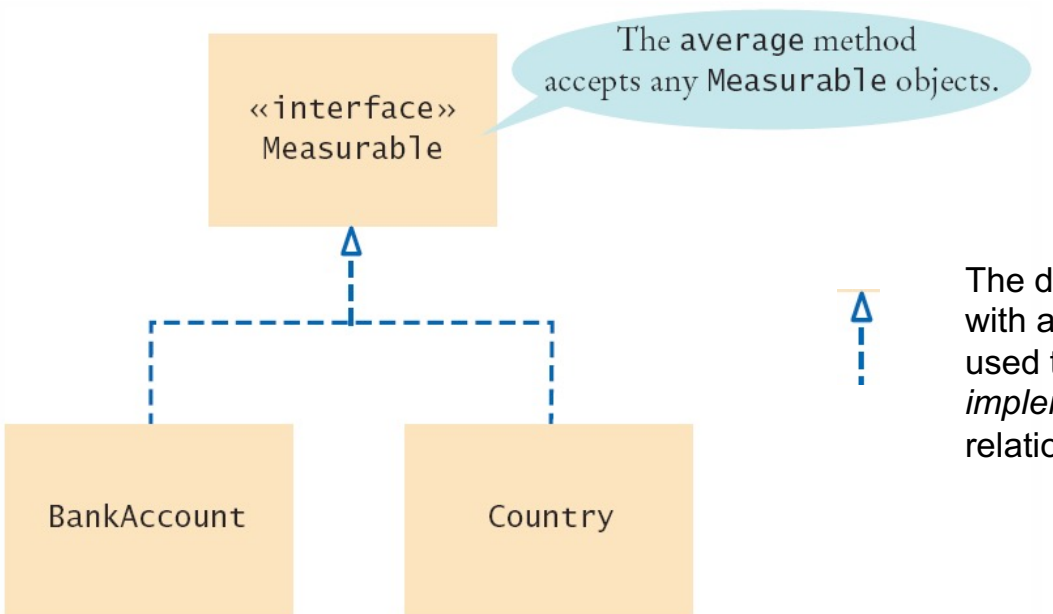
Use the `implements` reserved word in the class declaration.

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

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

An Implementation Diagram

The `average` method is usable for objects of **any class** that implements the `Measurable` interface.



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 standard 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
  - You 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, is passed b, and return values include:
    - Negative: a comes before b
    - Positive: a comes after b
    - 0: a is the same as b

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

Implement the `Comparable` interface so that objects of your class can be compared

```
public interface Comparable
{
   int compareTo(Object otherObject);
}
```

Implementing an Interface

- Defining the `compareTo` method ensures that `BankAccount` implements the `Comparable` interface

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

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.

Self Check 9.26

Suppose you want to use the `average` method to find the average salary of `Employee` objects. What condition must the `Employee` class fulfill?

Answer: It must implement the `Measurable` interface and provide a `getMeasure` method returning the salary.

Special Topic

- 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;
   . . .
}
```

```
SwingConstants.NORTH;
```