Chapter 9 – Inheritance

Employing Information Hiding

- Keyword **`protected`**

Modifier	Classes and interfaces	Methods and variables
default (no modifier)	Visible in its package.	Inherited by any subclass in the same package as its class. Accessible by any class in the same package as its class.
public	Visible anywhere.	Inherited by all subclasses of its class. Accessible anywhere.
protected	N/A	Inherited by all subclasses of its class. Accessible by any class in the same package as its class.
private	Visible to the enclosing class only	Not inherited by any subclass. Not accessible by any other class.

@Override Annotation

- **Why should we use @Override**
  - Use @Override annotation at the top of the method
- Do it so that you can take advantage of the compiler checking to make sure you actually are overriding a method when you think you are.
  - This way, if you make a common mistake of misspelling a method name or not correctly matching the parameters, you will be warned that you method does not actually override as you think it does.
- Secondly, it makes your code easier to understand because it is more obvious when methods are overwritten.
- Aside from that, there is no other benefits: it does not impact polymorphism subtyping in anyway

Methods You Cannot Override

- `static` methods
- `final` methods
- Methods within `final` classes

A Subclass Cannot Override `static` Methods in Its Superclass

- A subclass cannot override methods declared `static` in the superclass
- A subclass can hide a `static` method in the superclass by declaring a `static` method with the same signature as the `static` method in the superclass
  - Then call the new `static` method from within the subclass or in another class by using a subclass object
  - Within the `static` method of a subclass, you cannot access the parent method by using the `super` object

A Subclass Cannot Override `static` Methods in Its Superclass (cont'd.)

- Although a child class cannot override its parent's `static` methods, it can access its parent's `static` methods in the same way any other class can
  - [name of the parent class].[name of the static method]
  - In non-static methods: super.[name of the static method]

A Subclass Cannot Override `static` Methods in Its Superclass (cont'd.)

```
public class ProfessionalBaseballPlayer extends BaseballPlayer
{
   double salary;
   public static void showOrigins()
   {
      BaseballPlayer.showOrigins();
      System.out.println("The first professional " +
         "major league baseball game was played in 1871");
   }
}
```

Figure 10-22 The `ProfessionalBaseballPlayer` class

A Subclass Cannot Override `final` Methods in Its Superclass

- A subclass cannot override methods declared `final` in the superclass
- `final` modifier
  - Does not allow the method to be overridden

A Subclass Cannot Override `final` Methods in Its Superclass (cont'd.)

- Advantages to making the method `final`:
  - The compiler knows there is only one version of the method
  - It can optimize a program's performance by removing calls to `final` methods
    - Replaces them with expanded code of their definitions at each method call location
    - Called **inlining** the code

A Subclass Cannot Override Methods in a `final` Superclass

- When a class is declared `final`:
  - All of its methods are `final` regardless of which access modifier precedes the method name
  - It cannot be a parent class

A Subclass Cannot Override Methods in a `final` Superclass (cont'd.)

```
public final class HideAndGoSeekPlayer
{
   private int count;
   public void displayRules()
   {
      System.out.println("You have to count to " + count +
         " before you start looking for hiders");
   }
}
public final class ProfessionalHideAndGoSeekPlayer
   extends HideAndGoSeekPlayer
{
   private double salary;
}
```

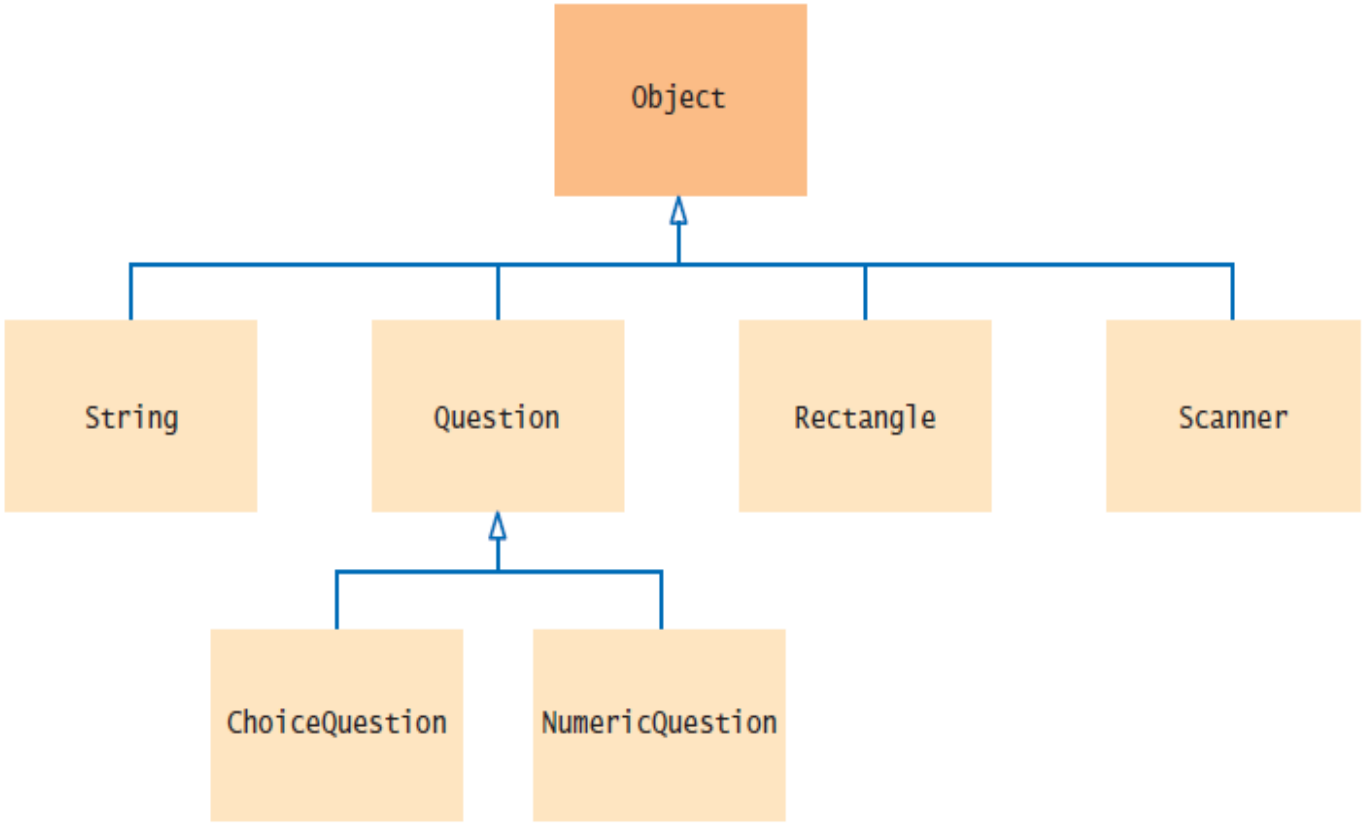
Don't Do It
You cannot extend a `final` class.

Figure 10-28 The `HideAndGoSeekPlayer` and `ProfessionalHideAndGoSeekPlayer` classes

Object: The Cosmic Superclass

- Every class defined without an explicit extends clause automatically extend
- Object:
- The class Object is the direct or indirect superclass of every class in Java. Some methods defined in Object:
  - toString- which yields a string describing the object
  - equals- which compares objects with each other
  - hashCode- which yields a numerical code for storing the object in a set

Object: The Cosmic Superclass



Overriding the equals Method

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

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

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

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

Overriding the equals Method

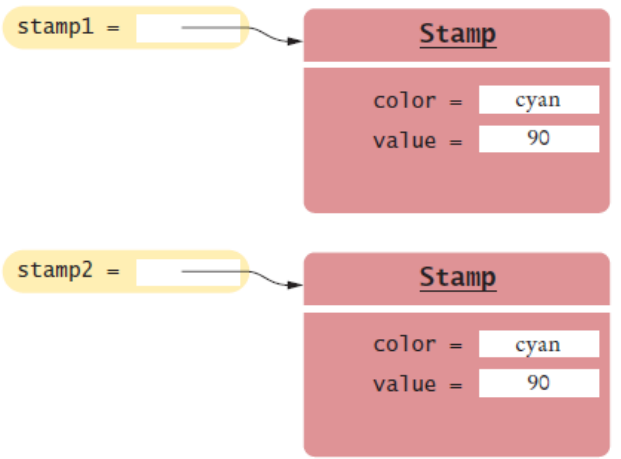


Figure 10 Two References to Equal Objects

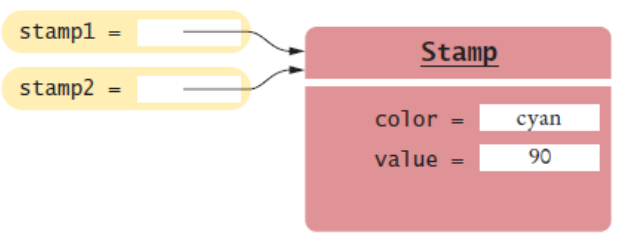


Figure 11 Two References to the Same Object

Overriding the equals Method

- To implement the equals method for a `Stamp` class -

  Override the `equals` method of the `Object` class:

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

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

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

Overriding the equals Method

- After casting, you can compare two Stamps

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

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

The instanceof Operator

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

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

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

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

- If `obj` refers to an object of an unrelated type, "class cast" exception is thrown.
- The `instanceof` operator tests whether an object belongs to a particular type.

```
obj instanceof Question
```

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

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

Abstract Method

- Abstract methods are declared in a parent class
  - The parent class then needs to be declared as an abstract class
  - Abstract methods are only declarations, they do not have definitions.

- By creating abstract methods in the parent class, we define some rules for the derived classes
  - All derived classes must define body for the abstract methods in the derived class.

- Because of the rules, we can easily create polymorphism

Polymorphism

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

Polymorphism

- Let us see a demo

Thank you

- Let me know if you have any questions