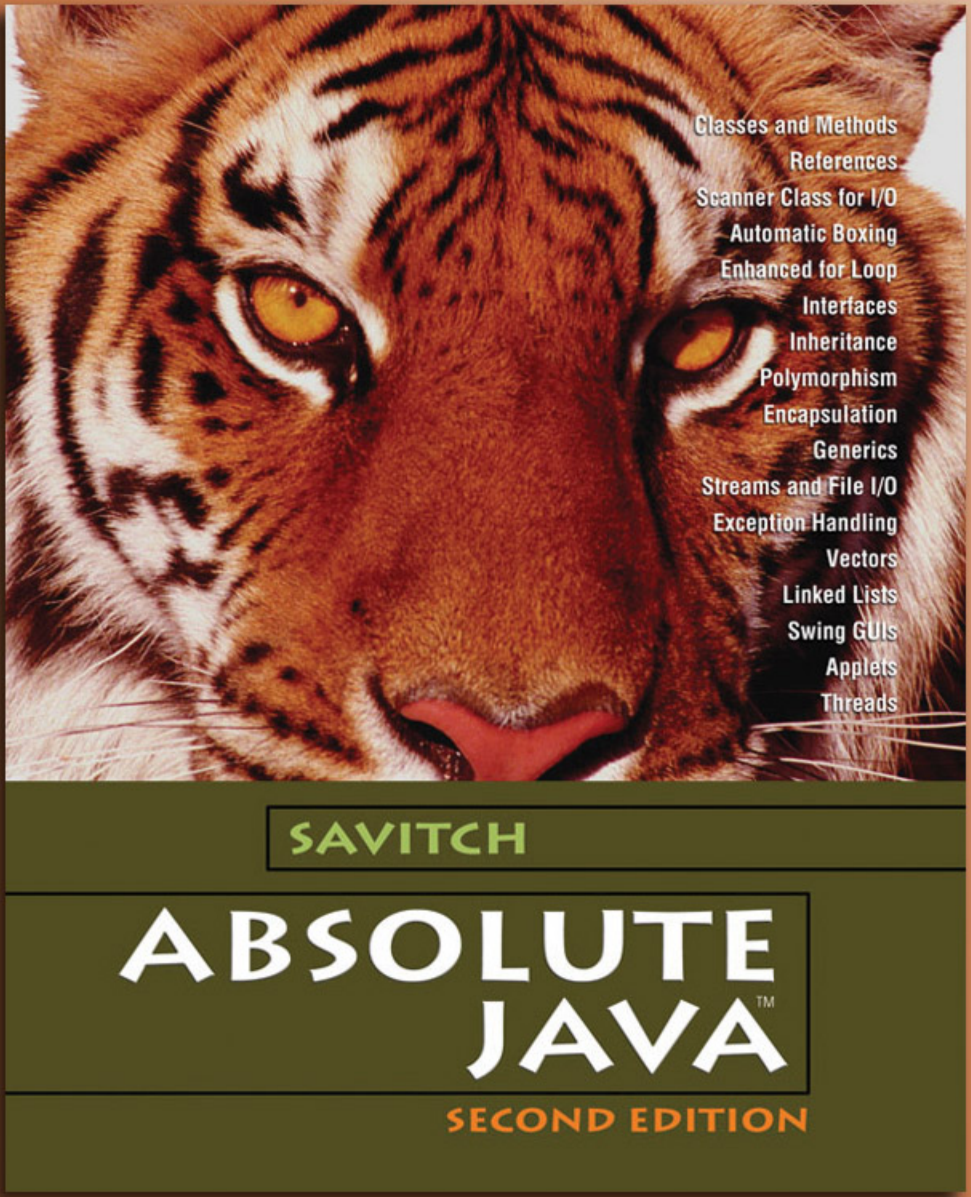

Chapter 8

Abstract Classes

Slides prepared by Rose Williams, *Binghamton University*

Copyright © 2006 Pearson Addison-Wesley. All rights reserved.

Introduction to Abstract Classes

- The `Employee` base class and two of its derived classes, `HourlyEmployee` and `SalariedEmployee` were defined

- The following method is added to the `Employee` class
  - It compares employees to to see if they have the same pay:

```
public boolean samePay(Employee other)
{
  return(this.getPay() == other.getPay());
}
```

© 2006 Pearson Addison-Wesley. All rights reserved

Introduction to Abstract Classes

- There are several problems with this method:
  - The `getPay` method is invoked in the `samePay` method
  - There are `getPay` methods in each of the derived classes
  - There is no `getPay` method in the `Employee` class, nor is there any way to define it reasonably without knowing whether the employee is hourly or salaried

© 2006 Pearson Addison-Wesley. All rights reserved

Introduction to Abstract Classes

- The ideal situation would be if there were a way to
  - Postpone the definition of a `getPay` method until the type of the employee were known (i.e., in the derived classes)
  - Leave some kind of note in the `Employee` class to indicate that it was accounted for
- Surprisingly, Java allows this using abstract classes and methods

© 2006 Pearson Addison-Wesley. All rights reserved

Introduction to Abstract Classes

- In order to postpone the definition of a method, Java allows an *abstract method* to be declared
  - An abstract method has a heading, but no method body
  - The body of the method is defined in the derived classes
- The class that contains an abstract method must be an *abstract class*

© 2006 Pearson Addison-Wesley. All rights reserved

Abstract Method

- An abstract method is like a placeholder for a method that will be fully defined in a descendent class
- It has a complete method heading, to which has been added the modifier **`abstract`**
- It cannot be private
- It has no method body, and ends with a semicolon in place of its body

```
public abstract double getPay();
public abstract void doIt(int count);
```

© 2006 Pearson Addison-Wesley. All rights reserved

Abstract Class

- A class that has at least one abstract method is called an *abstract class*
  - An abstract class must have the modifier **`abstract`** included in its class heading:

```
public abstract class Employee
{
  private instanceVariables;
  . . .
  public abstract double getPay();
  . . .
}
```

© 2006 Pearson Addison-Wesley. All rights reserved

Abstract Class

- An abstract class can have any number of abstract and/or fully defined methods

- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add `abstract` to its modifier

- If a class is not an abstract class then it is called a *concrete class*

© 2006 Pearson Addison-Wesley. All rights reserved

Pitfall: You Cannot Create Instances of an Abstract Class

- An abstract class can only be used to derive more specialized classes
  - While it may be useful to discuss employees in general, in reality an employee must be a salaried worker or an hourly worker
- An abstract class constructor cannot be used to create an object of the abstract class
  - However, a derived class constructor will include an invocation of the abstract class constructor in the form of `super`

© 2006 Pearson Addison-Wesley. All rights reserved

Tip: An Abstract Class Is a Type

- Although an object of an abstract class cannot be created, it is perfectly fine to have a parameter of an abstract class type
  - This makes it possible to plug in an object of any of its descendent classes

- It is also fine to use a variable of an abstract class type, as long as it names objects of its concrete descendent classes only

© 2006 Pearson Addison-Wesley. All rights reserved

Why have abstract classes?

- Suppose you wanted to create a class Shape, with subclasses Oval, Rectangle, Triangle, Hexagon, etc.
- You don’t want to allow creation of a “Shape”
  - Only *particular* shapes make sense, not *generic* ones
  - If Shape is abstract, you can’t create a new Shape
  - You *can* create a new Oval, a new Rectangle, etc.
- Abstract classes are good for defining a general category containing specific, “concrete” classes

Why have abstract methods?

- Suppose you have a class Shape, but it *isn't* abstract
  - Shape should *not* have a draw() method
  - Each subclass of Shape *should* have a draw() method
- Now suppose you have a variable Shape figure; where figure contains some subclass object (such as a Star)
  - It is *a syntax error* to say figure.draw(), because the Java compiler can't tell in advance what kind of value will be in the figure variable
  - A class "knows" its superclass, but doesn't know its subclasses
  - An object knows its class, but a class doesn't know its objects
- **Solution:** Give Shape an *abstract* method draw()
  - Now the class Shape is abstract, so it can't be instantiated
  - The figure variable cannot contain a (generic) Shape, because it is impossible to create one
  - Any object (such as a Star object) that *is* a (kind of) Shape *will* have the draw() method
  - The Java compiler can depend on figure.draw() being a legal call and does not give a syntax error

A problem

-
  ```
  class Shape { ... }
  ```

-
  ```
  class Star extends Shape {
      void draw() { ... }
      ...
  }
  ```

-
  ```
  class Crescent extends Shape {
      void draw() { ... }
      ...
  }
  ```
- `Shape someShape = new Star();`
  - This is legal, because a Star ***is*** a Shape
- `someShape.draw();`
  - This is a syntax error, because *some* Shape might not have a draw() method
  - Remember: ***A class knows its superclass, but not its subclasses***

A solution

-
  ```
  abstract class Shape {
      void draw();
  }
  ```

-
  ```
  class Star extends Shape {
      void draw() { ... }
      ...
  }
  ```

-
  ```
  class Crescent extends Shape {
      void draw() { ... }
      ...
  }
  ```
- `Shape someShape = new Star();`
  - This is legal, because a Star ***is*** a Shape
  - However, `Shape someShape = new Shape();` is *no longer* legal
- `someShape.draw();`
  - This is legal, because every actual instance *must* have a `draw()` method