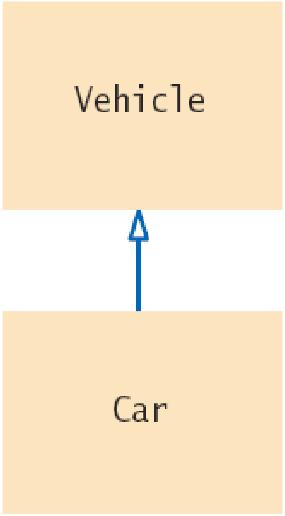
INHERITANCE AND INTERFACES

CENG 522

Inheritance Hierarchies

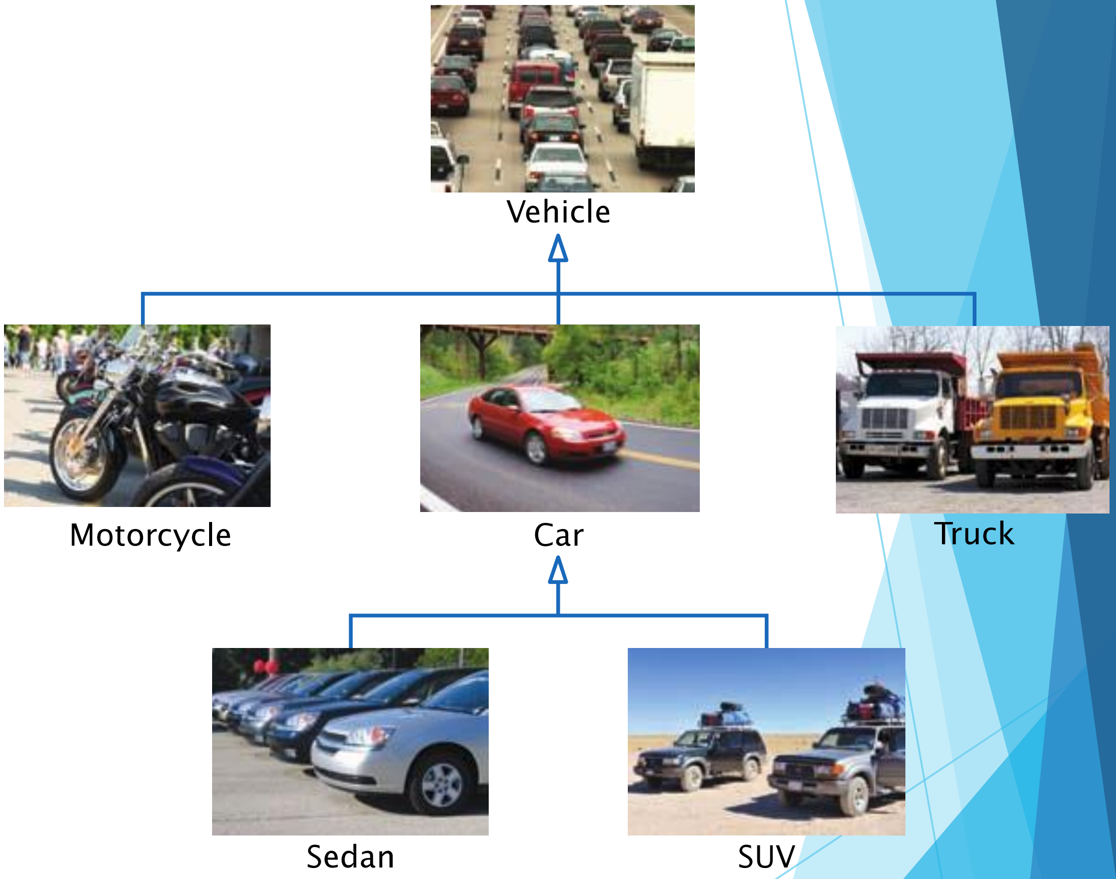
- In object-oriented programming, inheritance is a relationship between:
  - A *superclass*: a more generalized class
  - A *subclass*: a more specialized class



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

A Vehicle Class Hierarchy

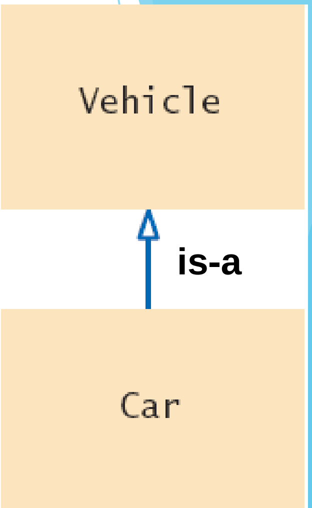
- General
- Specialized
- More Specific



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);
```

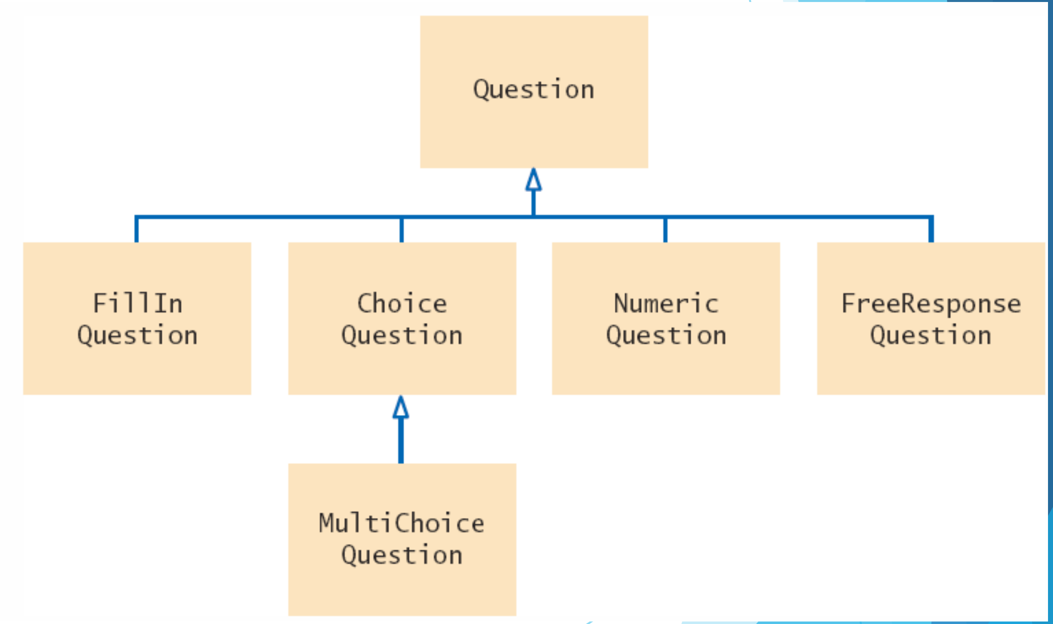


The ‘is-a’ relationship is represented by an arrow in a class diagram and means that the subclass can behave as an object of the superclass.

Quiz Question Hierarchy

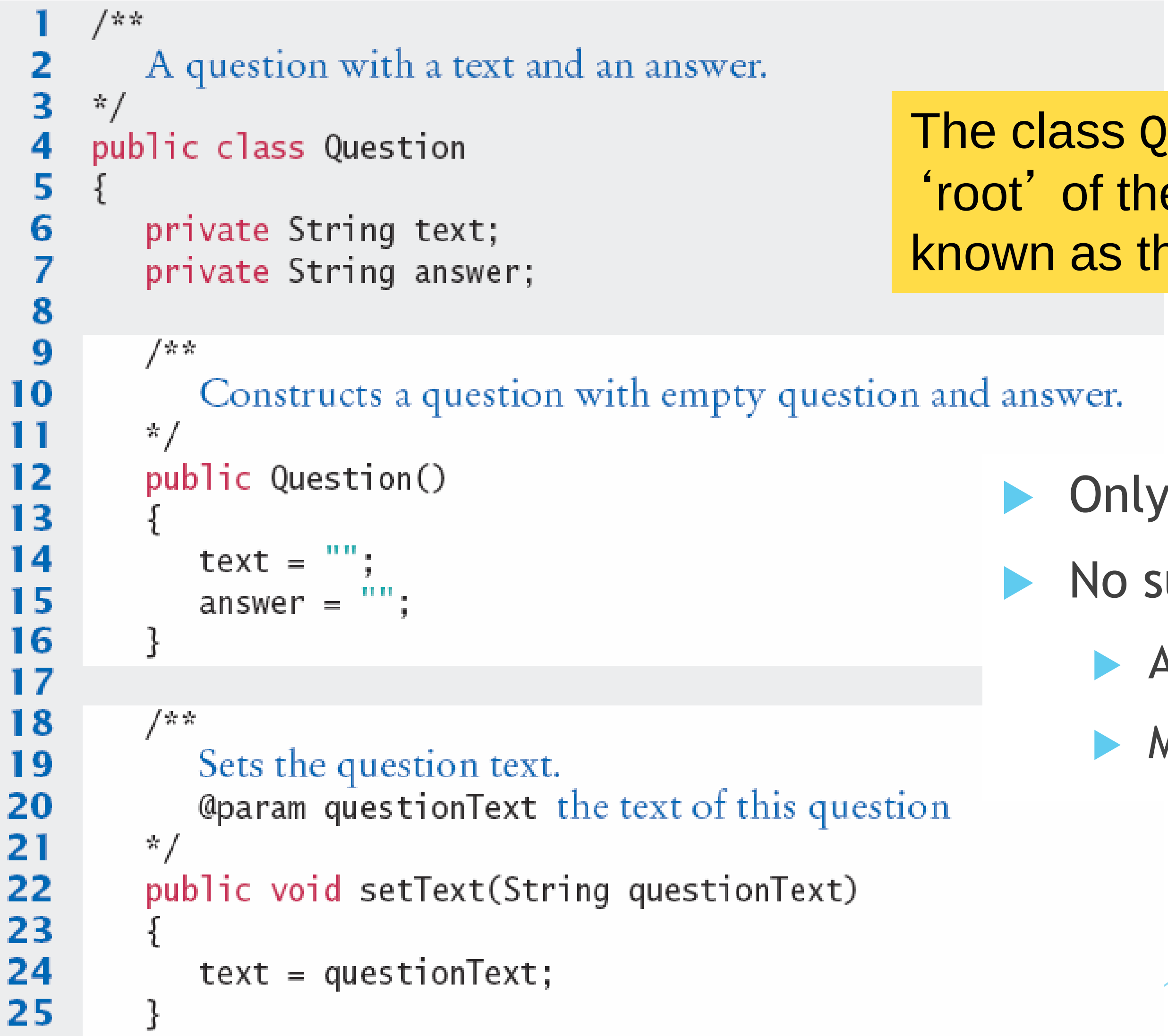
- There are different types of quiz questions:
  1) Fill-in-the-blank
  2) Single answer choice
  3) Multiple answer choice
  4) Numeric answer
  5) Free Response

The ‘root’ of the hierarchy is shown at the top.



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

Question.java (1)

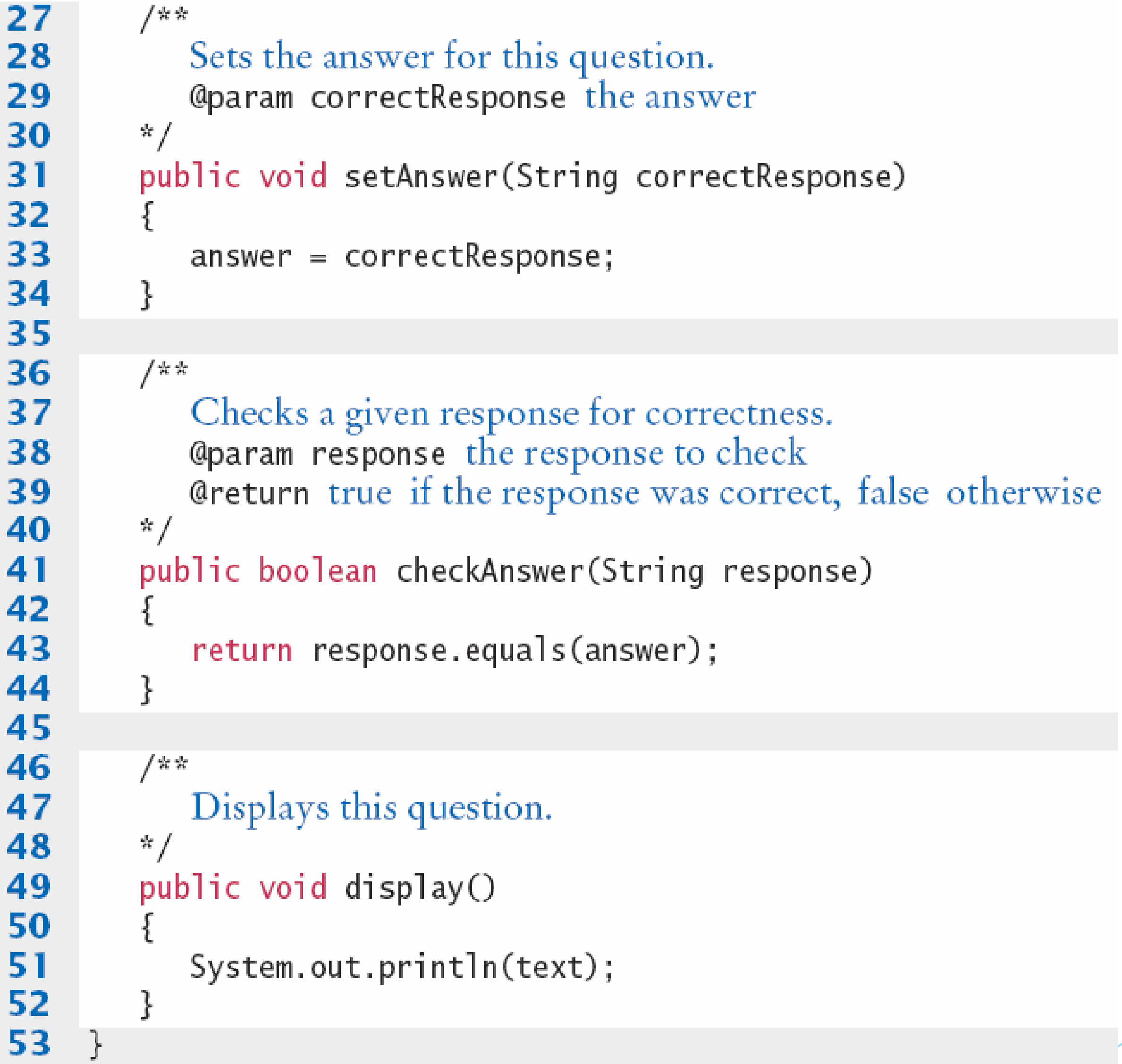


```java
/**
   A question with a text and an answer.
*/
public class Question
{
   private String text;
   private String answer;

   /**
      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;
   }
```

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

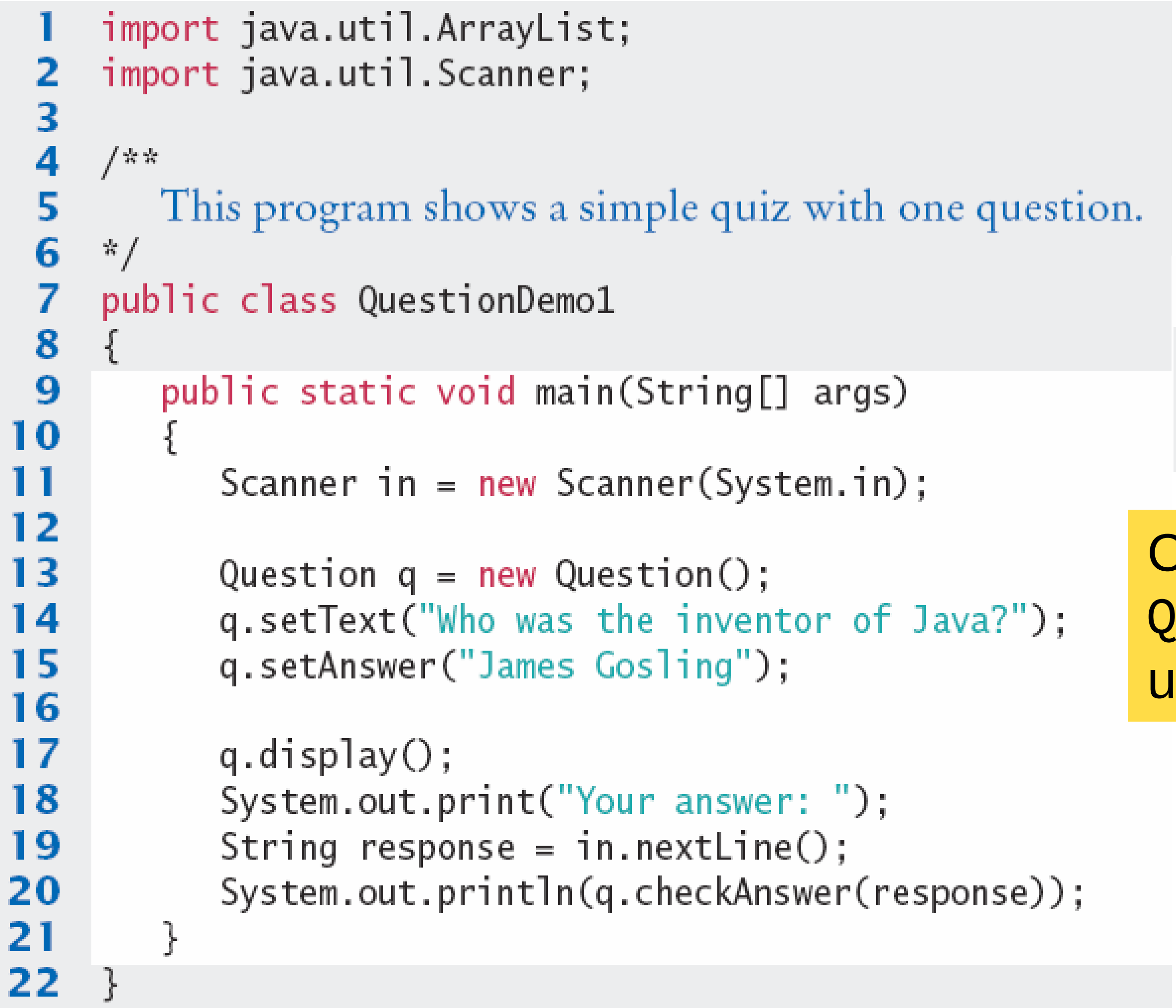
- Only handles Strings
- No support for:
  - Approximate values
  - Multiple answer choice

Question.java (2)

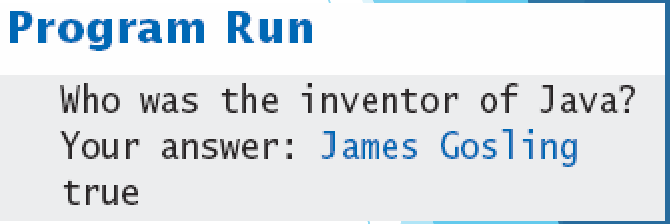


```
    /**
       Sets the answer for this question.
       @param correctResponse the answer
    */
    public void setAnswer(String correctResponse)
    {
       answer = correctResponse;
    }

    /**
       Checks a given response for correctness.
       @param response the response to check
       @return true if the response was correct, false otherwise
    */
    public boolean checkAnswer(String response)
    {
       return response.equals(answer);
    }

    /**
       Displays this question.
    */
    public void display()
    {
       System.out.println(text);
    }
}
```

QuestionDemo1.java



```java
import java.util.ArrayList;
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
```

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

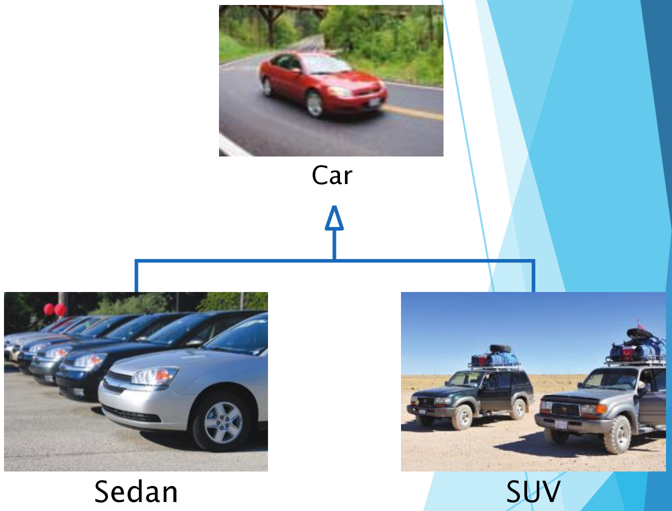
Programming Tip

- 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;
```

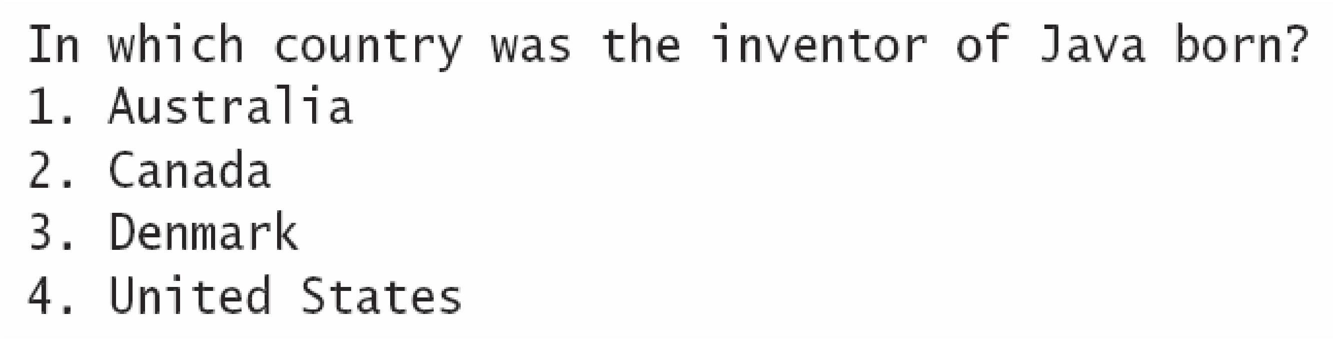
  - If two vehicles behave differently, use inheritance

Be careful not to over-use inheritance



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

Inheriting from the Superclass

- Subclasses inherit from the superclass:
  - All public methods that it does not override
  - All public 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.

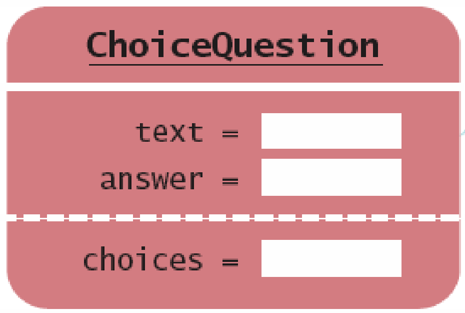
Overriding Superclass Methods

- 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

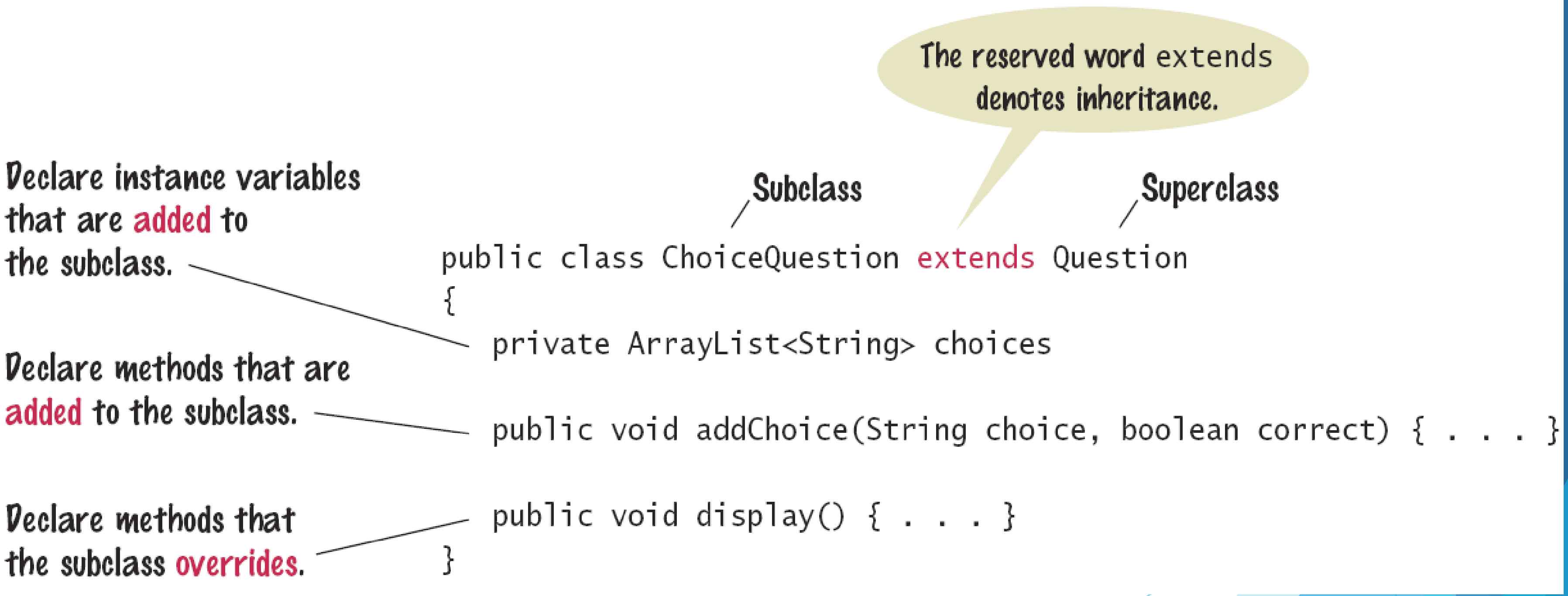
- Use the reserved word `extends` to inherit from `Question`
  - Inherits `text` and `answer` variables
  - Add new instance variable `choices`



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

Subclass Declaration

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



```
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 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

- 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;
   . . .
```

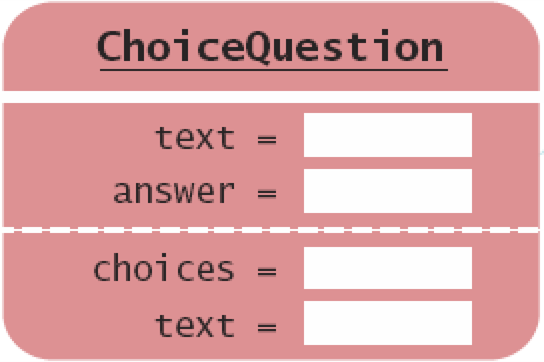
```
public class ChoiceQuestion extends Question
{
   . . .
   text = questionText;    // Complier Error!
```

Common Error

- 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
```

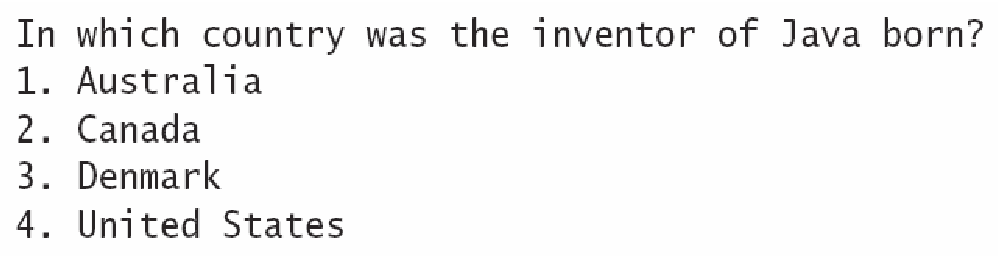
- The constructor sets one text variable
- The display method outputs the other



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



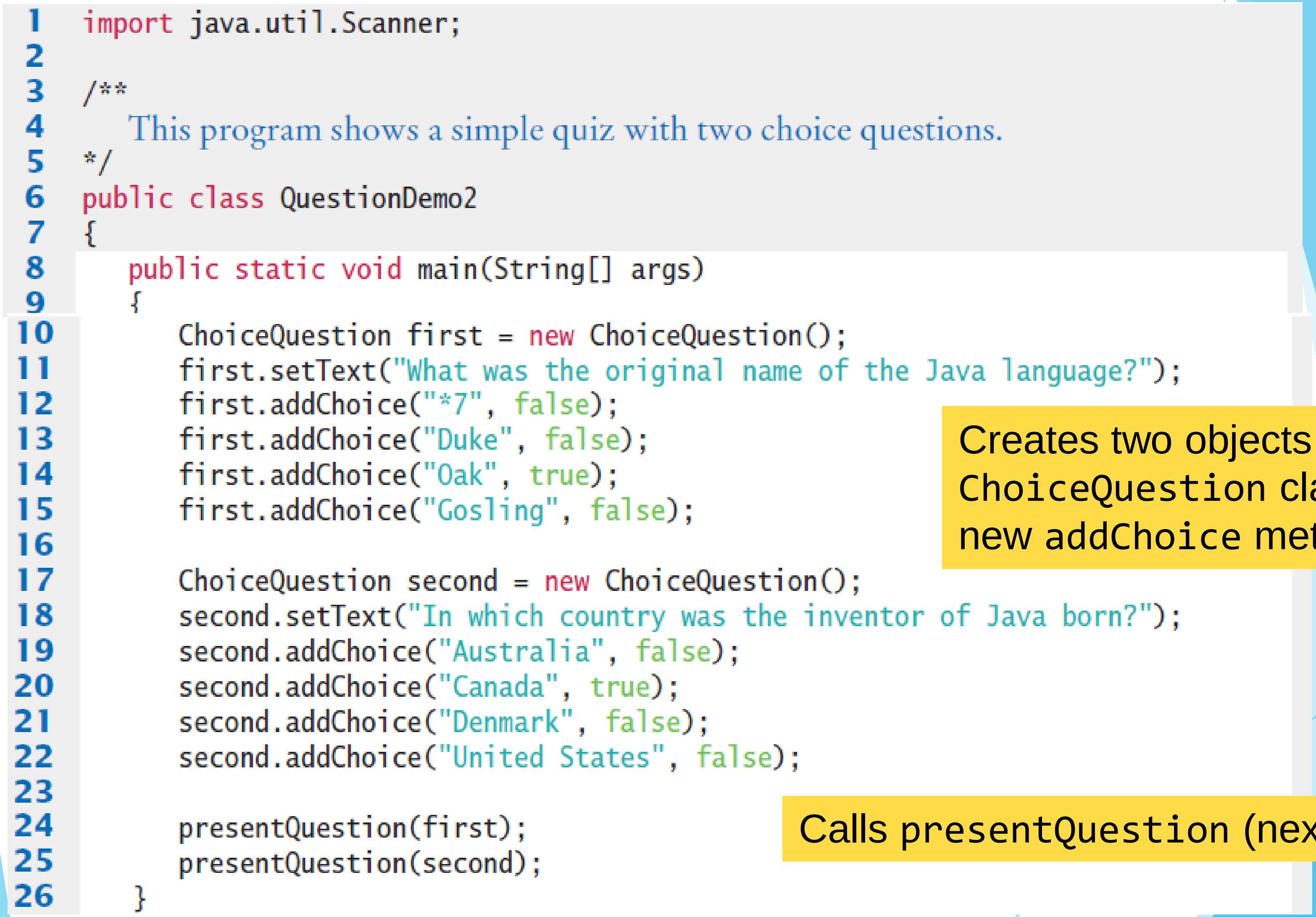
```
In which country was the inventor of Java born?
1. Australia
2. Canada
3. Denmark
4. United States
```

- 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)

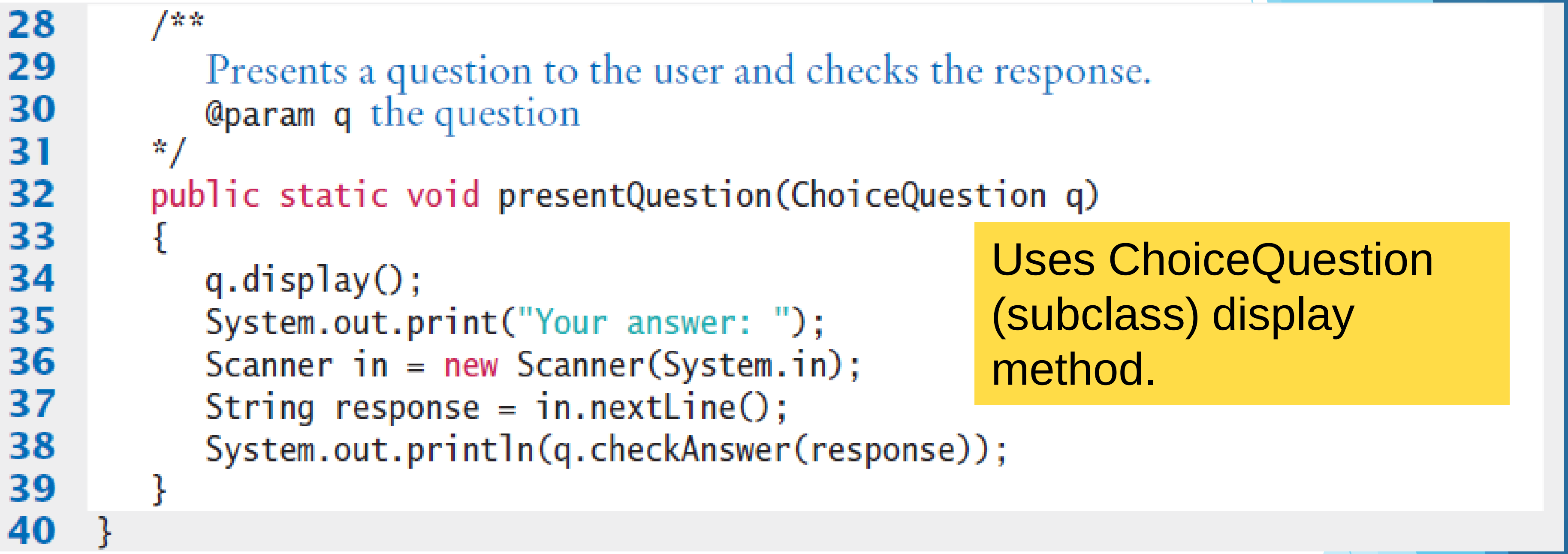


```java
import java.util.Scanner;

/**
   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);
   }
```

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

Calls `presentQuestion` (next page)

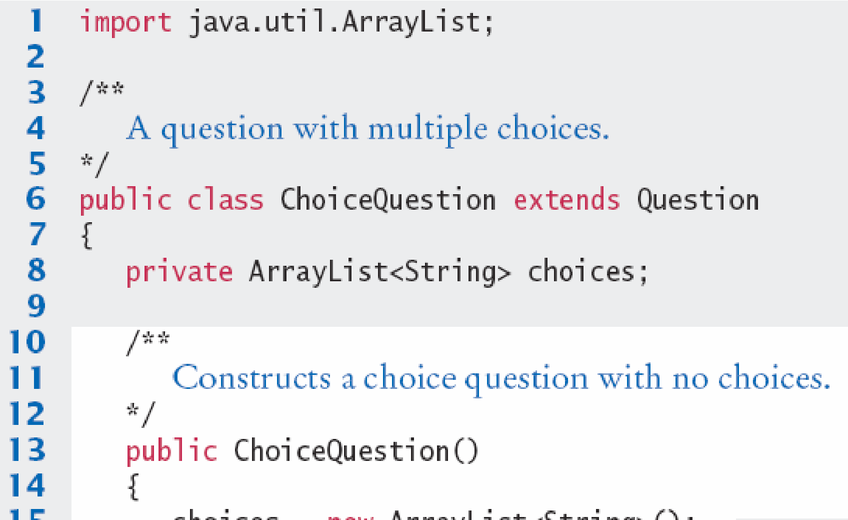
QuestionDemo2.java (2)



```
   /**
      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: ");
      Scanner in = new Scanner(System.in);
      String response = in.nextLine();
      System.out.println(q.checkAnswer(response));
   }
}
```

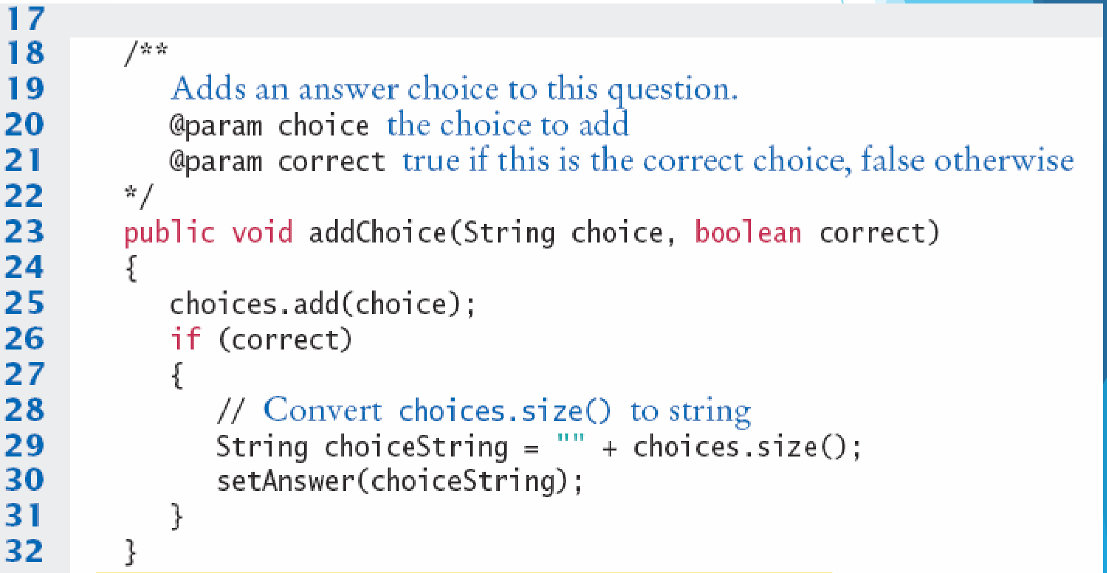
Uses ChoiceQuestion (subclass) display method.

ChoiceQuestion.java (1)



```
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>();
   }
```

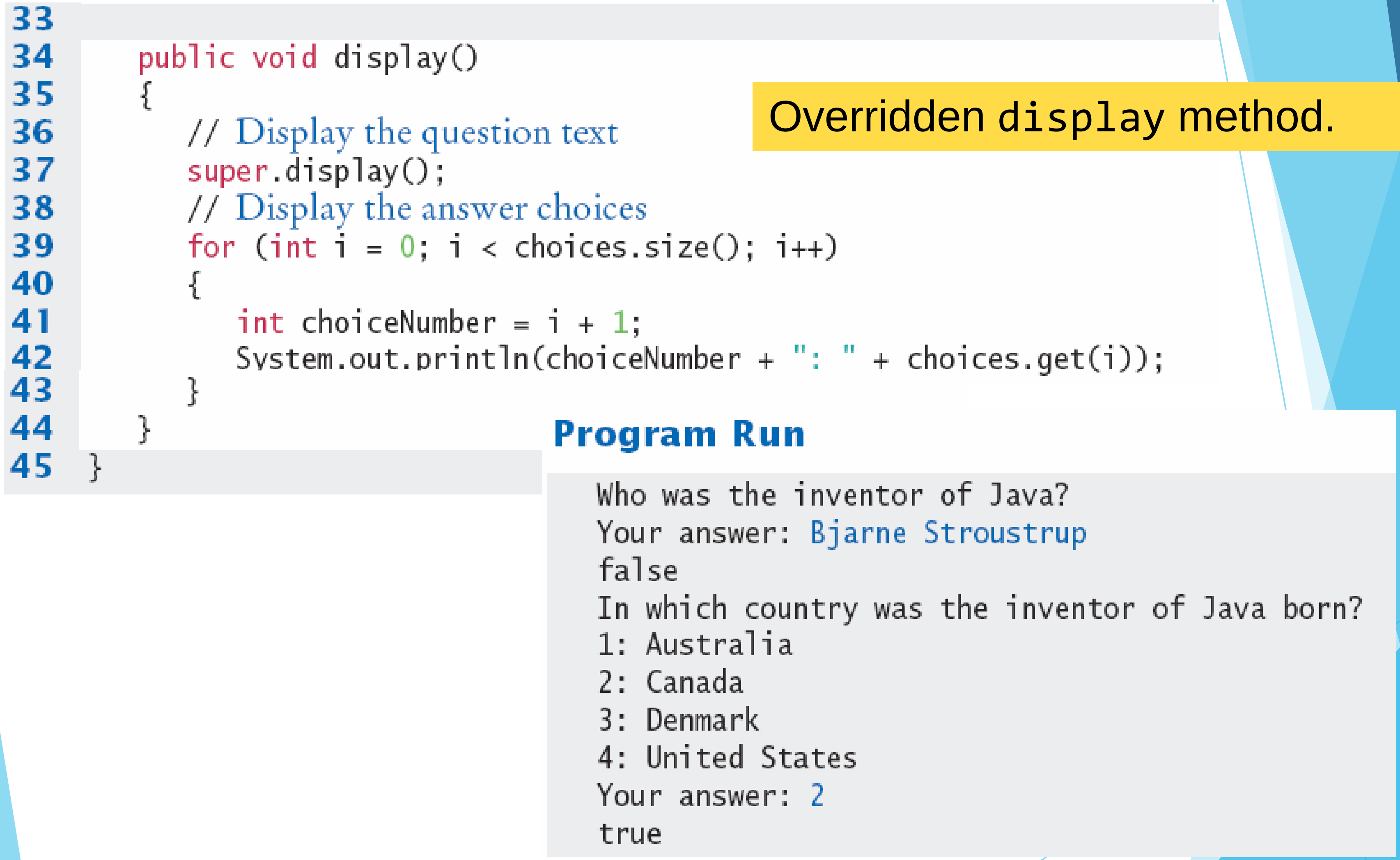
Inherits from `Question` class.



```

   /**
      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
         String choiceString = "" + choices.size();
         setAnswer(choiceString);
      }
   }
```

New `addChoice` method.

ChoiceQuestion.java (2)



```java

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

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

- 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;
   }
}
```

Calling the Superclass Constructor

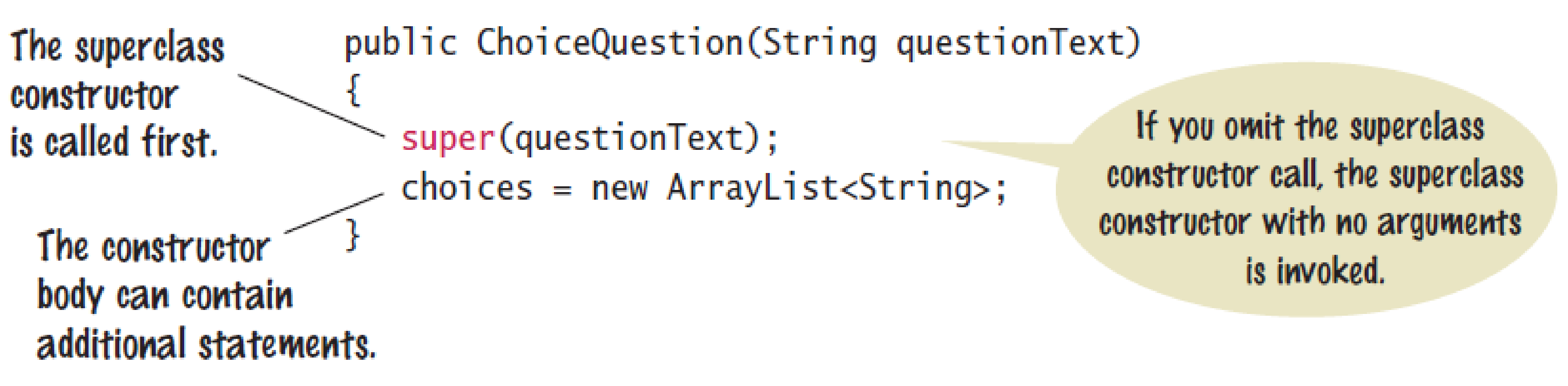
- When a subclass is instantiated, it will call the superclass constructor with no arguments
- If you prefer to call a more specific constructor, you can invoke it by using replacing the superclass name with the reserved word super followed by ():

```
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



```
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.

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:

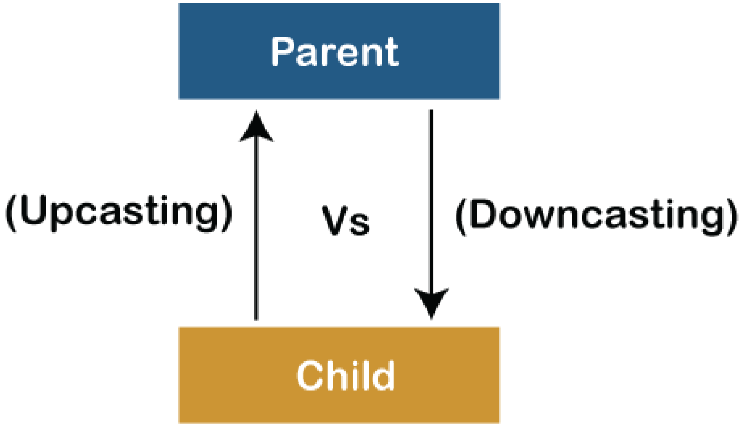
```
public class SecureAccount extends BankAccount
{
   . . .
   public final boolean checkPassword(String password)
   {
      . . .
   }
}
```

Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java:
  - compile-time polymorphism,
  - runtime polymorphism.
- We can perform polymorphism in java by method overloading and method overriding.
- If you overload a static method in Java, it is the example of compile time polymorphism

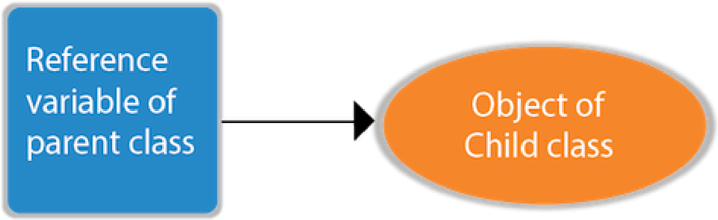
Polymorphism – Upcasting & Downcasting

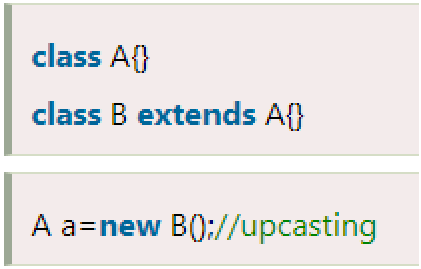
- In Java, the object can also be typecasted like the datatypes.
- **Typecasting** is used to ensure whether variables are correctly processed by a function or not.
- In **Upcasting** and **Downcasting**, we typecast **a child object to a parent object** and **a parent object to a child object** simultaneously.



Upcasting

- **Upcasting** is a type of object typecasting in which a **child object** is typecasted to a **parent class object.**
- we can easily access the variables and methods of the parent class to the child class.
- **Upcasting** is also known as **Generalization** and **Widening.**



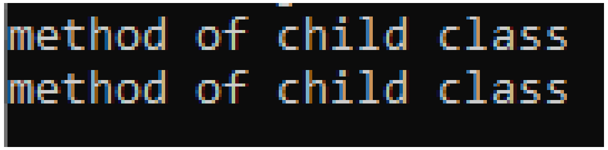


```
class A{}
class B extends A{}
```

```
A a=new B();//upcasting
```

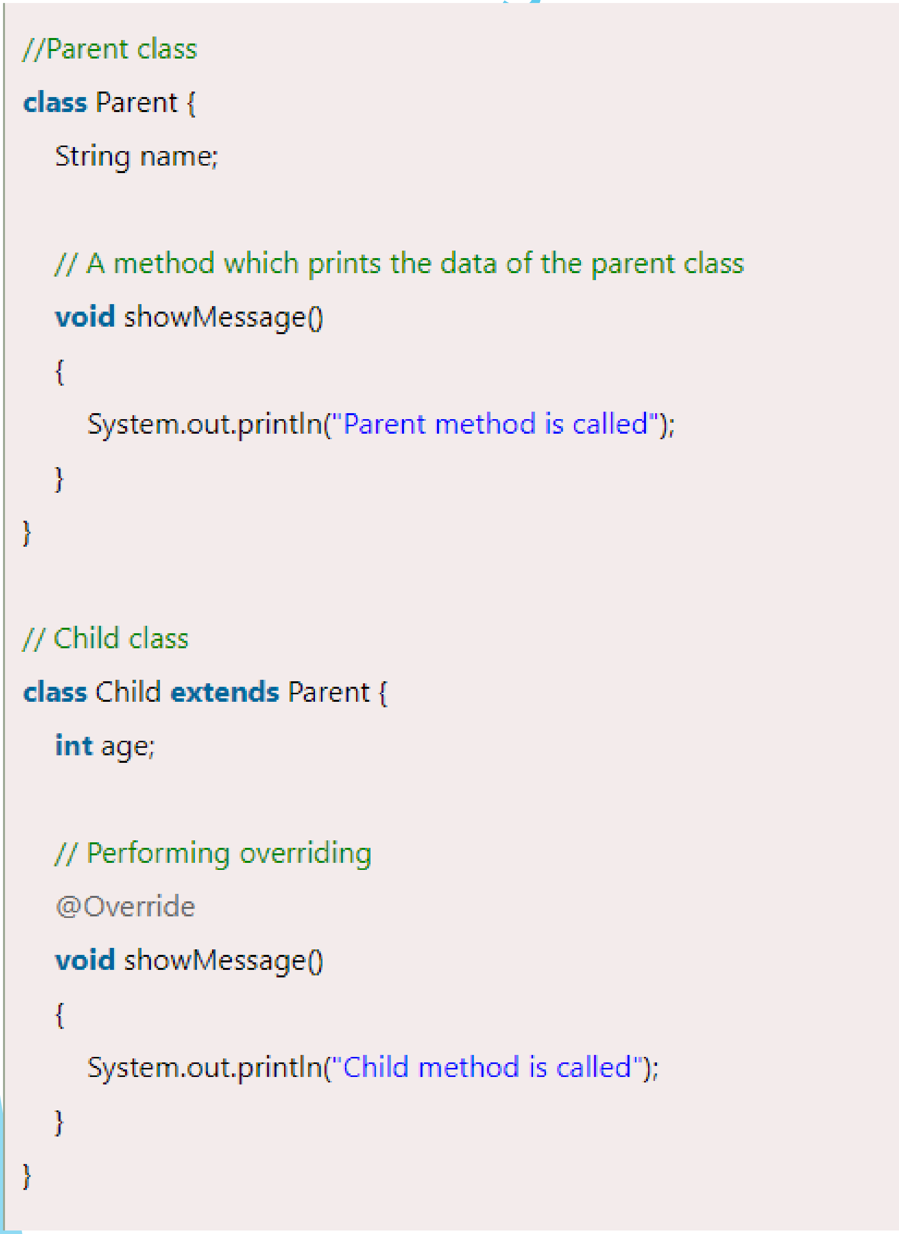
Upcasting

```
class  Parent{
  void PrintData() {
    System.out.println("method of parent class");
  }
}

class Child extends Parent {
  void PrintData() {
    System.out.println("method of child class");
  }
}
class UpcastingExample{
  public static void main(String args[]) {

    Parent obj1 = (Parent) new Child();
    Parent obj2 = (Parent) new Child();
    obj1.PrintData();
    obj2.PrintData();
  }
}
```

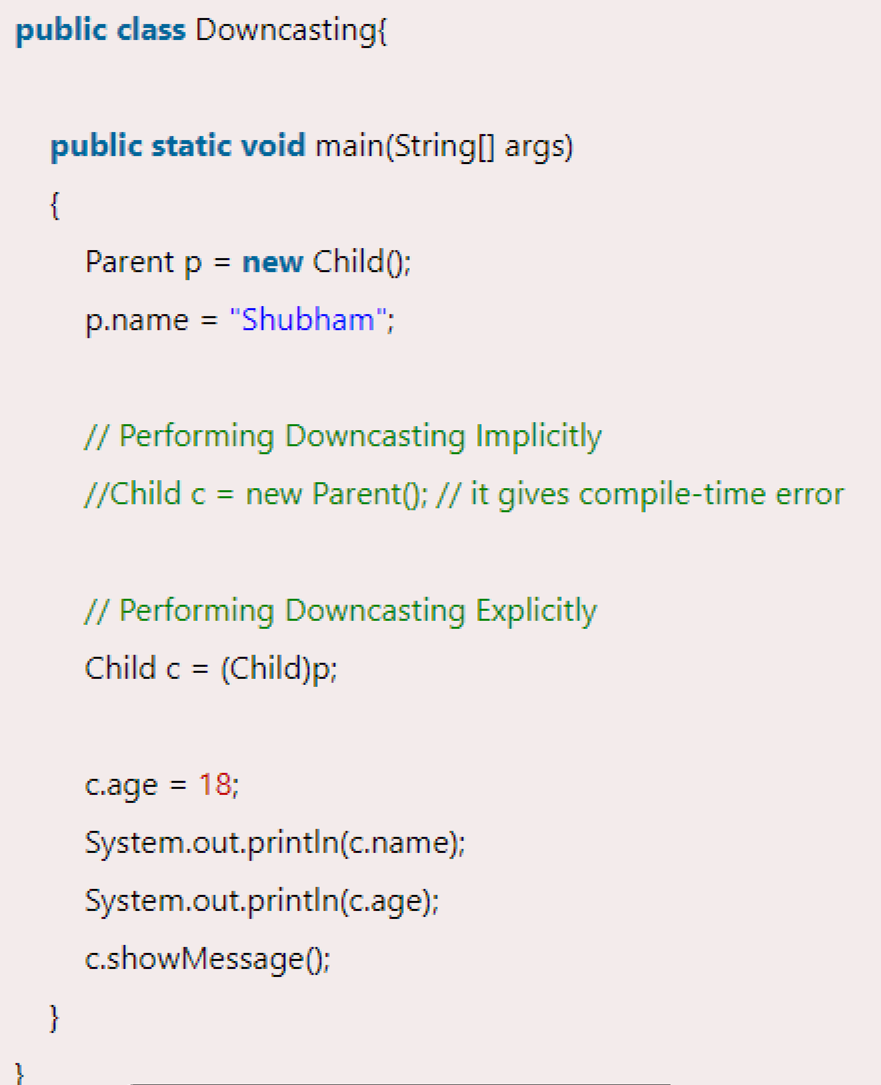


```
method of child class
method of child class
```

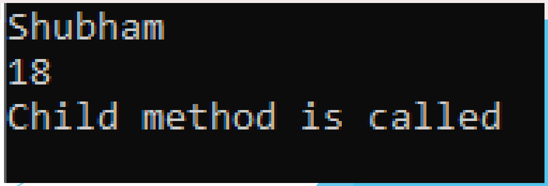
Downcasting



```java
//Parent class
class Parent {
   String name;

   // A method which prints the data of the parent class
   void showMessage()
   {
      System.out.println("Parent method is called");
   }
}

// Child class
class Child extends Parent {
   int age;

   // Performing overriding
   @Override
   void showMessage()
   {
      System.out.println("Child method is called");
   }
}
```

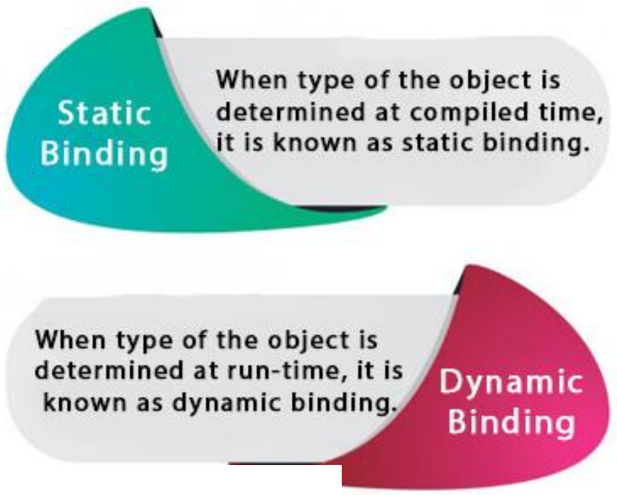


```java
public class Downcasting{

   public static void main(String[] args)
   {
      Parent p = new Child();
      p.name = "Shubham";

      // Performing Downcasting Implicitly
      //Child c = new Parent(); // it gives compile-time error

      // Performing Downcasting Explicitly
      Child c = (Child)p;

      c.age = 18;
      System.out.println(c.name);
      System.out.println(c.age);
      c.showMessage();
   }
}
```

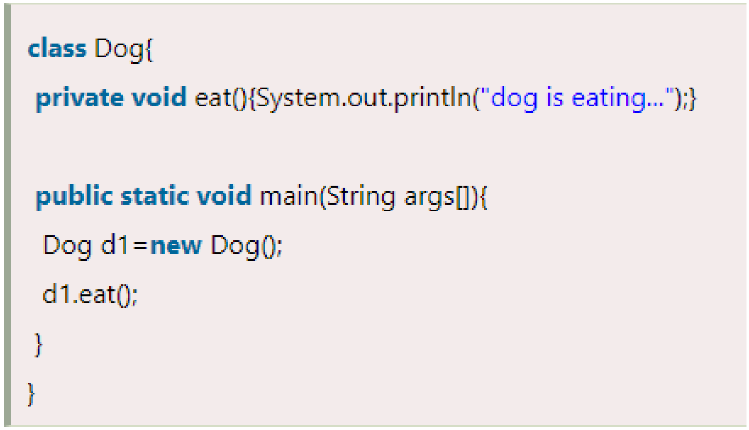


```
Shubham
18
Child method is called
```

Static Binding vs Dynamic Binding

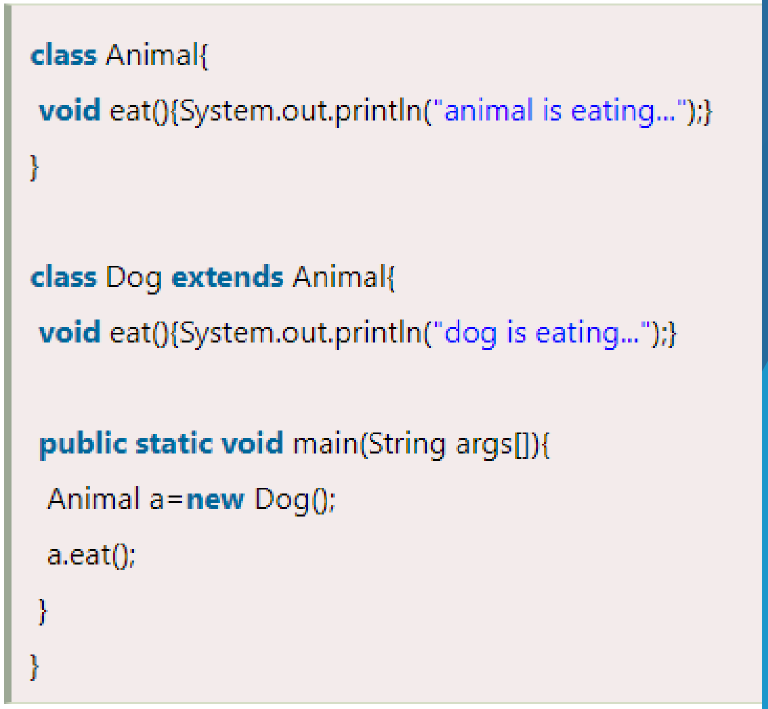

Static Binding
When type of the object is determined at compiled time, it is known as static binding.
When type of the object is determined at run-time, it is known as dynamic binding.
Dynamic Binding

Example of static binding



```
class Dog{
 private void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

Example of dynamic binding

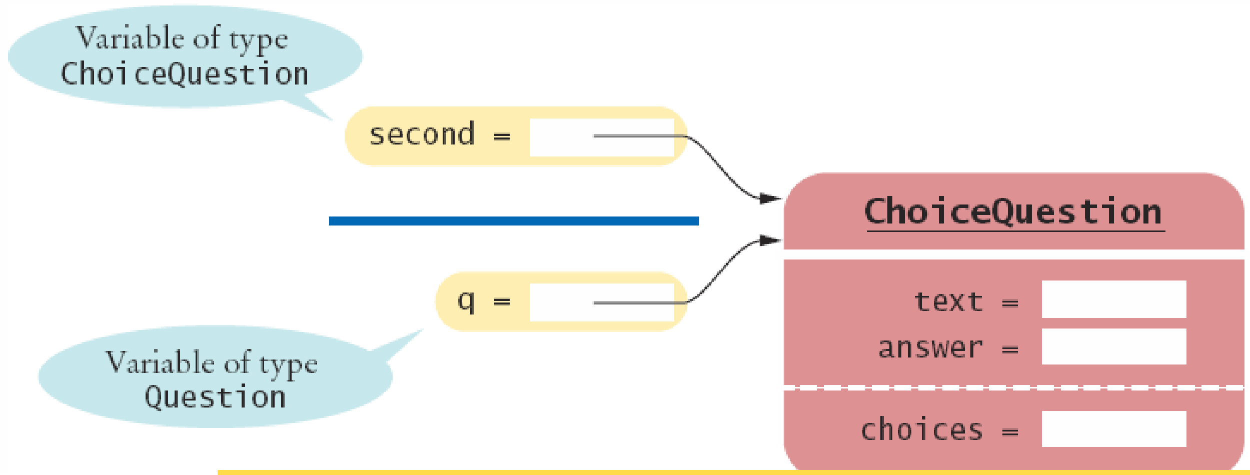


```
class Animal{
 void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

Polymorphism

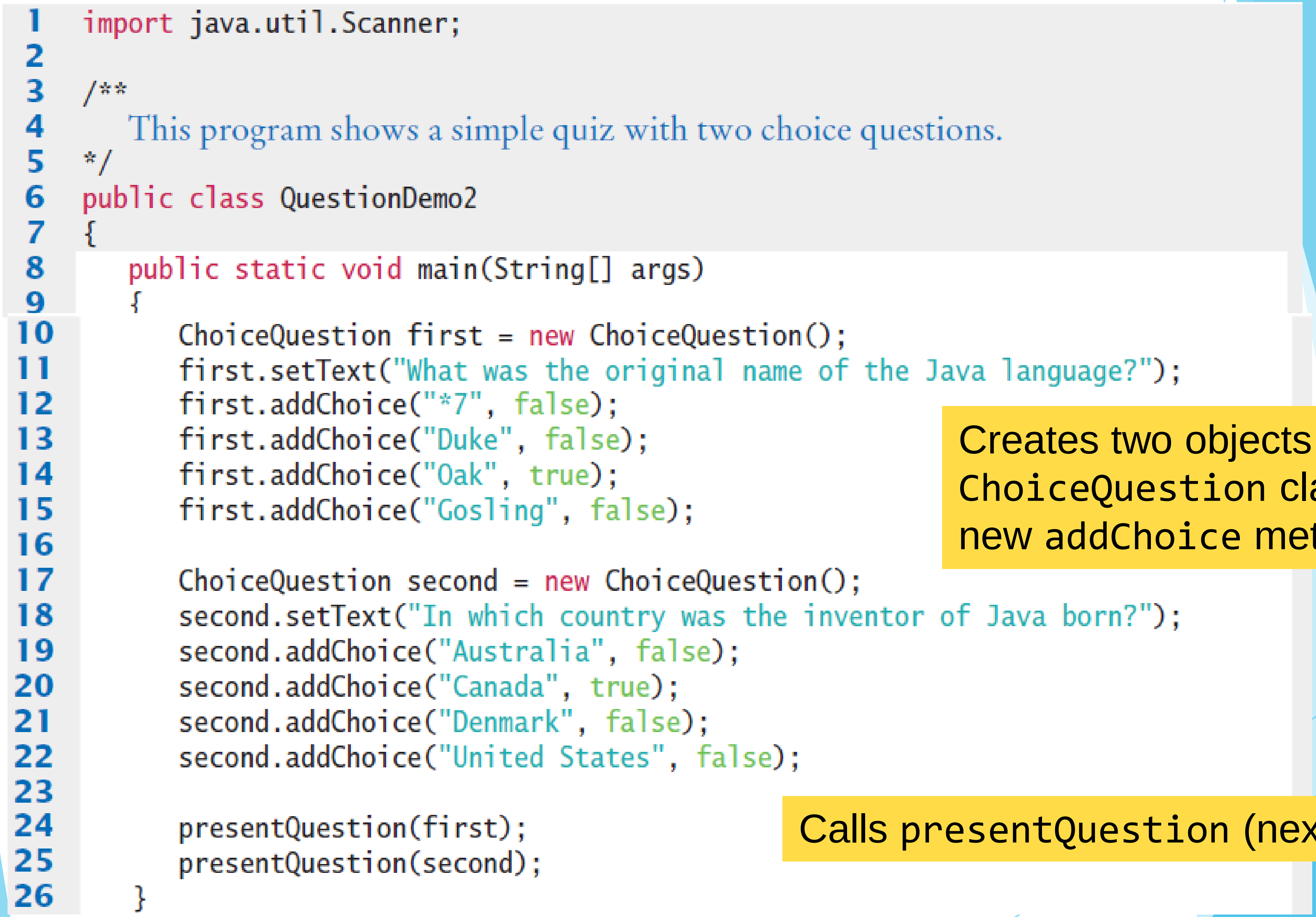
- QuestionDemo2 passed two ChoiceQuestion objects to the presentQuestion method
  - Can we write a presentQuestion method that displays both Question and ChoiceQuestion types?
  - **How would that work?**

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



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

QuestionDemo2.java (1)

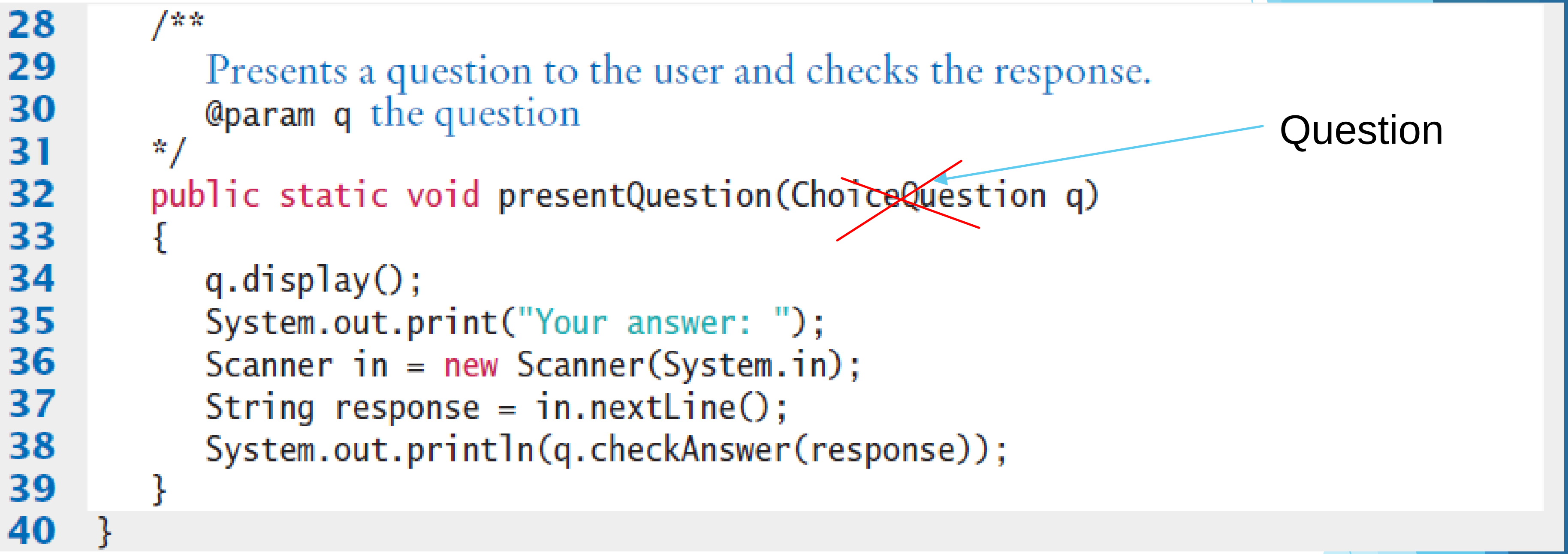


```java
import java.util.Scanner;

/**
   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);
   }
```

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

Calls presentQuestion (next page)

QuestionDemo2.java (2)



```
   /**
      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: ");
      Scanner in = new Scanner(System.in);
      String response = in.nextLine();
      System.out.println(q.checkAnswer(response));
   }
}
```

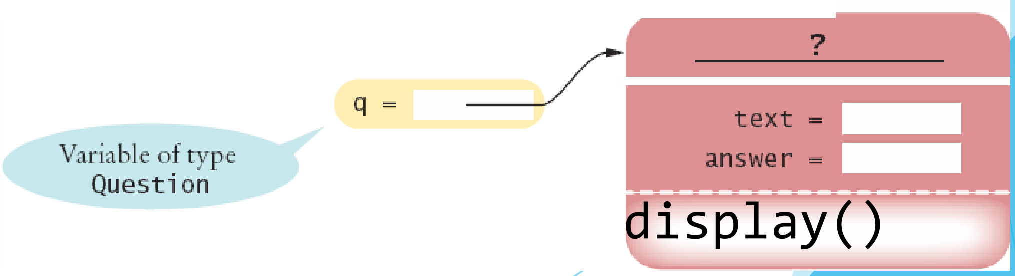
Question

Which display method was called?

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

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

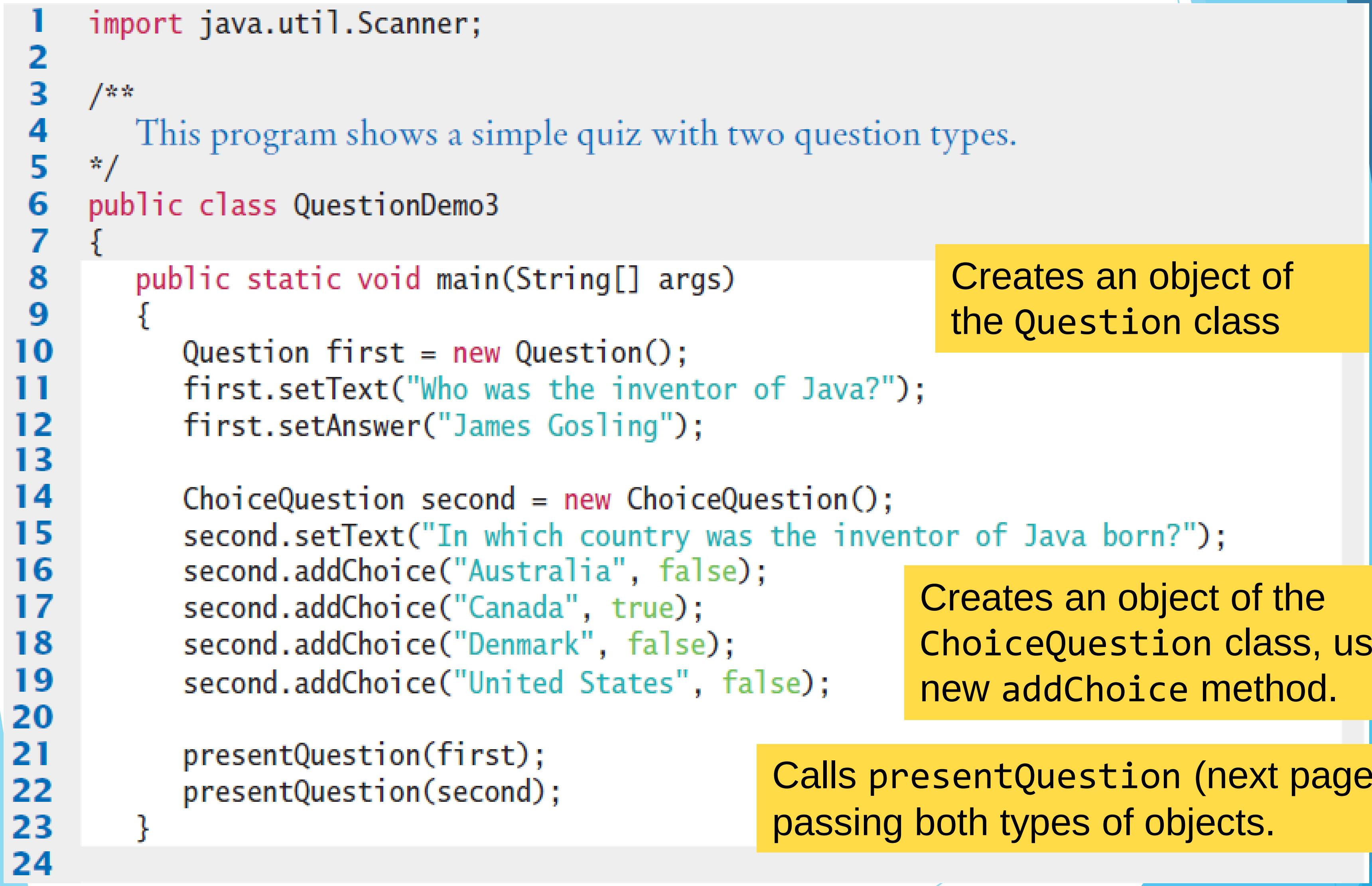
- The variable q does not know the type of object to which it refers:
  - If passed an object of the Question class:
    - Question display
  - If passed an object of the ChoiceQuestion class:
    - ChoiceQuestion display



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)



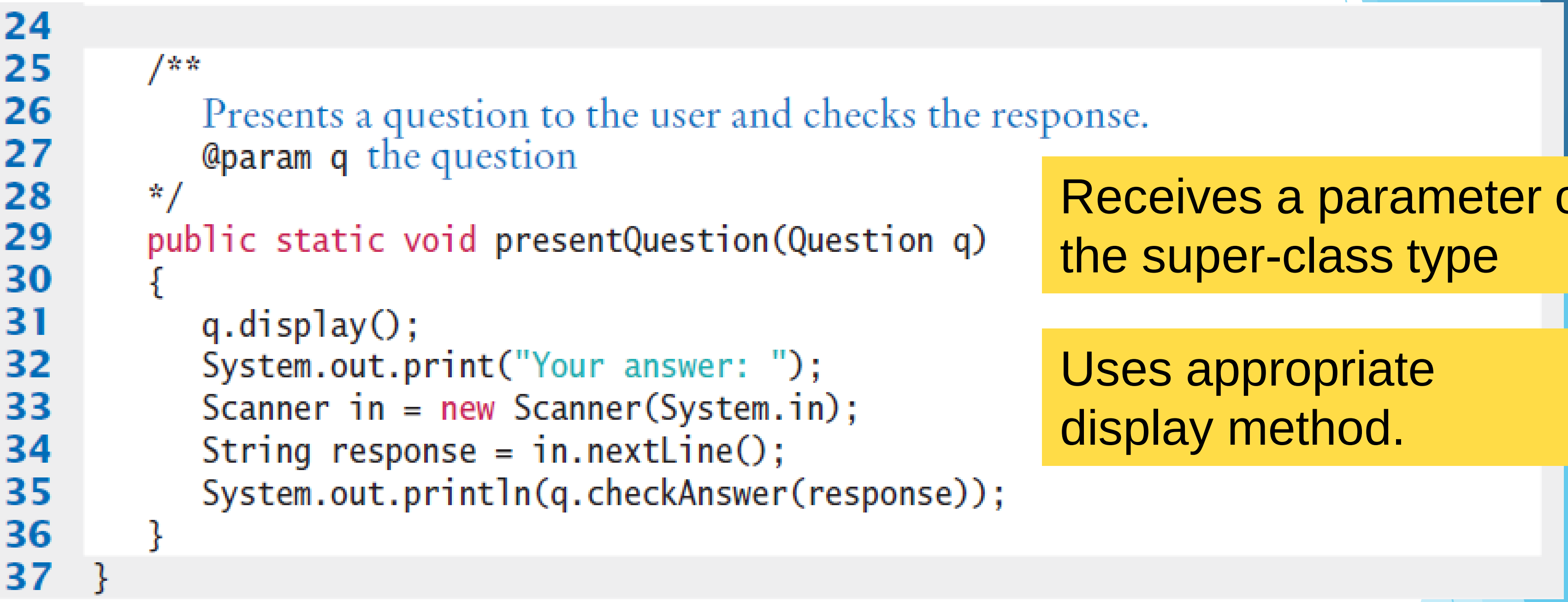
```
import java.util.Scanner;

/**
   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);
   }

```

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)



```
    /**
       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));
    }
}
```

Receives a parameter of the super-class type

Uses appropriate display method.

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 and the Implicit Parameter

- 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));
   }
```

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.

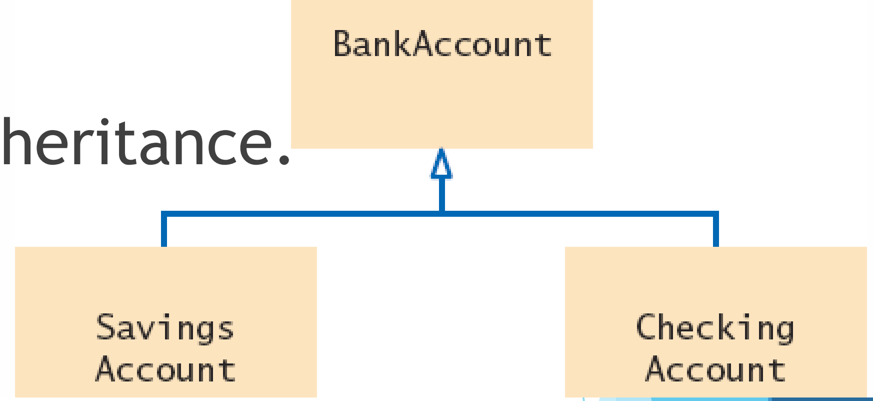
Steps to Using Inheritance

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

```
SavingsAccount
CheckingAccount
```

2) Organize the classes into an inheritance. hierarchy

Base on superclass `BankAccount`

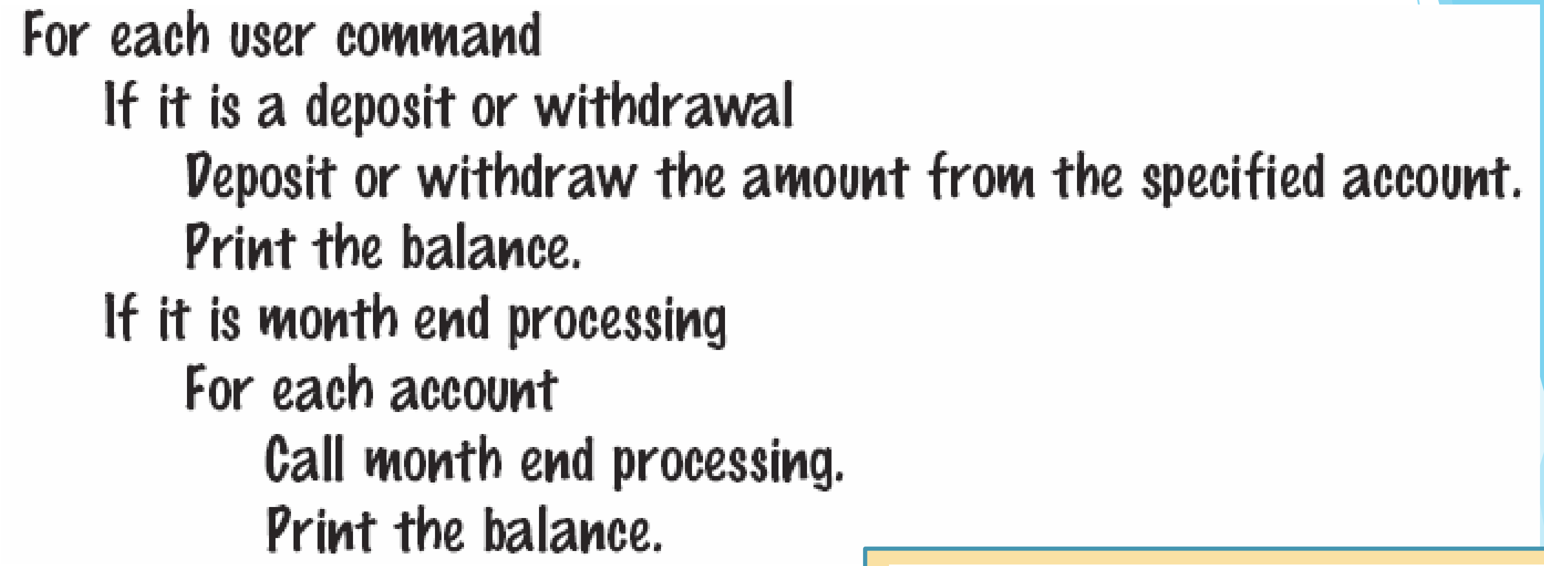


3) Determine the common responsibilities.

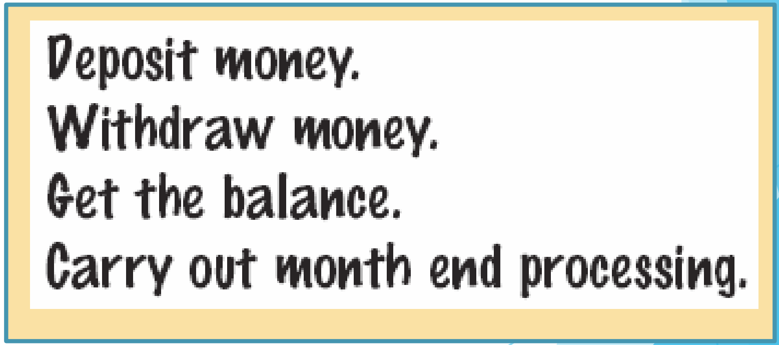
a. Write *Pseudocode* for each task

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.

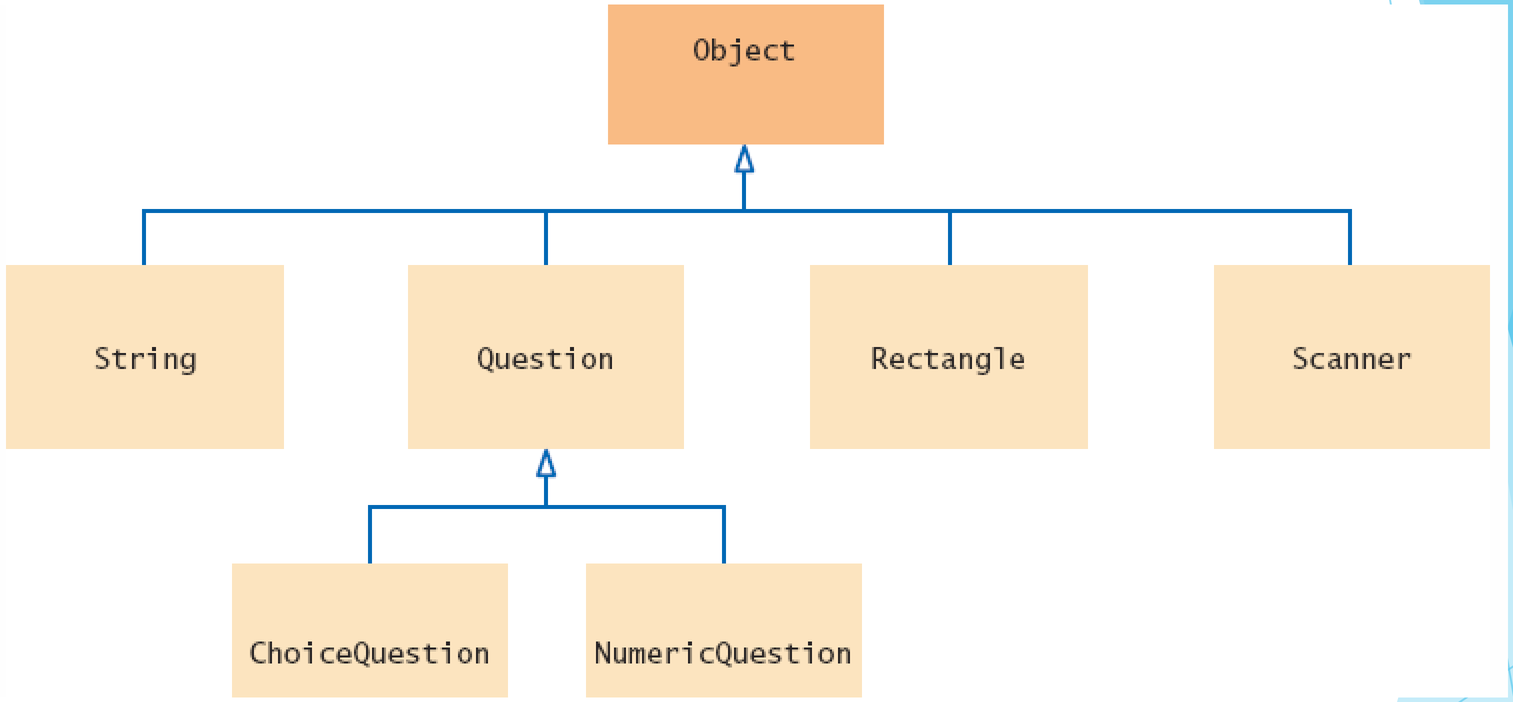
- List the instance variables for each class. Place instance variables that are common to all classes in the base of the hierarchy.

7) Implement constructors and methods.

8) Construct objects of different subclasses and process them.

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.

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

- We want to know what is inside the object

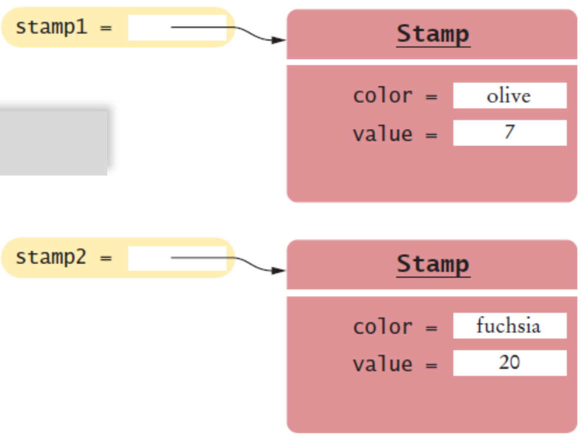
```
public class BankAccount
{
  public String toString()
  {
    // returns "BankAccount[balance=5000]"
    return "BankAccount[balance=" + balance + "]";
  }
}
```

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

Overriding the equals method

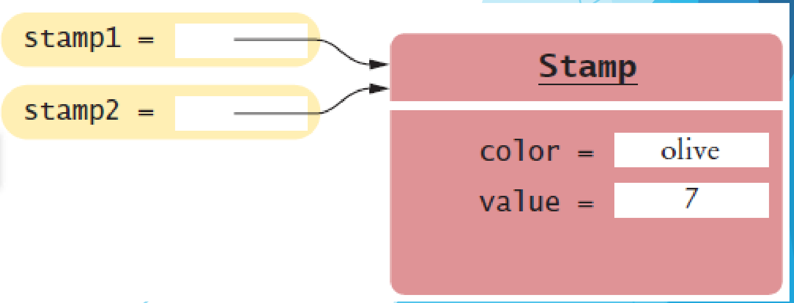
- In addition to the toString method, the Object class equals method checks whether two objects have the same contents:

```
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

- It is legal to 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;
```

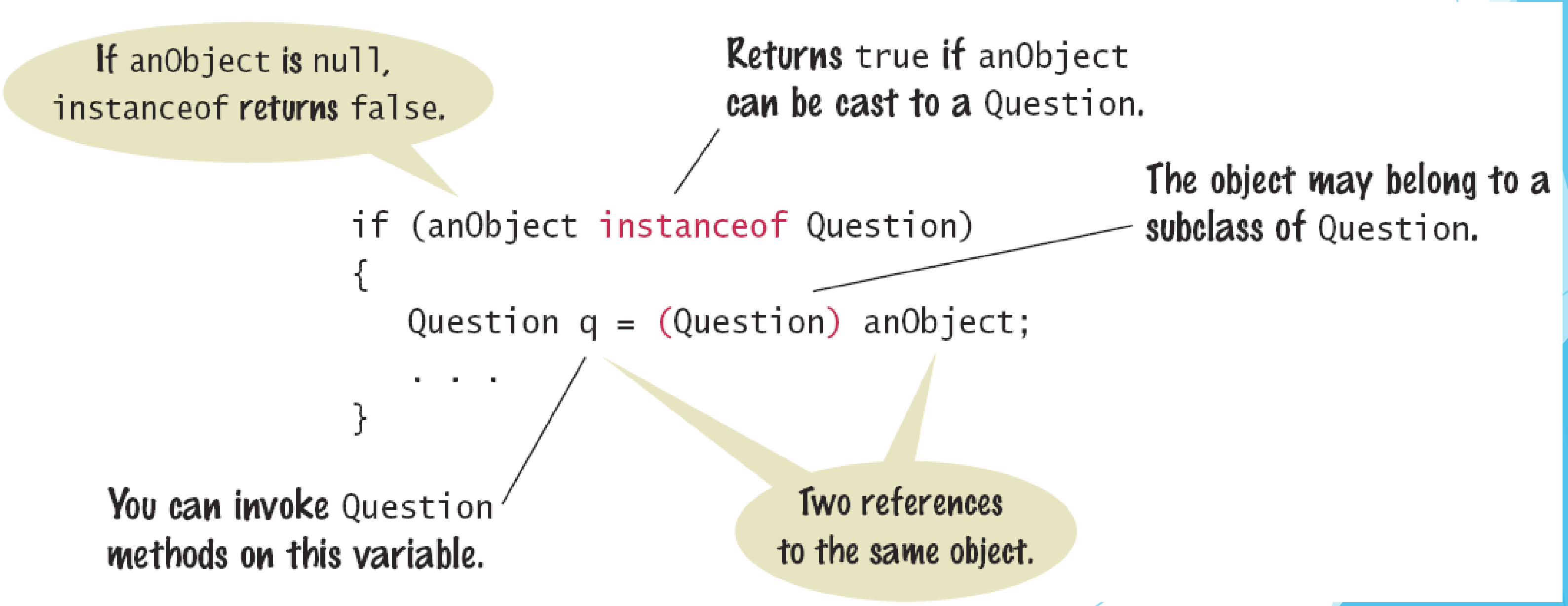
- To make sure it is an object of the `Question` type, you can test it with the `instanceof` operator:

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

`instanceof` returns a boolean

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



If anObject is null, instanceof returns false.

Returns true if anObject can be cast to a Question.

The object may belong to a subclass of Question.

```
if (anObject instanceof Question)
{
   Question q = (Question) anObject;
   . . .
}
```

You can invoke Question methods on this variable.

Two references to the same object.

What is the problem?

```
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
}
```

- Don't Use Type Tests
  - 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

Abstraction

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- There are two ways to achieve abstraction in java
  - Abstract class (0 to 100%)
  - Interface (100%)

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

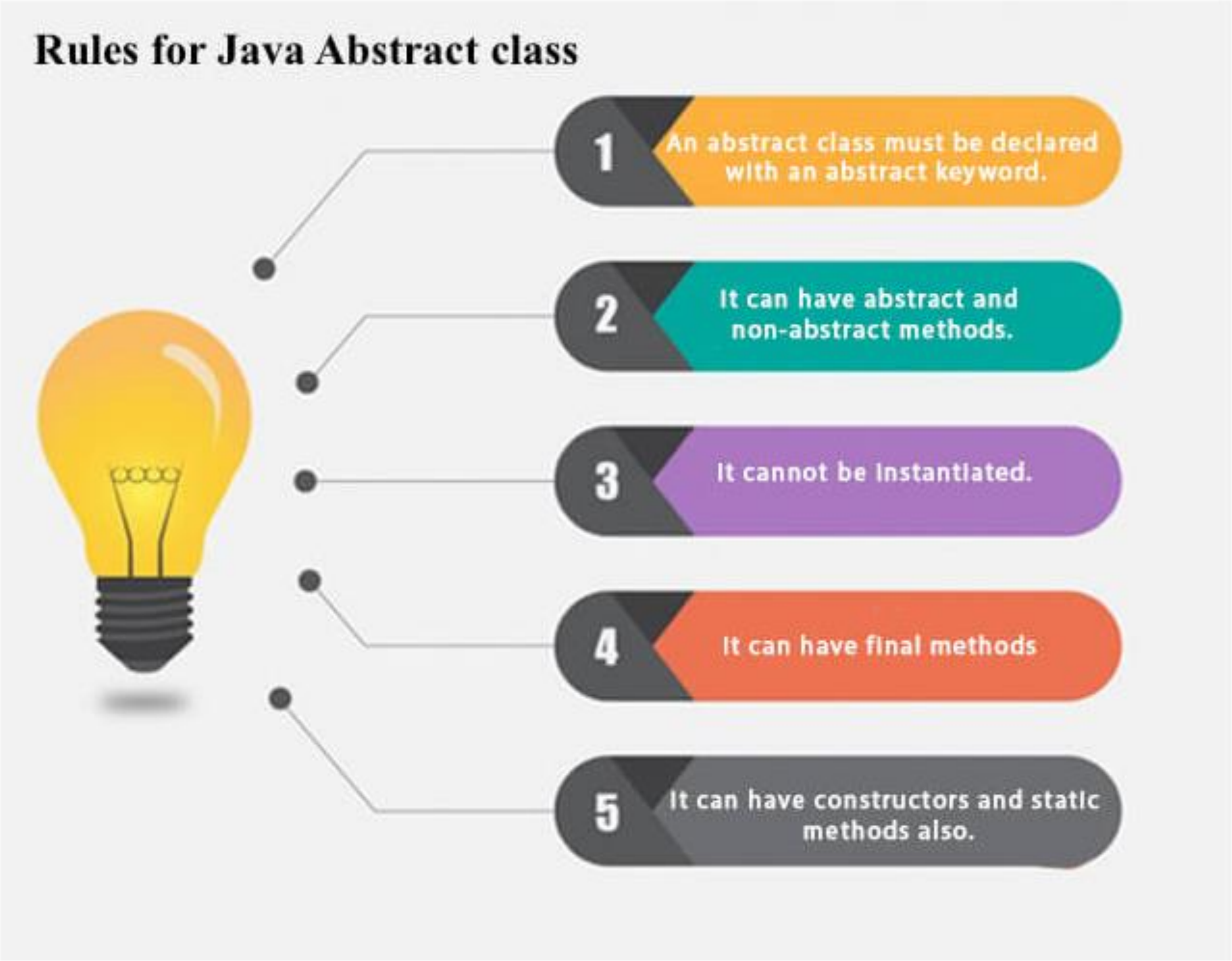
- A class that can be instantiated is called concrete class
- You cannot instantiate an object that has abstract methods
  - But you can declare an object reference whose type is an abstract class.
  - The actual object to which it refers must be an instance of a concrete subclass

```
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.

Points to Remember



Rules for Java Abstract class

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have final methods
5. It can have constructors and static methods also.

Interface

- 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:

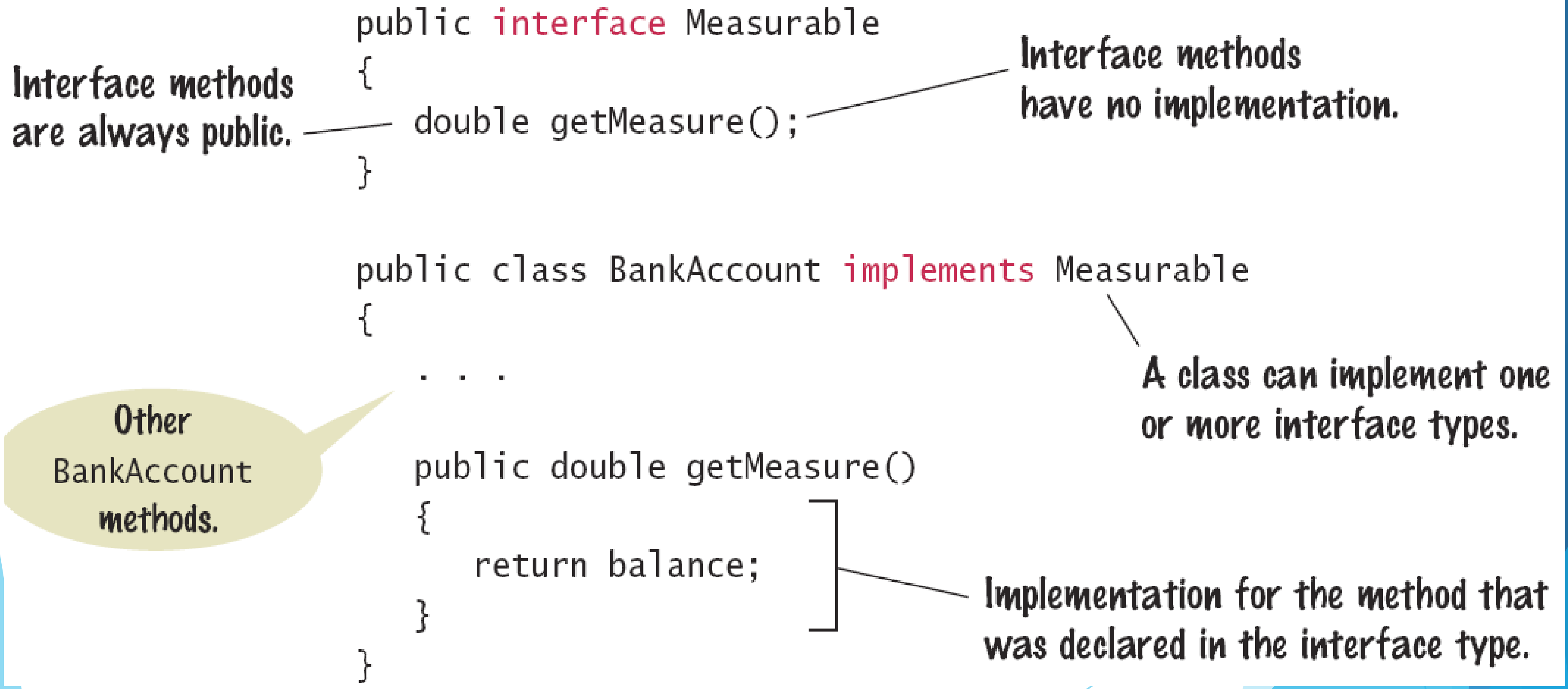
      They have a name, parameters, and a return type, but they don’t have an implementation
    - All methods in an interface type are automatically public
    - An interface type cannot have instance variables
    - An interface type cannot have static methods

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

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

Interface Types

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



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

public class BankAccount implements Measurable
{
   . . .

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

Interface methods are always public.

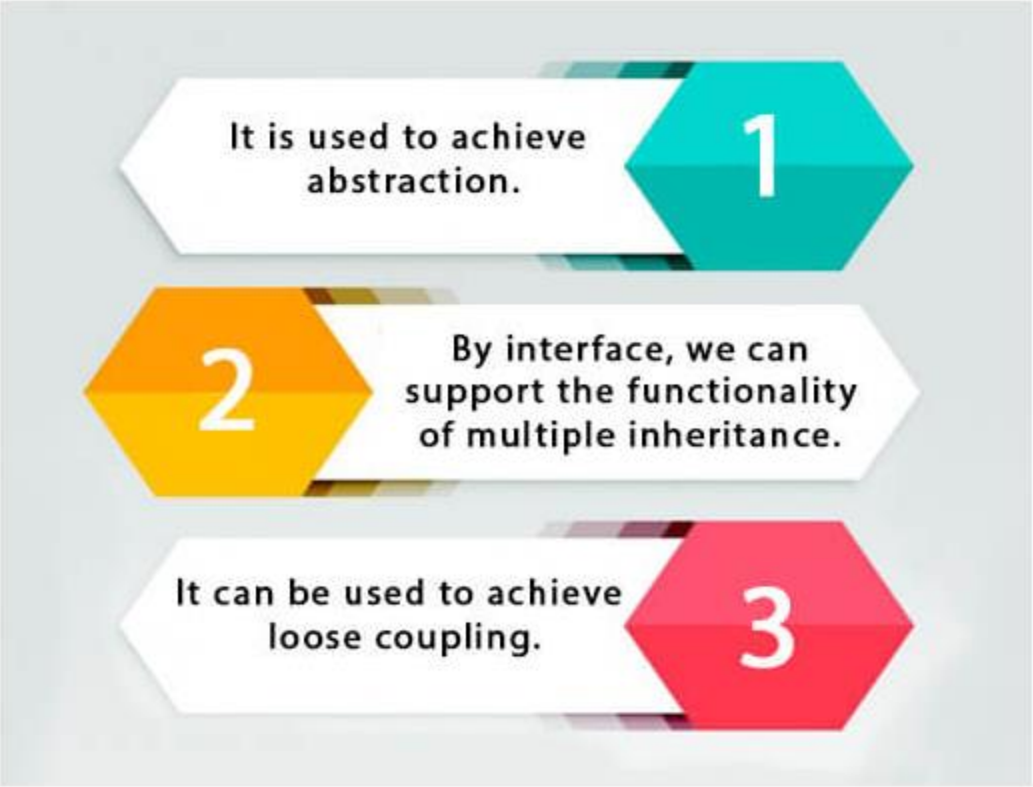
Interface methods have no implementation.

A class can implement one or more interface types.

Other BankAccount methods.

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

Why?



1. It is used to achieve abstraction.
2. By interface, we can support the functionality of multiple inheritance.
3. It can be used to achieve loose coupling.

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

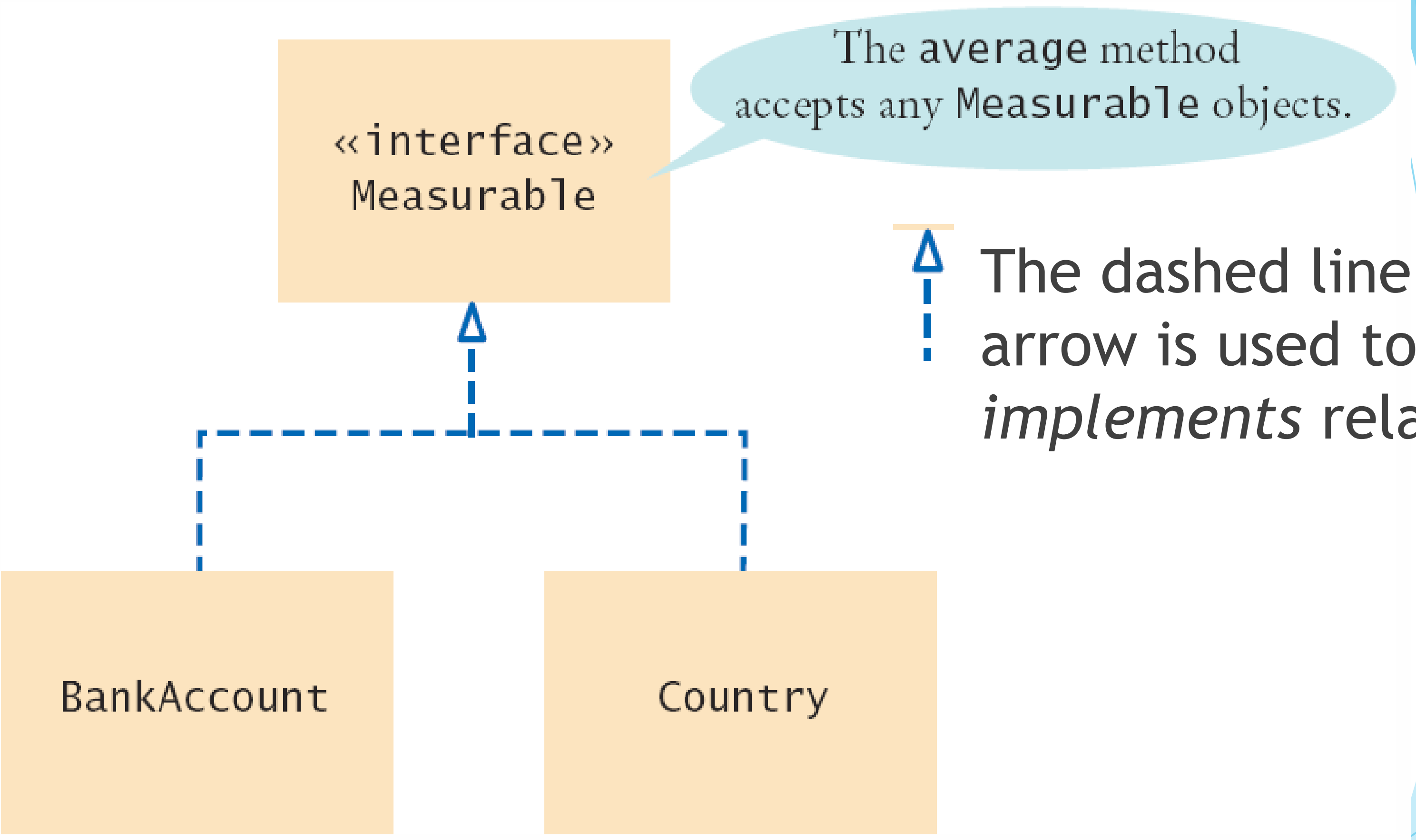
```
public class BankAccount implements Measurable
{
   public double getMeasure()
   {
      return balance;
   }
   . . .
}
```

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`

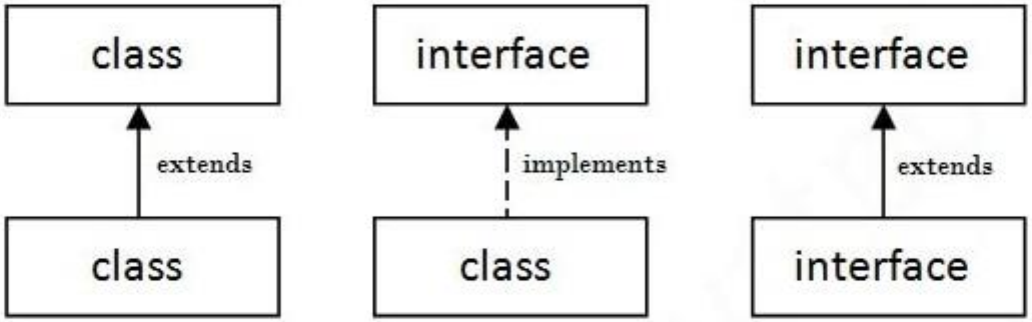
An Implementation Diagram



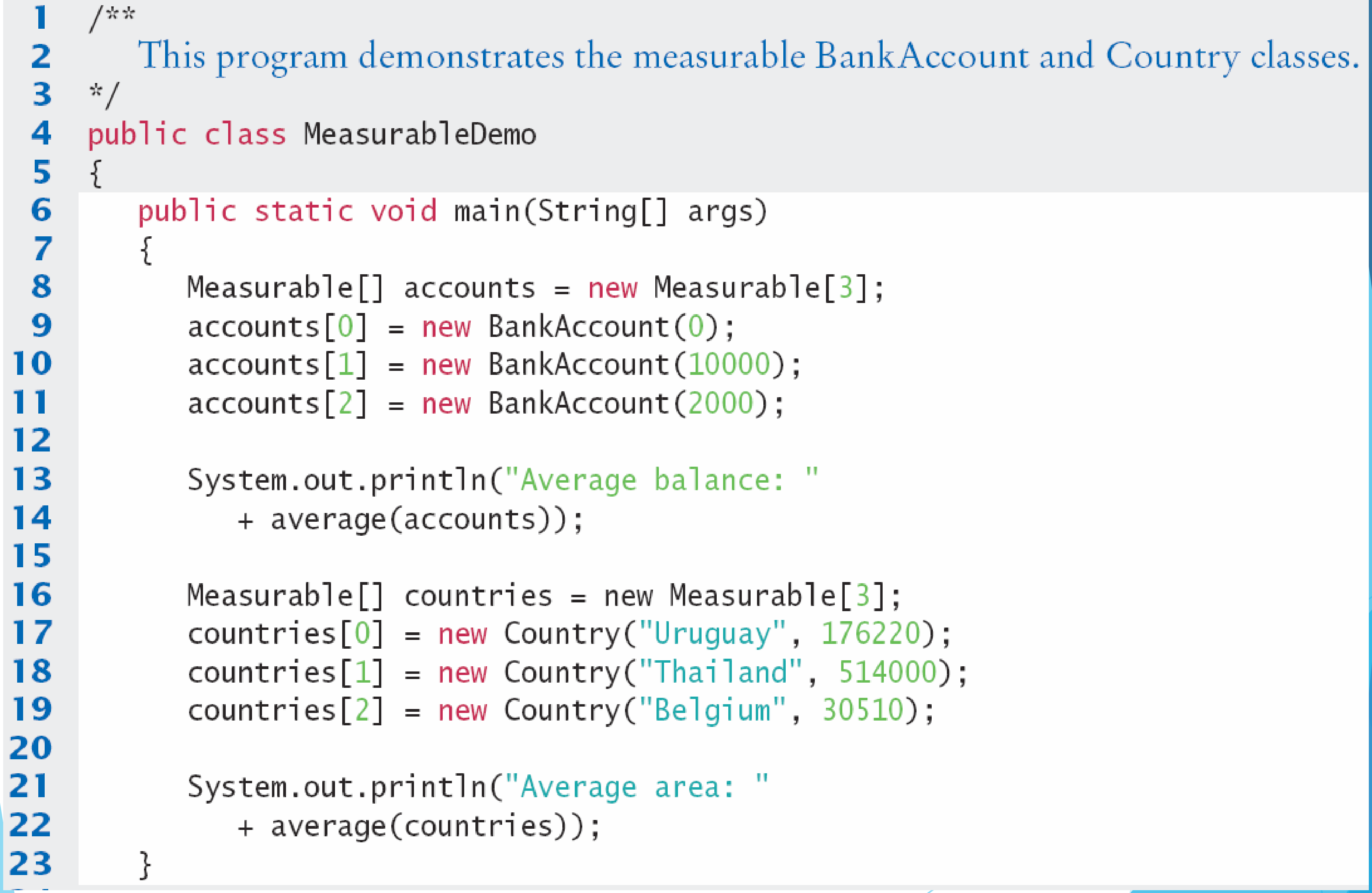
The dashed line with an arrow is used to denote *implements* relationships

The Relationship between interfaces and classes

- A class extends another class.
- An interface extends another interface.
- But a **class implements an interface**.

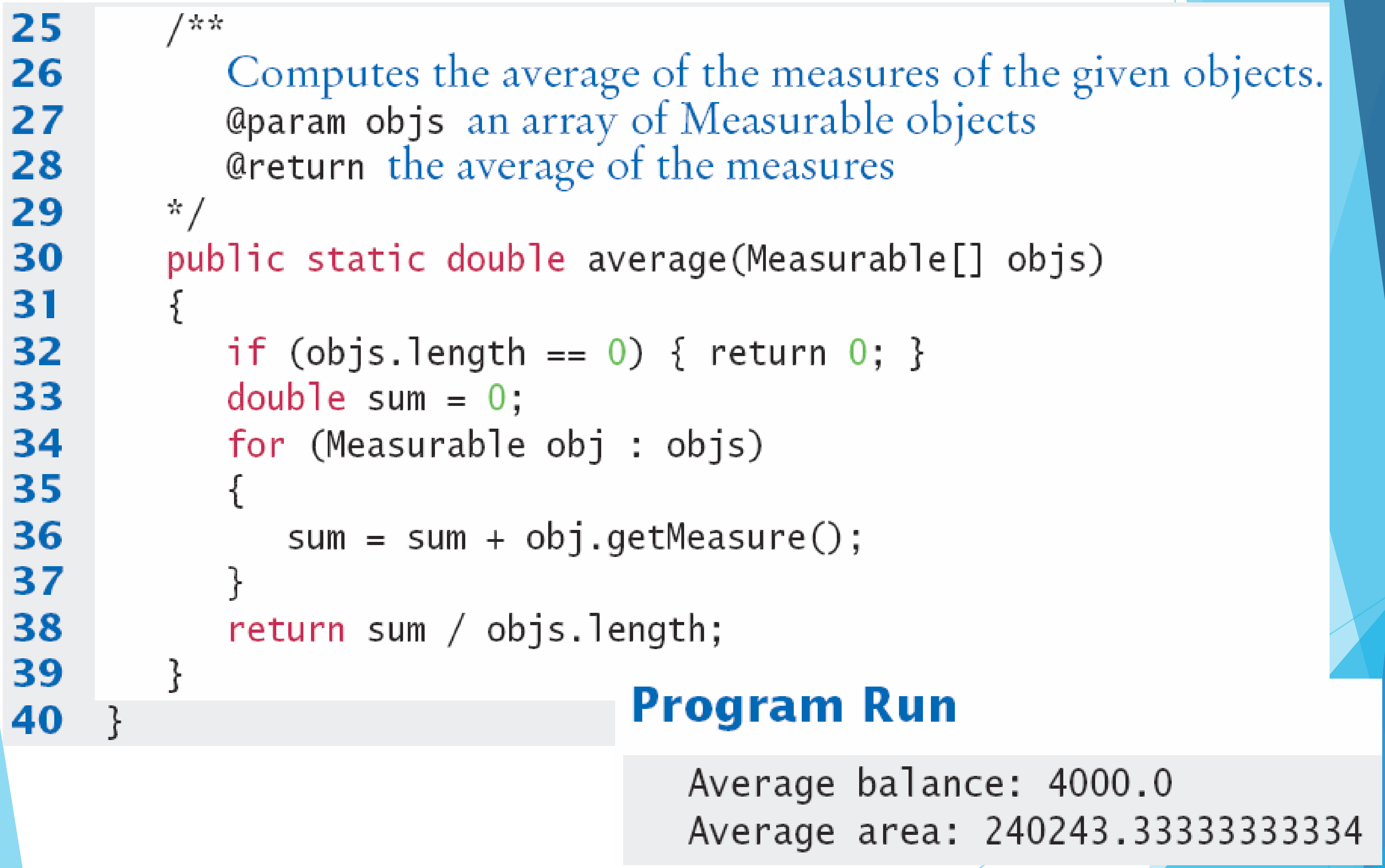


MeasureableDemo.java (1)



```
/**
   This program demonstrates the measurable BankAccount and Country classes.
*/
public class MeasurableDemo
{
   public static void main(String[] args)
   {
      Measurable[] accounts = new Measurable[3];
      accounts[0] = new BankAccount(0);
      accounts[1] = new BankAccount(10000);
      accounts[2] = new BankAccount(2000);

      System.out.println("Average balance: "
         + average(accounts));

      Measurable[] countries = new Measurable[3];
      countries[0] = new Country("Uruguay", 176220);
      countries[1] = new Country("Thailand", 514000);
      countries[2] = new Country("Belgium", 30510);

      System.out.println("Average area: "
         + average(countries));
   }
```

MeasureableDemo.java (2)



```
   /**
      Computes the average of the measures of the given objects.
      @param objs  an array of Measurable objects
      @return  the average of the measures
   */
   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;
   }
}
```

Program Run

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

Common Error

- 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;
  }
  . . .
}
```

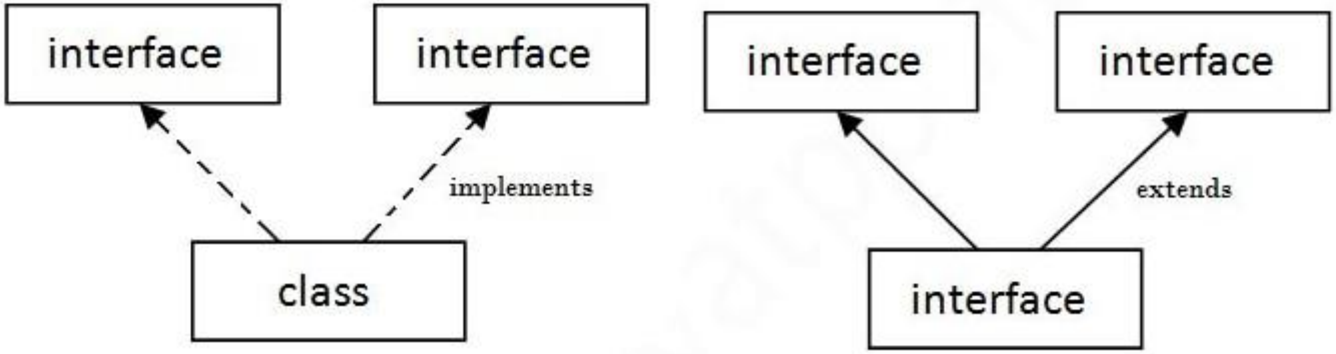
Interface Constants

- 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`.

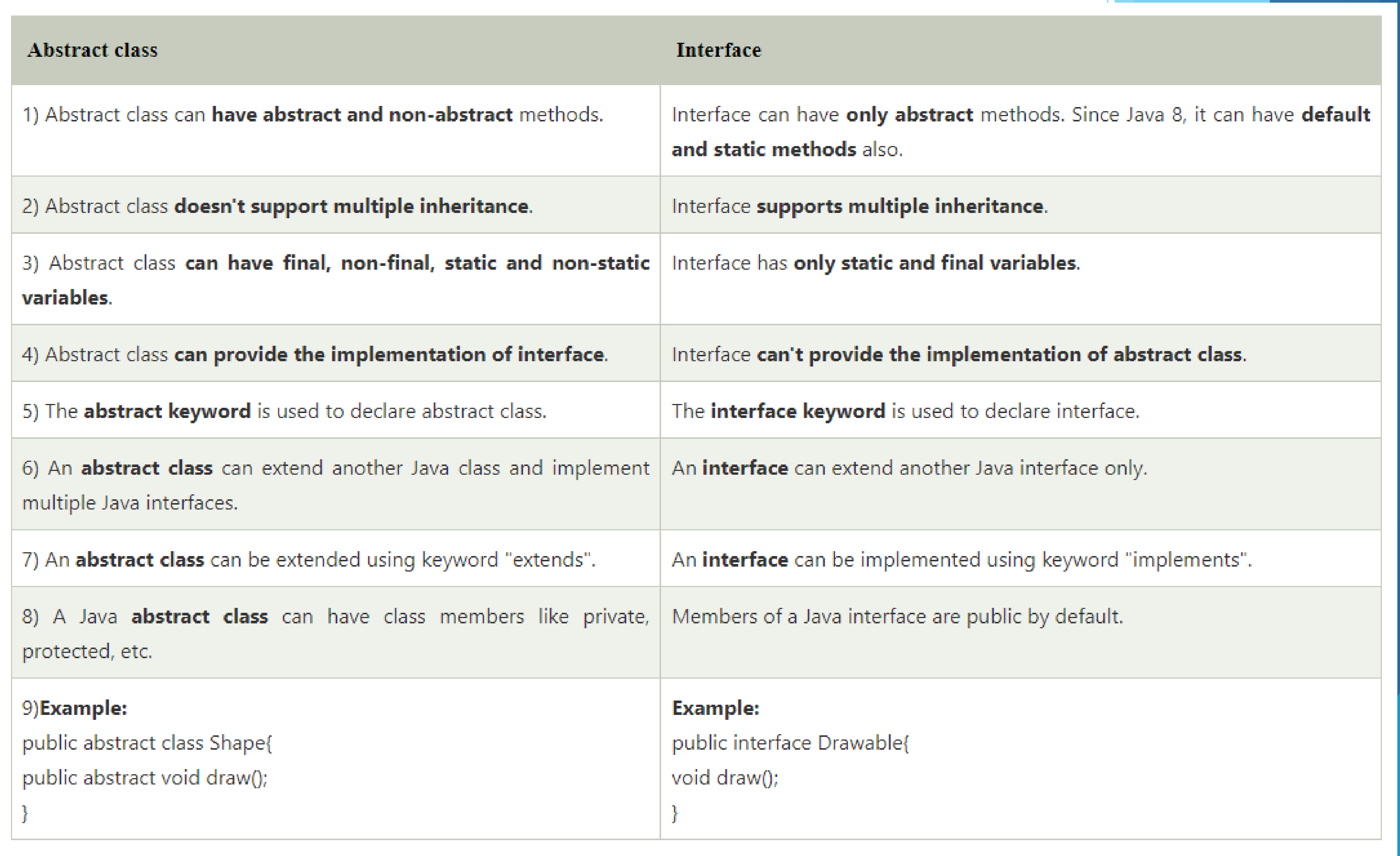
```
public interface SwingConstants
{
   int NORTH = 1;
   int NORTHEAST = 2;
   int EAST = 3;
   . . .
}
```

Multiple Inheritance

- A Java class can only extend one parent class.
  - Multiple inheritance is not allowed.
- However, an interface can extend more than one parent interface.
- Moreover, a class can implement multiple interfaces.



Abstract classes vs Interfaces



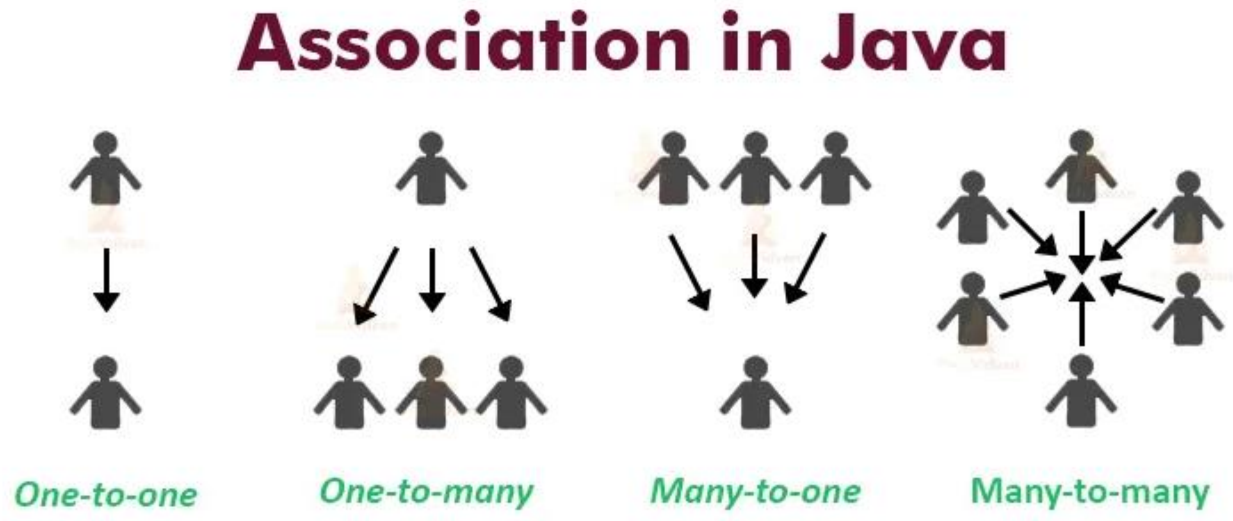
Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Association

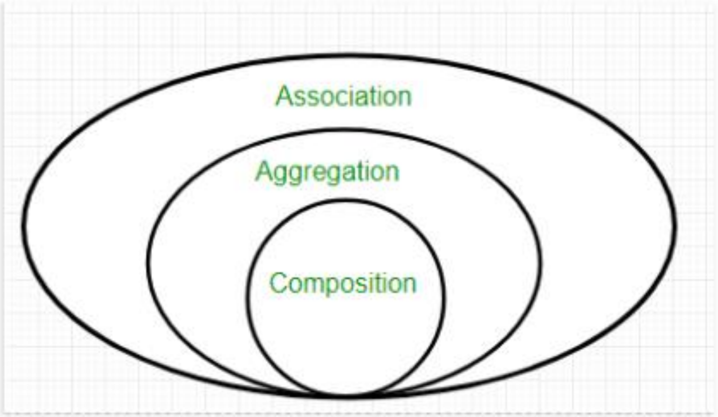
- Association in Java is one of the building blocks and the most basic concept of object-oriented programming.
- Association is a connection or relationship between two separate classes.
- It shows how objects of two classes are associated with each other.
- The Association defines the multiplicity between objects.
- We can describe the Association as a has-a relationship between the classes.

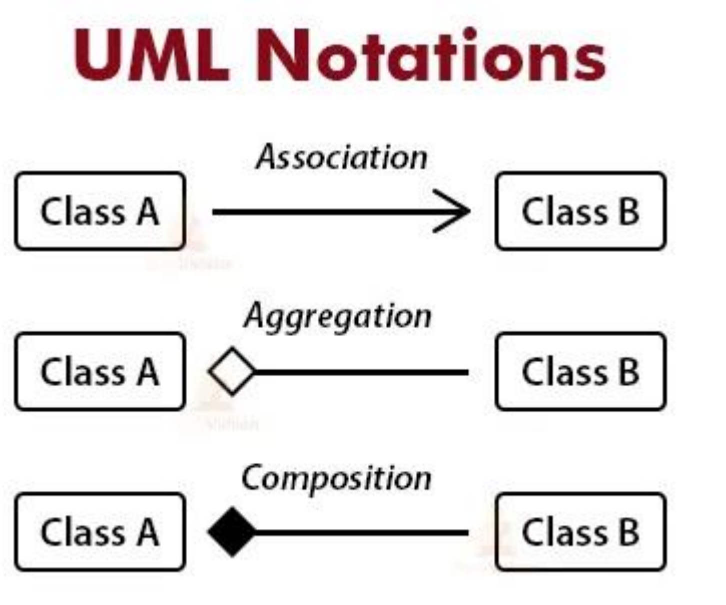
Association



Association

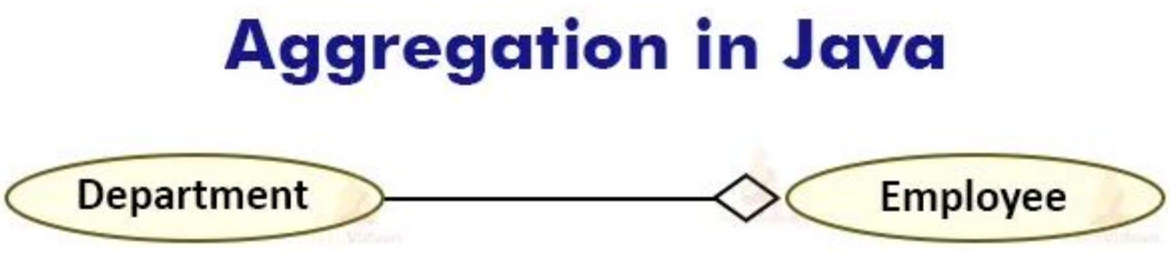
- Two forms;
  - Composition
  - Aggregation



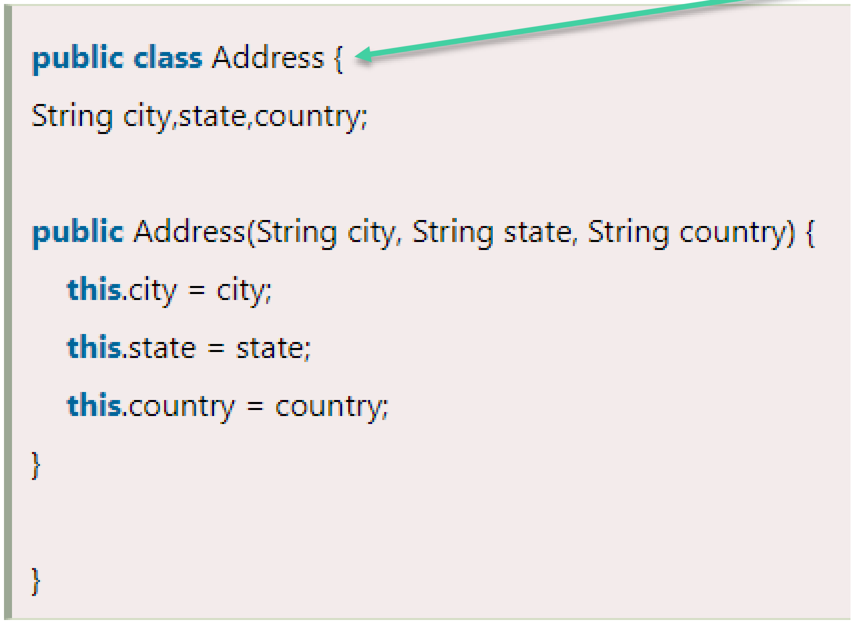


Aggregation

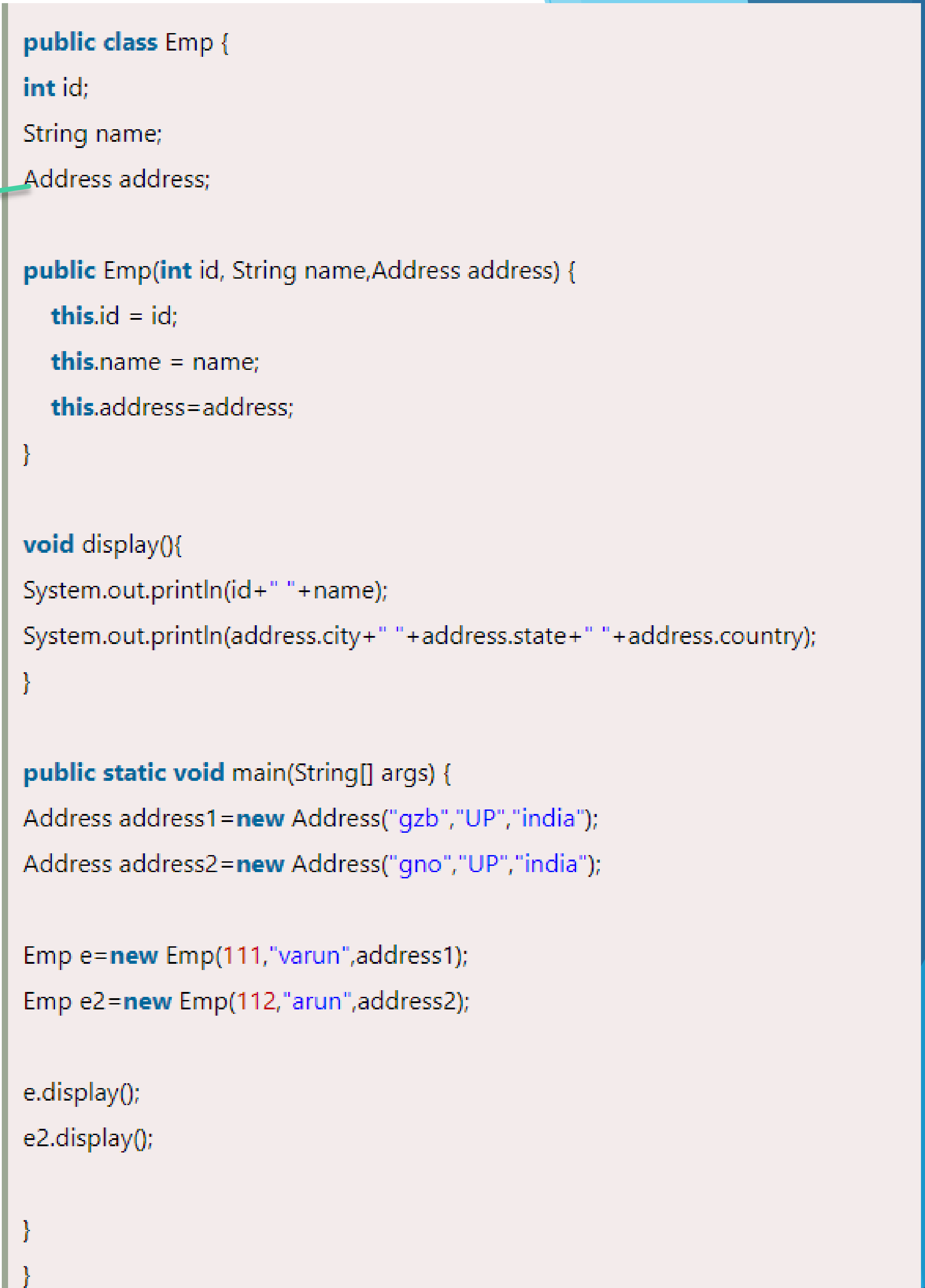
- It represents the Has-A relationship between classes.
- It is a **unidirectional association** i.e. a one-way relationship.
  - For example, a department can have students but vice versa is not possible and thus unidirectional in nature.
- In Aggregation, **both the entries can survive individually** which means ending one entity will not affect the other entity.



Aggregation

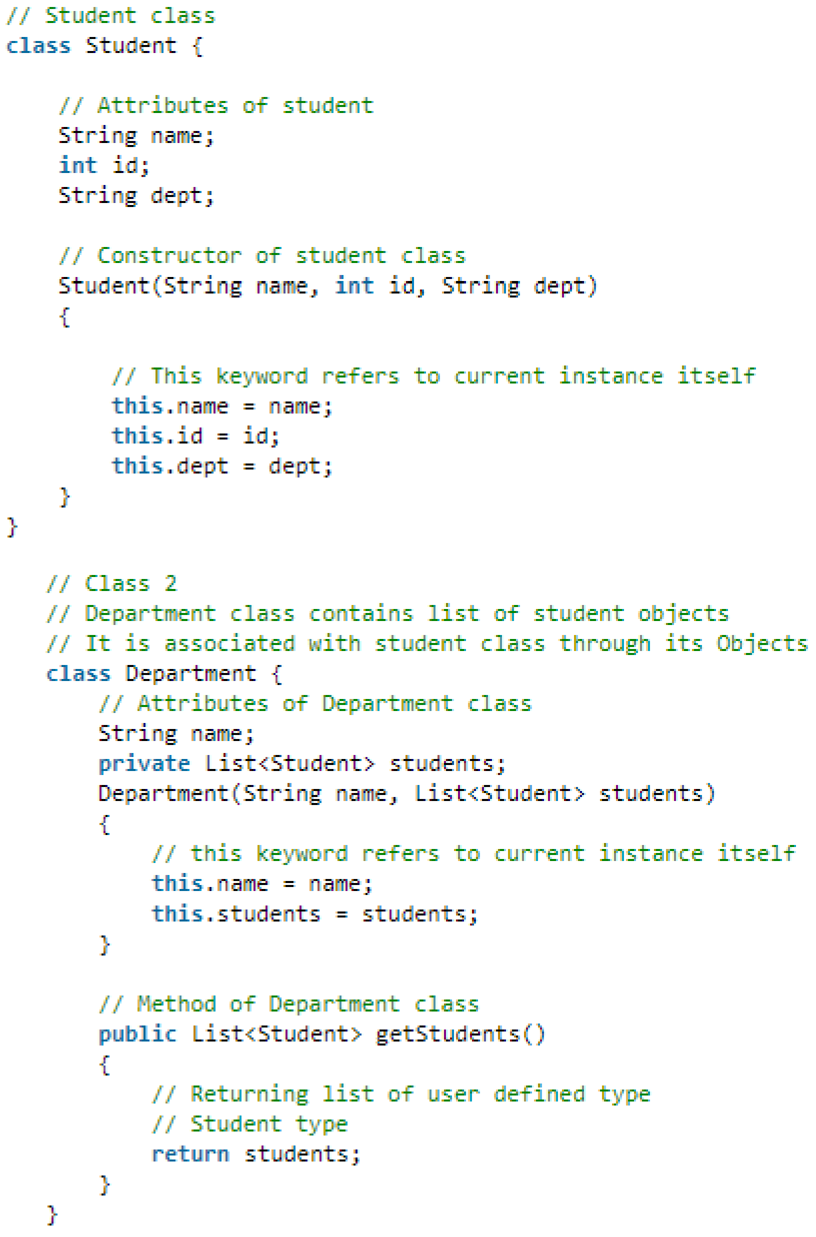


```
public class Address {
String city,state,country;

public Address(String city, String state, String country) {
    this.city = city;
    this.state = state;
    this.country = country;
}


}
```

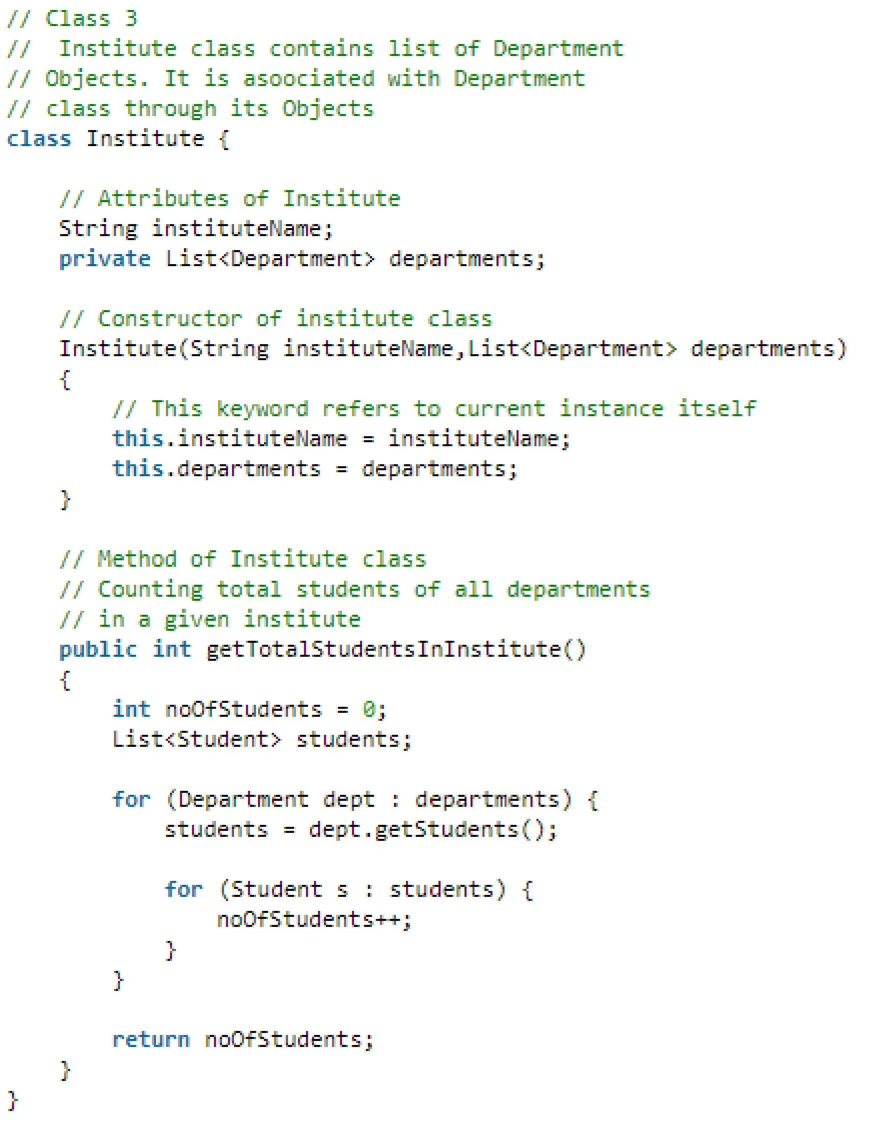


```
public class Emp {
int id;
String name;
Address address;

public Emp(int id, String name,Address address) {
    this.id = id;
    this.name = name;
    this.address=address;
}

void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}

public static void main(String[] args) {
Address address1=new Address("gzb","UP","india");
Address address2=new Address("gno","UP","india");

Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);

e.display();
e2.display();

}
}
```

Aggregation



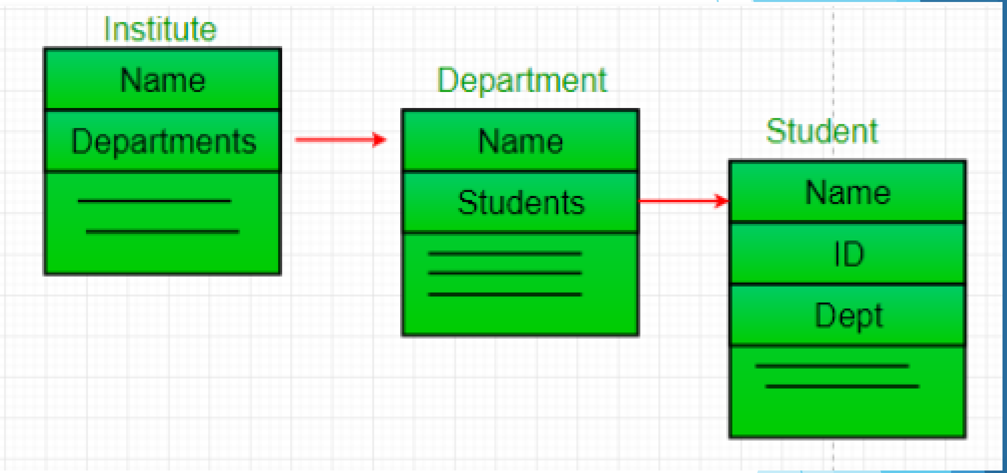
```
// Student class
class Student {

    // Attributes of student
    String name;
    int id;
    String dept;

    // Constructor of student class
    Student(String name, int id, String dept)
    {

        // This keyword refers to current instance itself
        this.name = name;
        this.id = id;
        this.dept = dept;
    }
}

   // Class 2
   // Department class contains list of student objects
   // It is associated with student class through its Objects
   class Department {
       // Attributes of Department class
       String name;
       private List<Student> students;
       Department(String name, List<Student> students)
       {
           // this keyword refers to current instance itself
           this.name = name;
           this.students = students;
       }

       // Method of Department class
       public List<Student> getStudents()
       {
           // Returning list of user defined type
           // Student type
           return students;
       }
   }
```



```
// Class 3
//  Institute class contains list of Department
// Objects. It is asoociated with Department
// class through its Objects
class Institute {

    // Attributes of Institute
    String instituteName;
    private List<Department> departments;

    // Constructor of institute class
    Institute(String instituteName,List<Department> departments)
    {
        // This keyword refers to current instance itself
        this.instituteName = instituteName;
        this.departments = departments;
    }

    // Method of Institute class
    // Counting total students of all departments
    // in a given institute
    public int getTotalStudentsInInstitute()
    {
        int noOfStudents = 0;
        List<Student> students;

        for (Department dept : departments) {
            students = dept.getStudents();

            for (Student s : students) {
                noOfStudents++;
            }
        }

        return noOfStudents;
    }
}
```

Aggregation

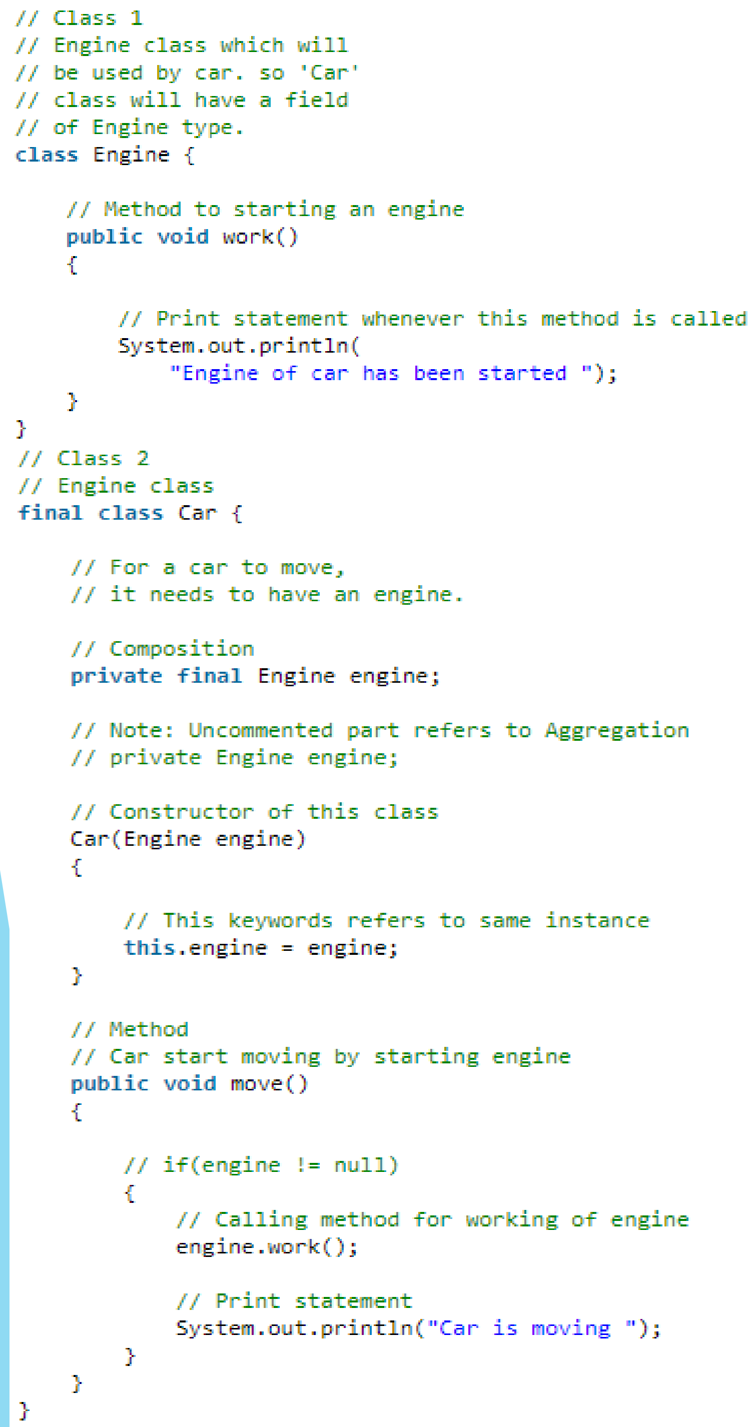
```
// Class 4
// main class
class GFG {

    // main driver method
    public static void main(String[] args)
    {
        // Creating object of Student class inside main()
        Student s1 = new Student("Mia", 1, "CSE");
        Student s2 = new Student("Priya", 2, "CSE");
        Student s3 = new Student("John", 1, "EE");
        Student s4 = new Student("Rahul", 2, "EE");

        // Creating a List of CSE Students
        List<Student> cse_students = new ArrayList<Student>();

        // Adding CSE students
        cse_students.add(s1);
        cse_students.add(s2);

        // Creating a List of EE Students
        List<Student> ee_students
            = new ArrayList<Student>();

        // Adding EE students
        ee_students.add(s3);
        ee_students.add(s4);

        // Creating objects of EE and CSE class inside
        // main()
        Department CSE = new Department("CSE", cse_students);
        Department EE = new Department("EE", ee_students);

        List<Department> departments = new ArrayList<Department>();
        departments.add(CSE);
        departments.add(EE);

        // Lastly creating an instance of Institute
        Institute institute = new Institute("BITS", departments);

        // Display message for better readibility
        System.out.print("Total students in institute: ");

        // Calling method to get total number of students
        // in institute and printing on console
        System.out.print(institute.getTotalStudentsInInstitute());
    }
}
```



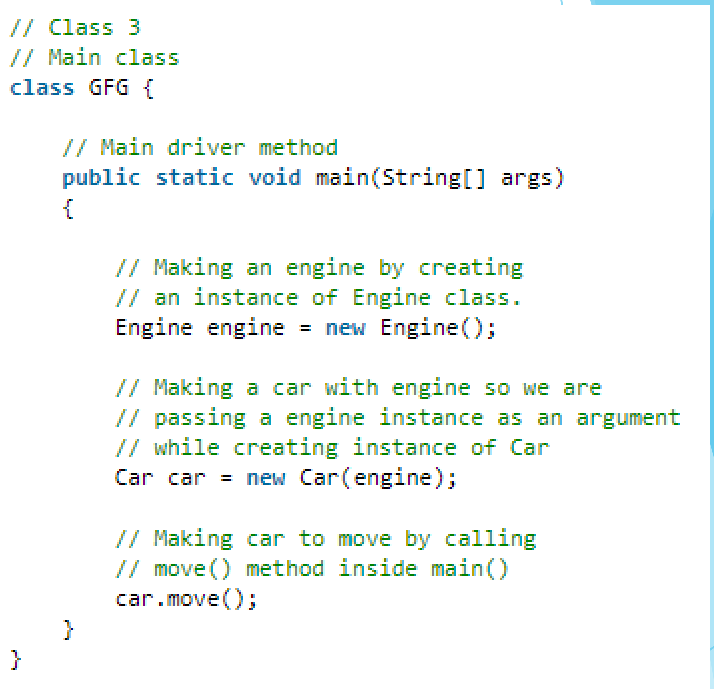
Composition

- Composition is a restricted form of Aggregation.
- Two entities are highly dependent on each other.
- It represents **part-of** relationship.
- In composition, both entities are dependent on each other.
- When there is a composition between two entities, the composed object **cannot exist** without the other entity.

Composition



```
// Class 1
// Engine class which will
// be used by car. so 'Car'
// class will have a field
// of Engine type.
class Engine {

    // Method to starting an engine
    public void work()
    {

        // Print statement whenever this method is called
        System.out.println(
            "Engine of car has been started ");
    }
}
// Class 2
// Engine class
final class Car {

    // For a car to move,
    // it needs to have an engine.

    // Composition
    private final Engine engine;

    // Note: Uncommented part refers to Aggregation
    // private Engine engine;

    // Constructor of this class
    Car(Engine engine)
    {

        // This keywords refers to same instance
        this.engine = engine;
    }

    // Method
    // Car start moving by starting engine
    public void move()
    {

        // if(engine != null)
        {
            // Calling method for working of engine
            engine.work();

            // Print statement
            System.out.println("Car is moving ");
        }
    }
}
```



```
// Class 3
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Making an engine by creating
        // an instance of Engine class.
        Engine engine = new Engine();

        // Making a car with engine so we are
        // passing a engine instance as an argument
        // while creating instance of Car
        Car car = new Car(engine);

        // Making car to move by calling
        // move() method inside main()
        car.move();
    }
}
```

Benefits of using Composition

- Composition allows us to reuse the code.
- In Java, we can use multiple Inheritance by using the composition concept.
- The Composition provides better test-ability of a class.
- Composition allows us to easily replace the composed class implementation with a better and improved version.
- Composition allows us to dynamically change our program's behavior by changing the member objects at run time.

Aggregation vs Composition

- **Dependency**
  - Aggregation implies a relationship where the child **can exist independently** of the parent.
  - Composition implies a relationship where the child **cannot exist independent** of the parent.
- **Type of Relationship**
  - Aggregation relation is **“has-a” relation.**
  - Composition is **“part-of”** relation.
- **Type of association**
  - Composition is a **strong** Association.
  - Whereas Aggregation is a **weak** Association.

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.