Advanced Computer Programming

[Lecture 07]

Saeed Reza Kheradpisheh

kheradpisheh@ut.ac.ir

Department of Computer Science
Shahid Beheshti University
Spring 1397-98

INHERITANCE

Objects from related classes usually share common behavior. For example, shovels, rakes, and clippers all perform gardening tasks. By using **inheritance**, you will be able to share code between classes and provide services that can be used by multiple classes.

Inheritance Hierarchies

Definition

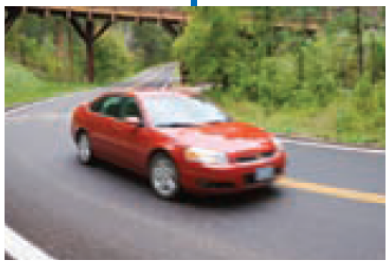
In object-oriented design, **inheritance** is a relationship between a more general class (called the **superclass**) and a more specialized class (called the **subclass**).

- A subclass inherits data and behavior from a superclass.
- You can always use a subclass object in place of a superclass object (substitution principle).

An Inheritance Hierarchy of Vehicle Classes

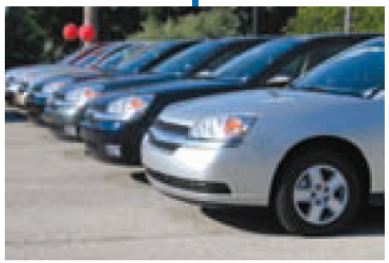
Vehicle

Motorcycle



Car

Truck



Sedan

SUV

Taking a Quizz!

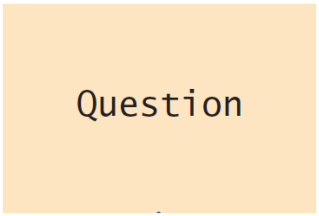
A quiz consists of questions, and there are different kinds of questions:

- Fill-in-the-blank
- Choice (single or multiple)
- Numeric (where an approximate answer is ok)
- Free response

Taking a Quizz!

A quiz consists of questions, and there are different kinds of questions:

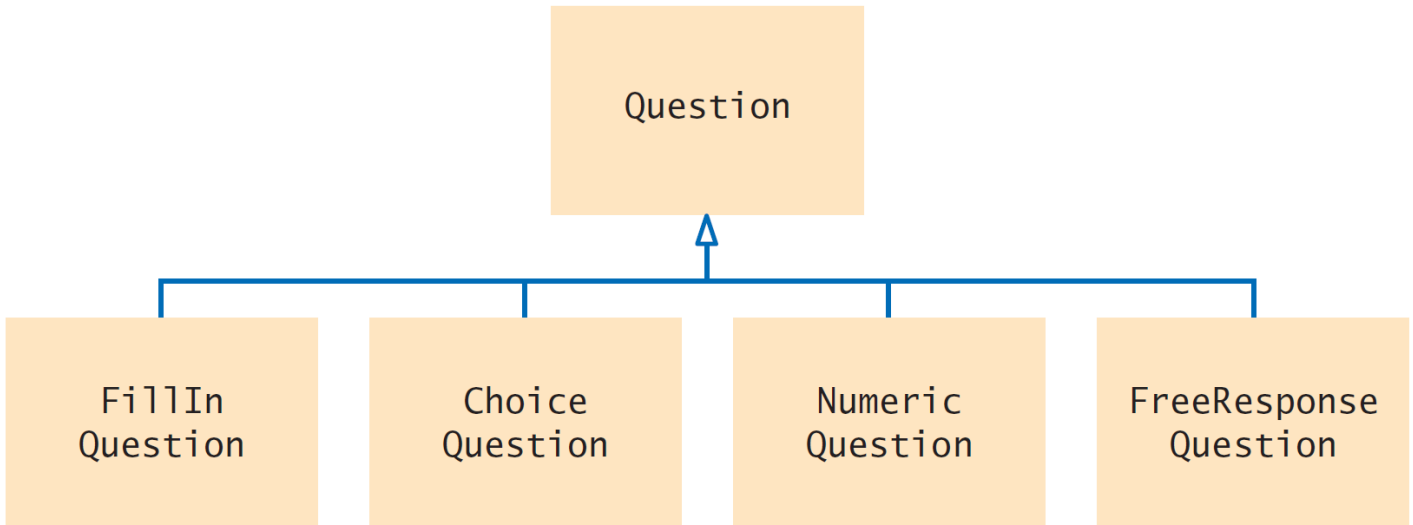
- Fill-in-the-blank
- Choice (single or multiple)
- Numeric (where an approximate answer is ok)
- Free response



Taking a Quizz!

A quiz consists of questions, and there are different kinds of questions:

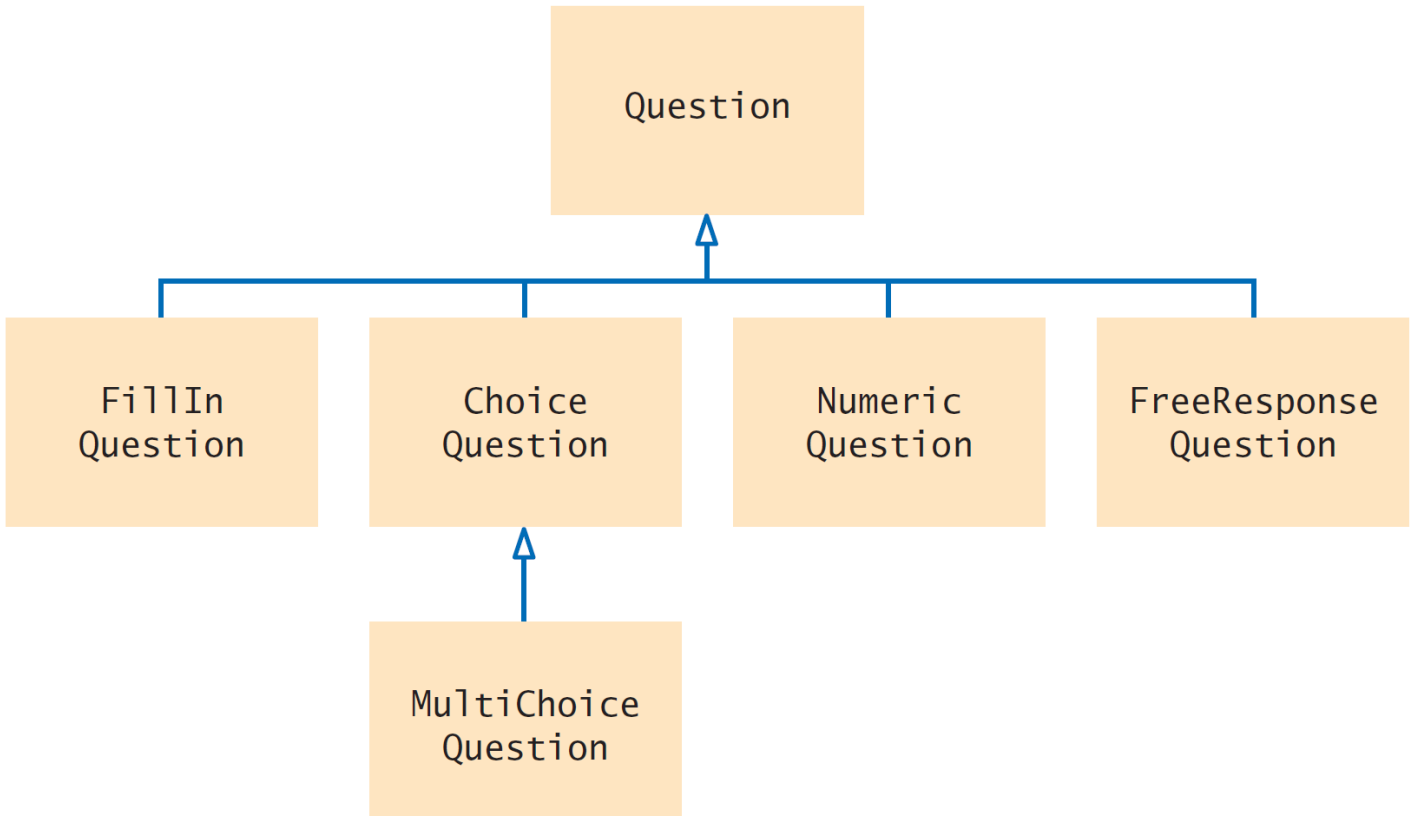
- Fill-in-the-blank
- Choice (single or multiple)
- Numeric (where an approximate answer is ok)
- Free response



Taking a Quizz!

A quiz consists of questions, and there are different kinds of questions:

- Fill-in-the-blank
- Choice (single or multiple)
- Numeric (where an approximate answer is ok)
- Free response



The `Question` Type

At the root of this hierarchy is the `Question` type. What is it supposed to do?

- Display its text.
- Check whether a given response is a correct answer.

Data members:

- `String text`
- `String answer`

Public methods:

- `void setText(String questionText)`
- `void setAnswer(String correctResponse)`
- `boolean checkAnswer(String response)`
- `void display()`

Exercise (`Question.java`)

Implement the class `Question`.

Self Check

- Consider classes `Manager` and `Employee`. Which should be the superclass and which should be the subclass?
- Consider the method `doSomething(Car c)`. List all vehicle classes from Figure 1 whose objects cannot be passed to this method.
- Should a class `Quiz` inherit from the class Question? Why or why not?

Do not overuse Inheritance!

Programming Tip

Use a single class for variation in values, Inheritance for variation in behavior.

Consider two different applications that work with regular cars and hybrid cars:

- Tracking the fuel efficiency of cars by logging the distance traveled and the refueling amounts.

Do not overuse Inheritance!

Programming Tip

Use a single class for variation in values, Inheritance for variation in behavior.

Consider two different applications that work with regular cars and hybrid cars:

- Tracking the fuel efficiency of cars by logging the distance traveled and the refueling amounts.
  No differences in behavior, no need for inheritance.
- Showing how to repair different kinds of vehicles.

Do not overuse Inheritance!

Programming Tip

Use a single class for variation in values, Inheritance for variation in behavior.

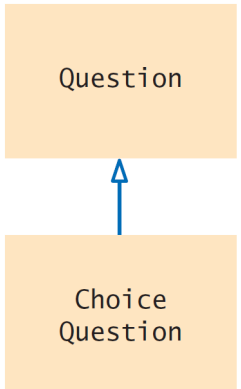
Consider two different applications that work with regular cars and hybrid cars:

- Tracking the fuel efficiency of cars by logging the distance traveled and the refueling amounts.
  No differences in behavior, no need for inheritance.
- Showing how to repair different kinds of vehicles.
  Different behaviors, need for different classes and inheritance.

Implementing Subclasses

- In Java, you form a subclass by specifying what makes the subclass different from its superclass.
- Subclass objects automatically have the instance variables that are declared in the superclass. *You only declare instance variables that are not part of the superclass objects.*
  - However, the private instance variables of the superclass are inaccessible.
- The subclass inherits all public methods 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.*

Implementing the Subclass `ChoiceQuestion`



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

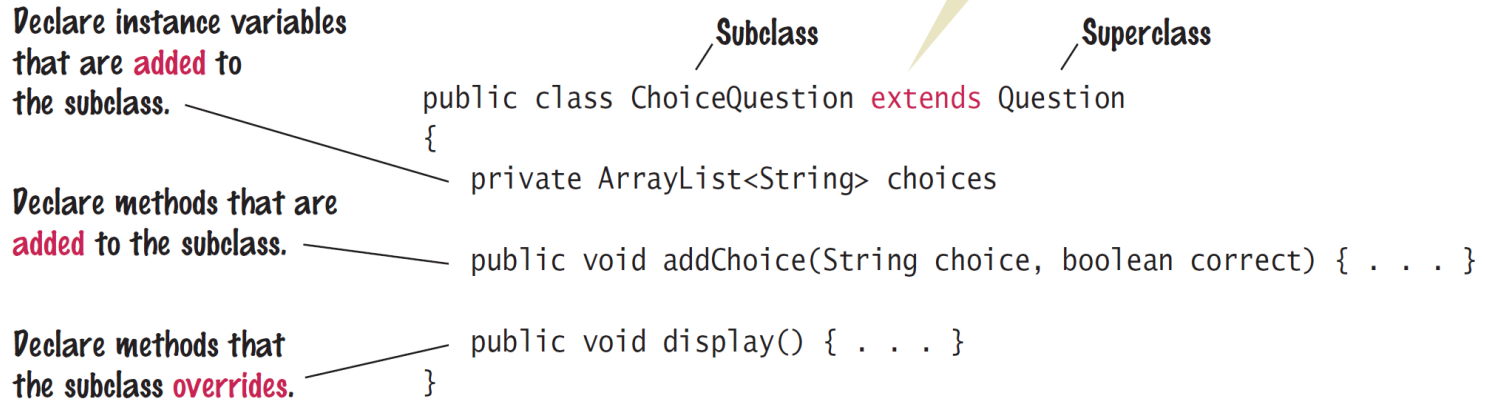
- Its objects store the various choices for the answer.
- There is a method for adding answer choices.
- The `display` method of the `ChoiceQuestion` class shows these choices so that the respondent can choose one of them.

extends

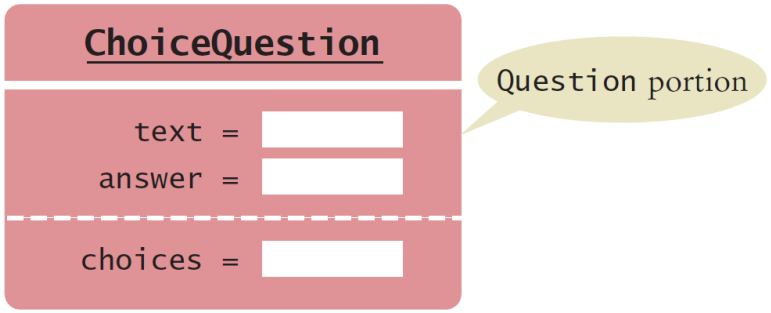
The **`extends`** reserved word indicates that a class inherits from a superclass.

Syntax

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



Back to `ChoiceQuestion`



- It adds an additional instance variable, `choices`.
- The `addChoice` method is specific to the `ChoiceQuestion` class. You can only apply it to `ChoiceQuestion` objects, not general Question objects.
- The `display` method is a method that already exists in the superclass. The subclass **overrides** this method, so that the choices can be properly displayed.

Overriding Methods

Usage

An **overriding method** can extend or replace the functionality of the superclass method.

Consider the `display` method of the `ChoiceQuestion` class:

- Display the question text (superclass can do it).
- Display the answer choices (the extension).

Usage

Use the reserved word **`super`** to call a superclass method.

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

```
public void display()
{
   // Display the question text
   display(); // Error—invokes this.display()
   . . .
}
```

Exercise (`ChoiceQuestion.java`)

Implement the class `ChoiceQuestion`.

Self Check

- Suppose `q` is an object of the class `Question` and `cq` an object of the class `ChoiceQuestion`. Which of the following calls are legal?
  1. `q.setAnswer(response)`
  2. `cq.setAnswer(response)`
  3. `q.addChoice(choice, true)`
  4. `cq.addChoice(choice, true)`

Self Check

- Suppose `q` is an object of the class `Question` and `cq` an object of the class `ChoiceQuestion`. Which of the following calls are legal?
  1. `q.setAnswer(response)`
  2. `cq.setAnswer(response)`
  3. `q.addChoice(choice, true)`
  4. `cq.addChoice(choice, true)`
- 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));
      }
   }
}
```

Self Check

- Look again at the implementation of the `addChoice` method that calls the `setAnswer` method of the superclass. Why don't you need to call `super.setAnswer`?

Overriding or Overloading?

- In Java, two methods can have the same name, provided they differ in their parameter types (**overloaded methods**).
- Overloading is different from overriding, where a subclass method provides an implementation of a method whose parameter variables have the same types.

Overriding or Overloading?

- In Java, two methods can have the same name, provided they differ in their parameter types (**overloaded methods**).
- Overloading is different from overriding, where a subclass method provides an implementation of a method whose parameter variables have the same types.
- If you mean to override a method but use a parameter variable with a different type, then you accidentally introduce an overloaded method.
- When overriding a method, be sure to check that the types of the parameter variables match exactly.

Calling the Superclass Constructor

- A subclass constructor can only initialize the instance variables of the subclass.
- The superclass instance variables also need to be initialized.
- Mostly constructors are in charge of initializing instance variables.
- In order to specify another constructor, you use the `super` reserved word, together with the arguments of the superclass constructor, as the first statement of the subclass constructor.

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

Polymorphism

Definition

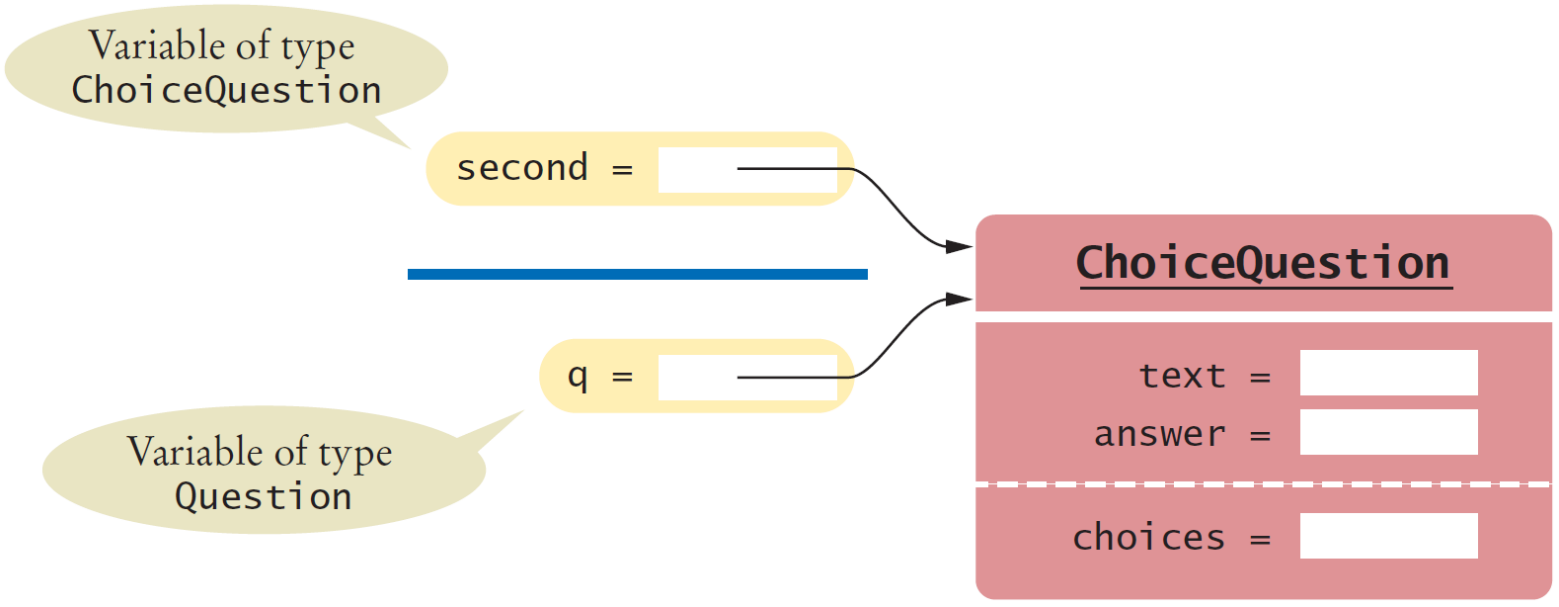
Polymorphism ("having multiple shapes") allows us to manipulate objects that share a set of tasks, even though the tasks are executed in different ways.

Can we write a program that shows a mixture of both question types?

```
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
```

Polymorphism



Definition

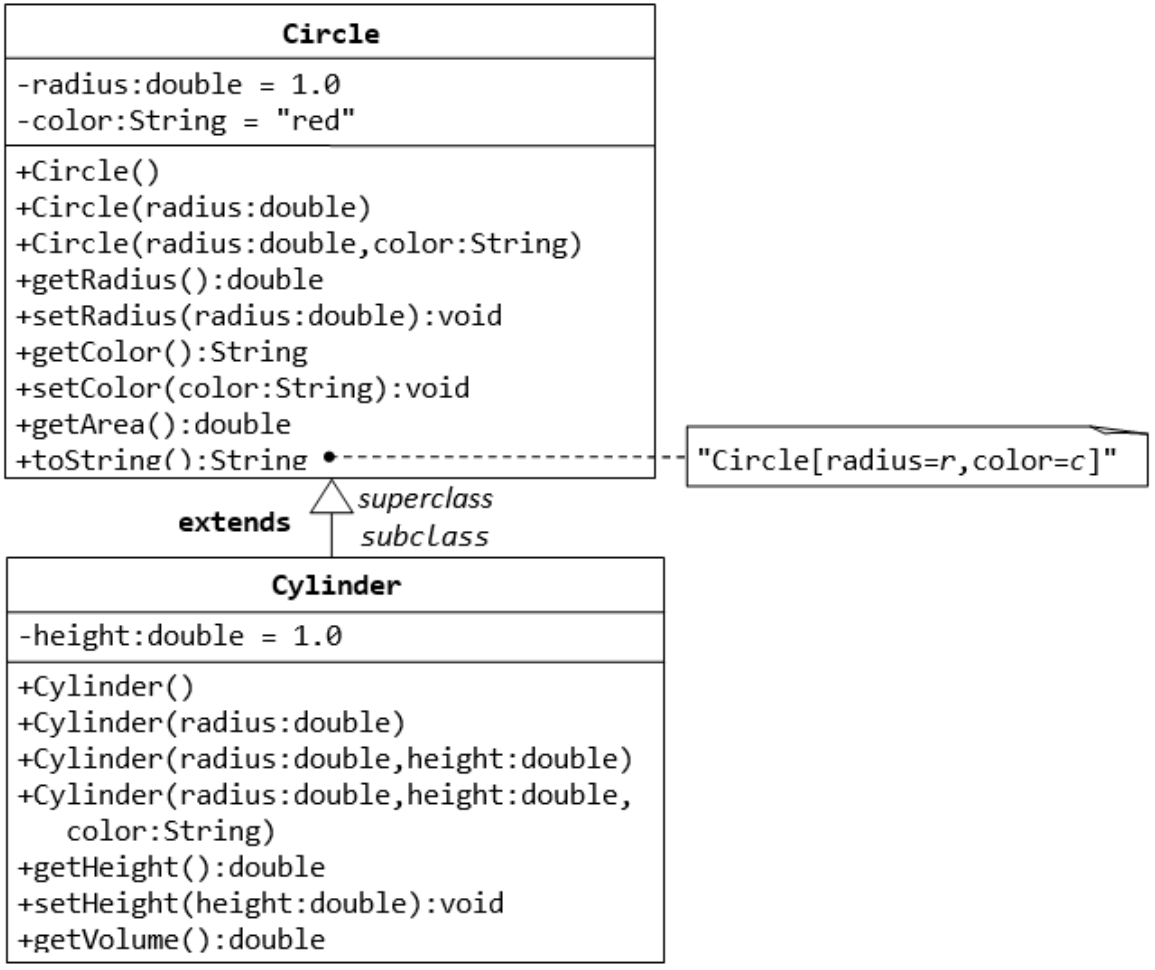
In Java, method calls are always determined by the type of the actual object, not the type of the variable containing the object reference. This is called **dynamic method lookup**.

Self Check

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

1. `BankAccount account = new SavingsAccount();`
2. `SavingsAccount account2 = new BankAccount();`
3. `BankAccount account = null;`
4. `SavingsAccount account2 = account;`

Example of Inheritance: The Circle and Cylinder Classes



Abstract Classes

Definition

- An **abstract method** is a method whose implementation is not specified.
- An **abstract class** is a class that cannot be instantiated.

Abstract Classes

Definition

- An **abstract method** is a method whose implementation is not specified.
- An **abstract class** is a class that cannot be instantiated.

- Sometimes, it is desirable to force programmers to override a method.
- An abstract method has no implementation. This forces the implementors of subclasses to specify concrete implementations of this method. e.g.

```
public abstract void deductFees();
```

- You cannot construct objects of classes with abstract methods. These classes are called *abstract classes*.
- In Java, you must declare all abstract classes with the reserved word `abstract`. e.g.

```
public abstract class Account
```

Abstract Classes

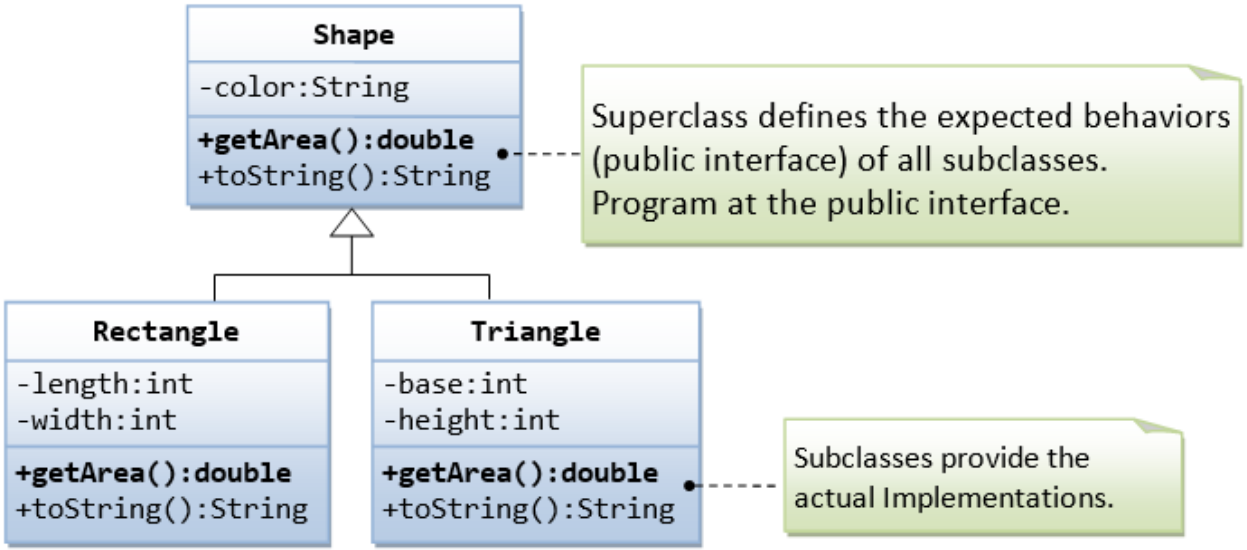
- Note that you cannot construct an object of an abstract class, but you can still have an object reference whose type is an abstract class. Why?

Abstract Classes

- Note that you cannot construct an object of an abstract class, but you can still have an object reference whose type is an abstract class. Why?
  Dynamic lookup.

```
Account anAccount; // OK
anAccount = new Account(); // Error—Account is abstract
anAccount = new SavingsAccount(); // OK
anAccount = null; // OK
```

Example of Abstract Class: Shapes



- Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on.
- We should design a superclass called Shape, which defines the public interfaces (or behaviors) of all the shapes.

Final Methods and Classes

Usage

If you want to prevent other programmers from creating subclasses or from overriding certain methods, use the **`final`** reserved word.

- Finalizing a class:

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

- Finalizing a method:

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

Protected Access

Definition

Protected data or method in an object can be accessed by the methods of the object's class and all its subclasses.

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

The `Object` Class

- In Java, every class that is declared without an explicit `extends` clause automatically extends the class `Object`.
- The `Object` class defines several very general methods, including:
  - `toString`, which yields a string describing the object.
  - `equals`, which compares objects with each other.
  - `hashCode`, which yields a numerical code for storing the object in a set.

Overriding the `toString` Method

- Returns a string representation for each object.
- It is called automatically whenever you concatenate a string with an object.
- Without overriding the `toString` method, it returns the **hash code** of the object.

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

The `equals` Method

Usage

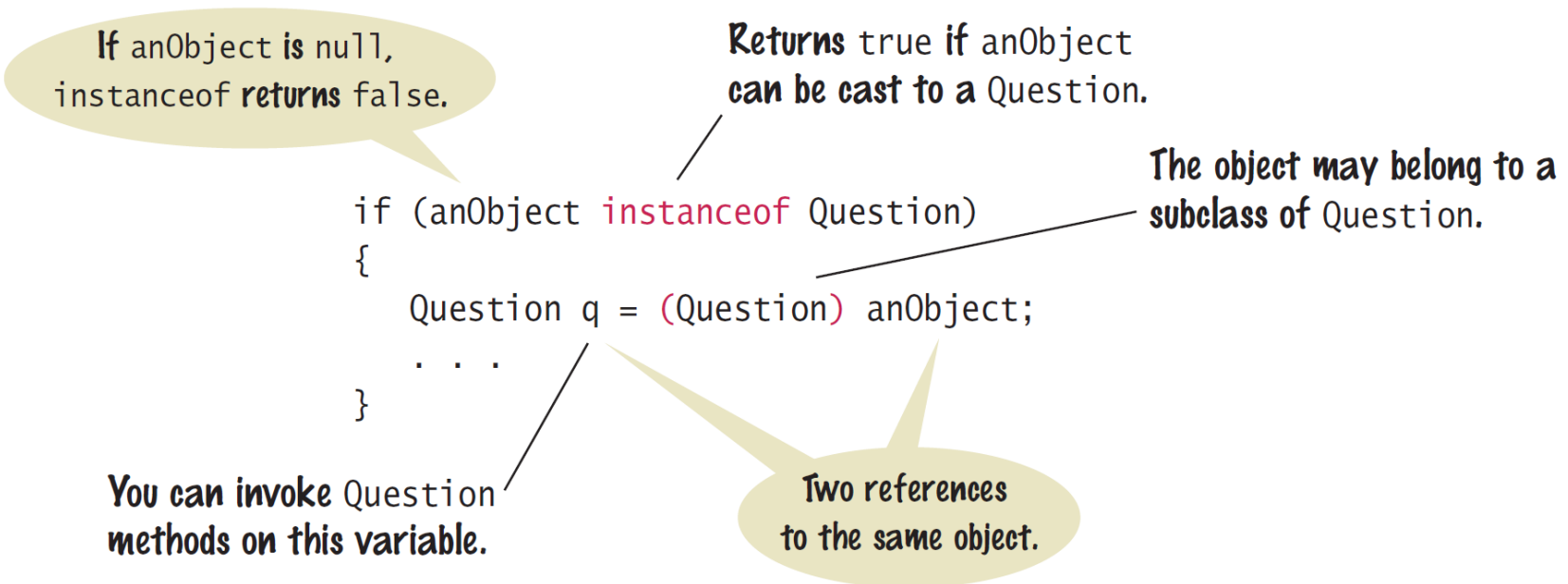
The **`equals`** method checks whether two objects have the same contents.
This is different from the test with the == operator, which tests whether two references are identical.
The **`equals`** method acts the same as == if it is not overridden.

The `instanceof` Operator

Usage

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

Syntax *object* `instanceof` *TypeName*



INTERFACE

It is often possible to design a general and reusable mechanism for processing objects by focusing on the essential operations that an algorithm needs. You use **interface types** to express these operations.

Interface Type

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 always public.

Interface methods have no implementation.

A class can implement one or more interface types.

Other BankAccount methods.

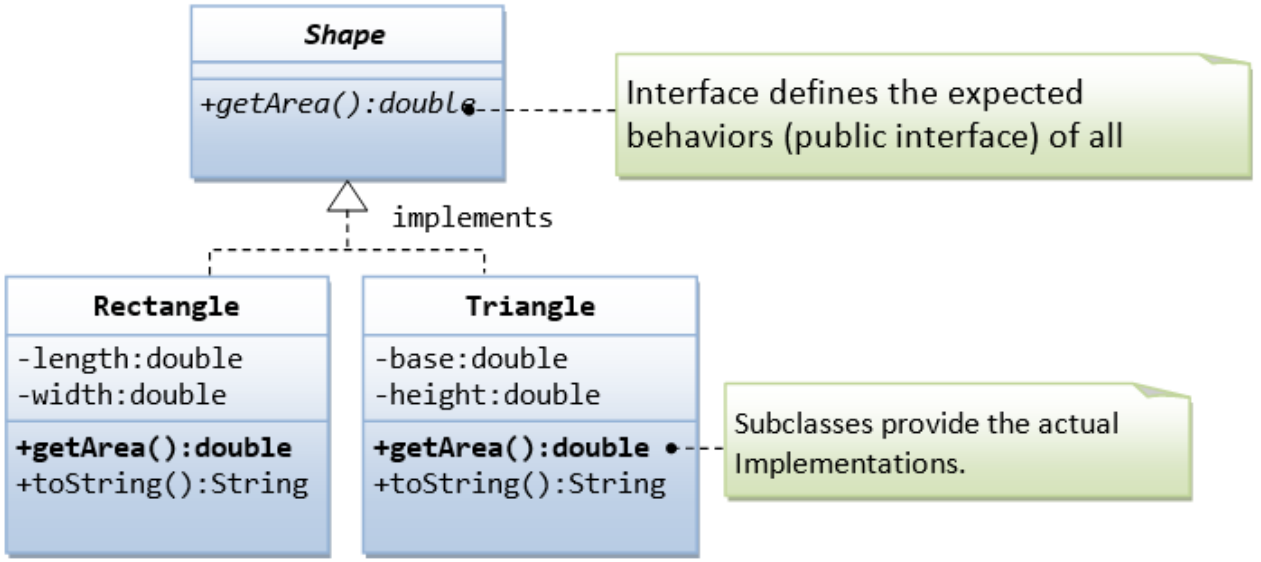
Implementation for the method that was declared in the interface type.

Interface Type V.S. Class

An interface type is similar to a class, but there are several important differences:

- All methods in an interface type are abstract; that is, they have a name, parameter variables, and a return type, but they dont 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.

Example of Interfaces: Shapes



- Suppose that our program uses many kinds of shapes, such as triangle, rectangle and so on.
- We should design an Interface called Shape, which defines the public interfaces (or behaviors) of all the shapes.