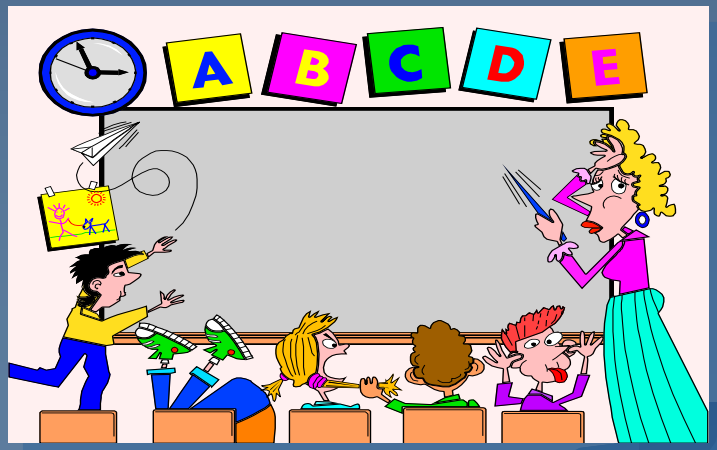
Object-Oriented Programming: Polymorphism

Chapter 10

What You Will Learn

- What is polymorphism?

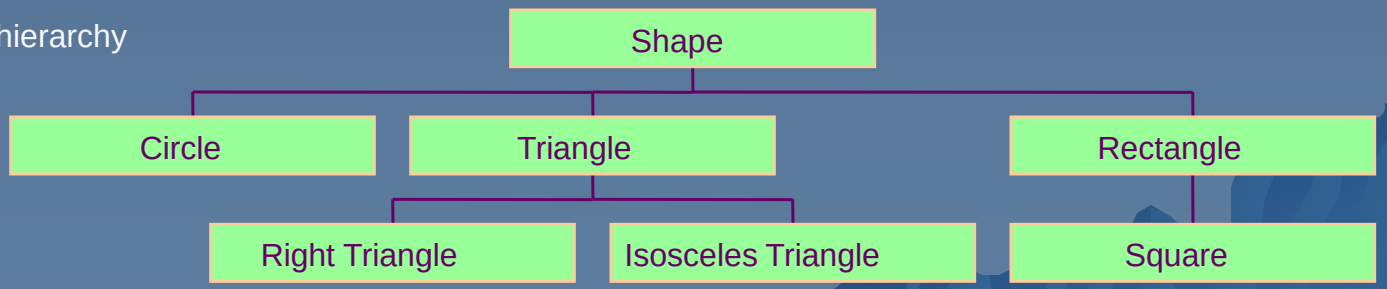
- How to declare and use **`virtual`** functions for abstract classes



Problem with Subclasses

- Given the class hierarchy below
- Consider the existence of a draw function for each subclass
- Consider also an array of references to the superclass (which can also point to various objects of the subclasses)
- How do you specify which draw statement to be called when using the references

Shape class hierarchy



Introduction

- Polymorphism
  - Enables "programming in the general"
  - The same invocation can produce "many forms" of results
- Interfaces
  - Implemented by classes to assign common functionality to possibly unrelated classes

Polymorphism

- When a program invokes a method through a superclass variable,
  - the correct subclass version of the method is called,
  - based on the type of the reference stored in the superclass variable
- The same method name and signature can cause different actions to occur,
  - depending on the type of object on which the method is invoked

Polymorphism

- Polymorphism enables programmers to deal in generalities and
  - let the execution-time environment handle the specifics.
- Programmers can command objects to behave in manners appropriate to those objects,
  - without knowing the types of the objects
  - (as long as the objects belong to the same inheritance hierarchy).

Polymorphism Promotes Extensibility

- Software that invokes polymorphic behavior
  - independent of the object types to which messages are sent.
- New object types that can respond to existing method calls can be
  - incorporated into a system without requiring modification of the base system.
  - Only client code that instantiates new objects must be modified to accommodate new types.

Demonstrating Polymorphic Behavior

- A superclass reference can be aimed at a subclass object
  - a subclass object "*is-a*" superclass object
  - the type of the actual referenced object, not the type of the reference, determines which method is called
- A subclass reference can be aimed at a superclass object only if the object is downcasted
- View example, Figure 10.1

Polymorphism

- Promotes extensibility
- New objects types can respond to existing method calls
  - Can be incorporated into a system without modifying base system
- Only client code that instantiates the new objects must be modified
  - To accommodate new types

Abstract Classes and Methods

- Abstract classes
  - Are superclasses (called abstract superclasses)
  - Cannot be instantiated
  - Incomplete
    - subclasses fill in "missing pieces"
- Concrete classes
  - Can be instantiated
  - Implement every method they declare
  - Provide specifics

Abstract Classes and Methods

- Purpose of an abstract class
  - Declare common attributes …
  - Declare common behaviors of classes in a class hierarchy
- Contains one or more abstract methods
  - Subclasses must override
- Instance variables, concrete methods of abstract class
  - subject to normal rules of inheritance

Abstract Classes

- Classes that are too general to create real objects
- Used only as abstract superclasses for concrete subclasses and to declare reference variables
- Many inheritance hierarchies have abstract superclasses occupying the top few levels

Keyword `abstract`

- Use to declare a class `abstract`
- Also use to declare a method `abstract`
- Abstract classes normally contain one or more abstract methods
- All concrete subclasses must override all inherited abstract methods

Abstract Classes and Methods

- `Iterator` class
  - Traverses all the objects in a collection, such as an array
  - Often used in polymorphic programming to traverse a collection that contains references to objects from various levels of a hierarchy

Abstract Classes

- Declares common attributes and behaviors of the various classes in a class hierarchy.
- Typically contains one or more abstract methods
  - Subclasses must override if the subclasses are to be concrete.
- Instance variables and concrete methods of an abstract class subject to the normal rules of inheritance.

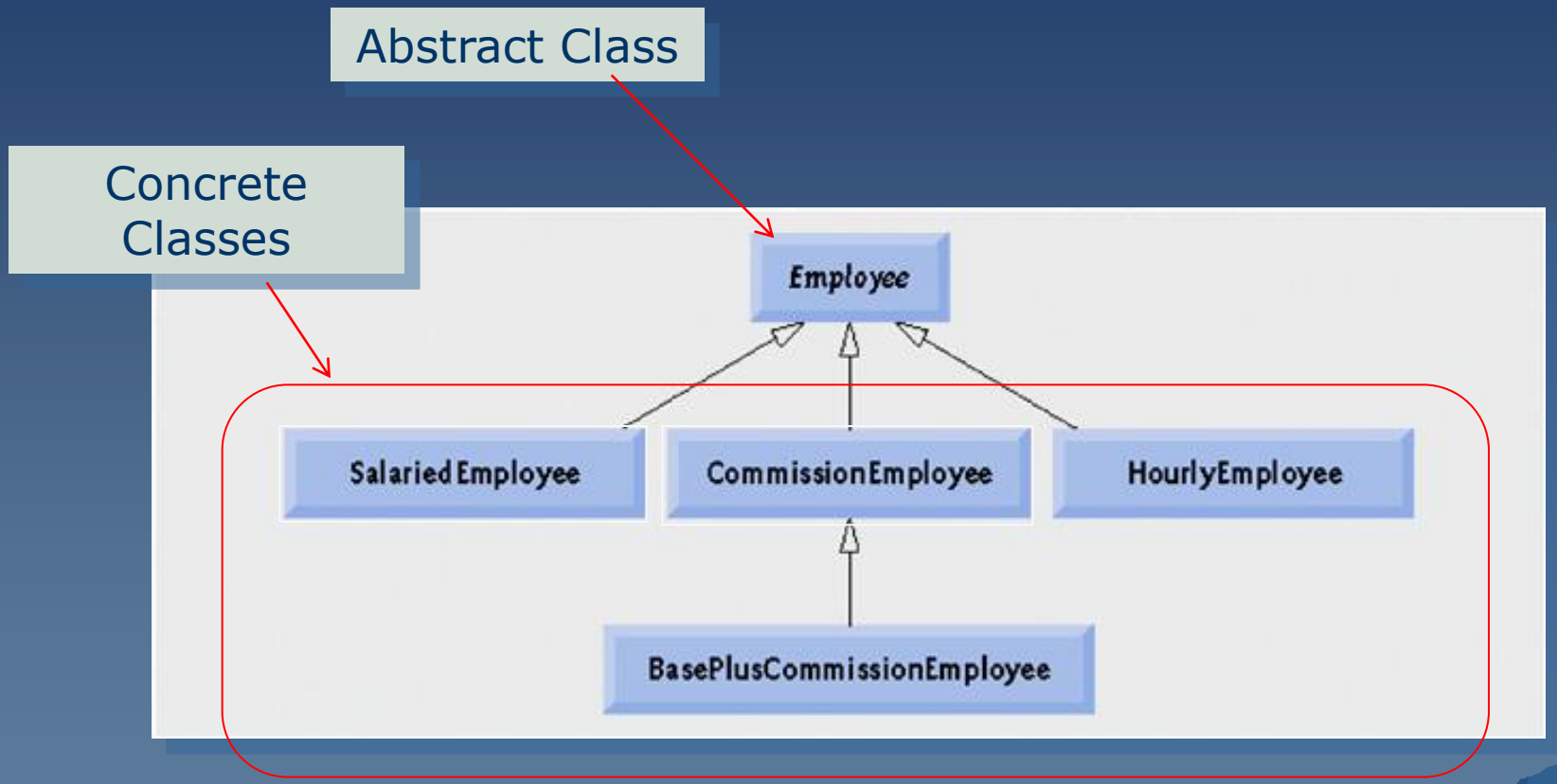
Beware! Compile Time Errors

- Attempting to instantiate an object of an abstract class
- Failure to implement a superclass's abstract methods in a subclass
  - unless the subclass is also declared `abstract`.

Creating Abstract Superclass `Employee`

- `abstract` superclass `Employee`, Figure 10.4
  - `earnings` is declared `abstract`
    - No implementation can be given for `earnings` in the `Employee` `abstract` class
  - An array of `Employee` variables will store references to subclass objects
    - `earnings` method calls from these variables will call the appropriate version of the `earnings` method

Example Based on Employee



Click on Classes to see source code

Polymorphic interface for the Employee hierarchy classes.

	earnings	toString
Employee	abstract	*firstName lastName* social security number: *SSN*
Salaried-Employee	weeklySalary	salaried employee: *firstName lastName* social security number: *SSN* weekly salary: *weeklysalary*
Hourly-Employee	*If hours <= 40* wage * hours *If hours > 40* 40 * wage + (hours - 40) * wage * 1.5	hourly employee: *firstName lastName* social security number: *SSN* hourly wage: *wage*; hours worked: *hours*
Commission-Employee	commissionRate * grossSales	commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*
BasePlus-Commission-Employee	(commissionRate * grossSales) + baseSalary	base salaried commission employee: *firstName lastName* social security number: *SSN* gross sales: *grossSales*; commission rate: *commissionRate*; base salary: *baseSalary*

Note in Example Hierarchy

- Dynamic binding
  - Also known as late binding
  - Calls to overridden methods are resolved at execution time, based on the type of object referenced
- `instanceof` operator
  - Determines whether an object is an instance of a certain type

How Do They Do That?

- How does it work?
  - Access a derived object via base class pointer
  - Invoke an abstract method
  - At run time the correct version of the method is used
- Design of the V-Table
  - Note description from C++

Note in Example Hierarchy

- Downcasting
  - Convert a reference to a superclass to a reference to a subclass
  - Allowed only if the object has an *is-a* relationship with the subclass
- `getClass` method
  - Inherited from `Object`
  - Returns an object of type `Class`
- `getName` method of class `Class`
  - Returns the class's name

Superclass And Subclass Assignment Rules

- Assigning a superclass reference to superclass variable straightforward
- Subclass reference to subclass variable straightforward
- Subclass reference to superclass variable safe
  - because of *is-a* relationship
  - Referring to subclass-only members through superclass variables a compilation error
- Superclass reference to a subclass variable a compilation error
  - Downcasting can get around this error

`final` Methods and Classes

- `final` methods
  - Cannot be overridden in a subclass
  - `private` and `static` methods implicitly `final`
  - `final` methods are resolved at compile time, this is known as static binding
    - Compilers can optimize by inlining the code
- `final` classes
  - Cannot be extended by a subclass
  - All methods in a `final` class implicitly `final`

Why Use Interfaces

- Java has single inheritance, only
- This means that a child class inherits from only one parent class
- Sometimes multiple inheritance would be convenient
- *Interfaces* give Java some of the advantages of multiple inheritance without incurring the disadvantages

Why Use Interfaces

- Provide capability for unrelated classes to implement a set of common methods
- Define and standardize ways people and systems can interact
- Interface specifies what operations must be permitted
- Does not specify how performed

What is an Interface?

- An interface is a collection of constants and method declarations
- An interface describes a set of methods that can be called on an object
- The method declarations do not include an implementation
  - there is no method body

What is an Interface?

- A child class that *extends* a parent class can also *implement* an interface to gain some additional behavior
- Implementing an interface is a "promise" to include the specified method(s)
- A method in an interface cannot be made *private*

When A Class Definition *Implements* An Interface:

- It must implement each method in the interface
- Each method must be *public* (even though the interface might not say so)
- Constants from the interface can be used as if they had been defined in the class (They should not be re-defined in the class)

Declaring Constants with Interfaces

- Interfaces can be used to declare constants used in many class declarations
  - These constants are implicitly `public`, `static` and `final`
  - Using a `static import` declaration allows clients to use these constants with just their names

Implementation vs. Interface Inheritance

Implementation Inheritance

- Functionality high in the hierarchy
- Each new subclass inherits one or more methods declared in superclass
- Subclass uses superclass declarations

Interface Inheritance

- Functionality lower in hierarchy
- Superclass specifies one or more abstract methods
- Must be declared for each class in hierarchy
- Overridden for subclass-specific implementations

Creating and Using Interfaces

- Declaration begins with **`interface`** keyword
- Classes **`implement`** an interface (and its methods)
- Contains `public abstract` methods
  - Classes (that `implement` the interface) must implement these methods

Creating and Using Interfaces

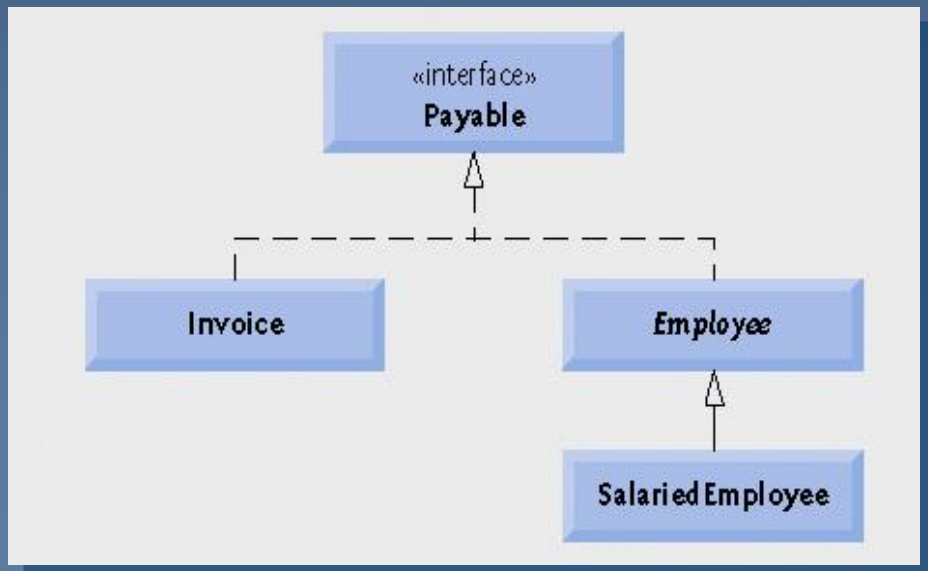
- Consider the possibility of having a class which manipulates mathematical functions
- You want to send a function as a parameter
  - Note that C++ allows this directly
  - Java does not
- This task can be accomplished with interfaces

Creating and Using Interfaces

- Declare interface `Function`
- Declare class `MyFunction` which implements `Function`
- Note other functions which are subclass objects of `MyFunction`
- View test program which passes `Function` subclass objects to function manipulation methods

Case Study: A `Payable` Hierarchy

- `Payable` interface
  - Contains method `getPaymentAmount`
  - Is implemented by the `Invoice` and `Employee` classes



Click to view the test program

End of Chapter 10 Lecture

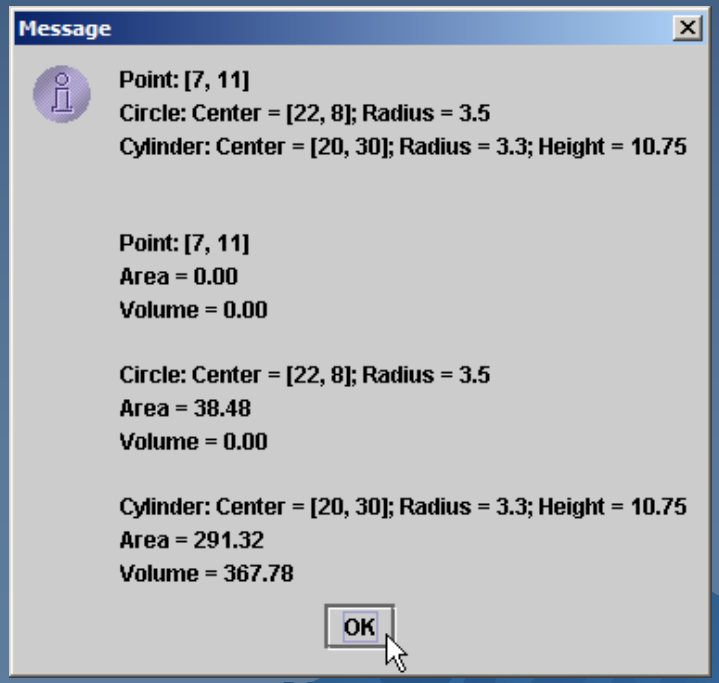
Following slides are from previous edition. Links may not work.

Source Code for **`Shape`** Superclass Hierarchy

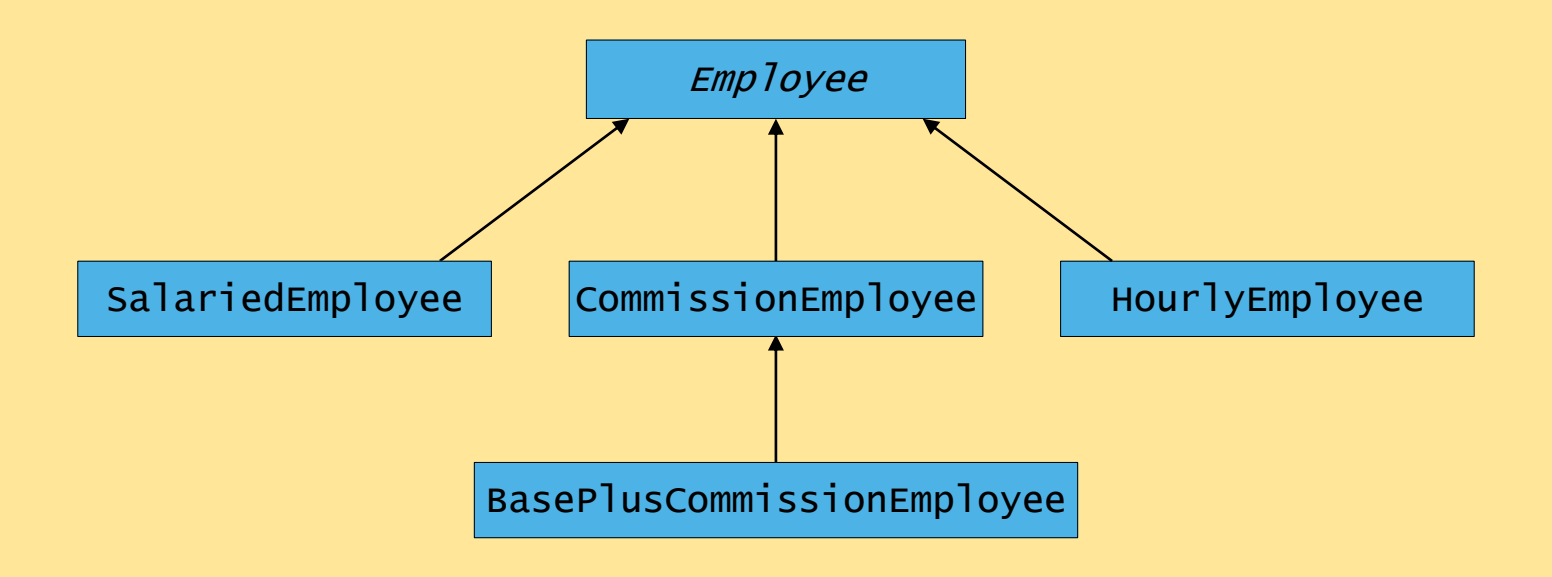
- Shape superclass, Figure 10.6
  - Note abstract method, **`getName`**
- Point subclass, Figure 10.7
  - Note, extends **`Shape`**, override of **`getName`**
- Circle subclass, Figure 10.8
  - Note extends **`Point`**, override of **`getArea`**, **`getName`**, and **`toString`**
- Cylinder subclass, Figure 10.9
  - Note extends Circle, override of **`getArea`**, **`getName`**, and **`toString`**

Source Code for `Shape` Superclass Hierarchy

- Driver program to demonstrate, Figure 10.10
  - Note array of superclass references
- Output of program



Polymorphic Payroll System



Class hierarchy for polymorphic payroll application

Polymorphic Payroll System

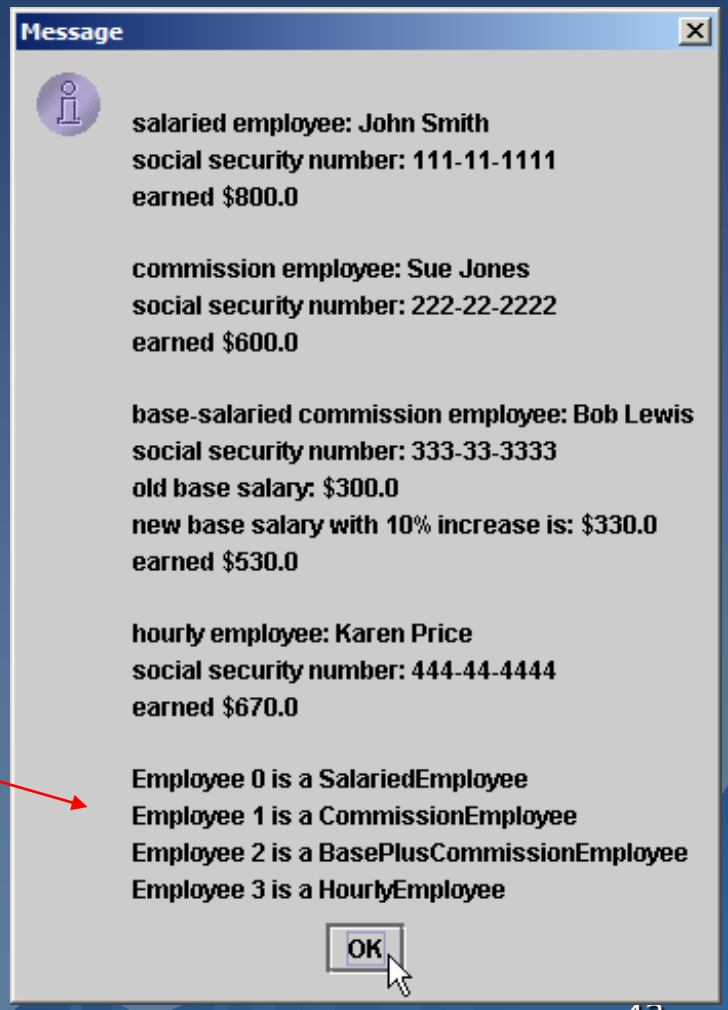
- Abstract superclass, **`Employee`**, Figure 10.12
  - Note abstract class specification, abstract **`earnings`** method
- Abstract subclass, **`SalariedEmployee`**, Figure 10.13
  - Note extends **`Employee`**, override of **`earnings`** method
- Look at other subclasses in text, pg 461 …
  - Note here also the override of **`earnings`**

Polymorphic Payroll System

- View test program, Figure 10.17
  - Note array of superclass references which are pointed at various subclass objects
  - Note generic calls of `toString` method
  - Note use of `instantceof` operator
  - Consider use of downcasting (line 37)
  - Note use of `getClass().getName()` method
    - Gives access to the name of the class

Polymorphic Payroll System

- Output of the payroll testing program
- Note results of `getClass().getName()` calls



Creating and Using Interfaces

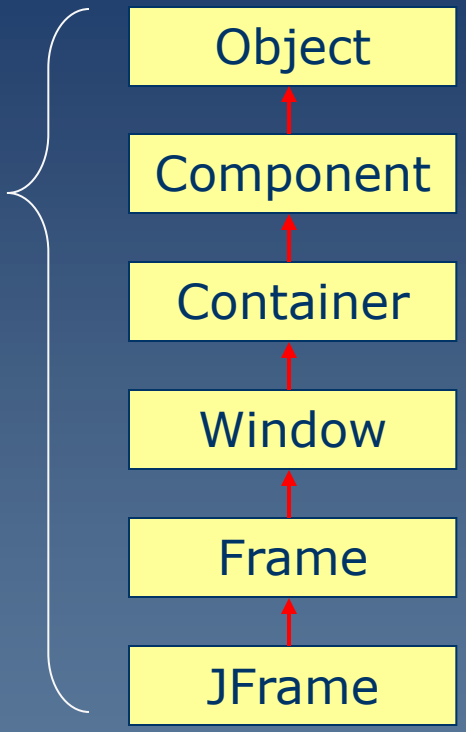
- View implementation of the interface Shape, Figure 10.19
  Note the following:
  - Line 4
    ```
    public class Point extends Object
          implements Shape { …
    ```
  - Declaration of additional methods
  - Declaration of abstract methods **`getArea`**, **`getVolume`**, and **`getName`**

Nested Classes

- Top-level classes
  - Not declared inside a class or a method
- Nested classes
  - Declared inside other classes
  - Inner classes
    - Non-static nested classes
- Demonstrated in Figure 10.22
  - Run the program
  - Audio

Mentioned In The Audio

- Inheritance Hierarchy
- Panes of a **`JFrame`**
  - Background
  - Content
  - Glass



Mentioned In The Audio

- Three things required when performing events
  1. Implement the interface
  2. Register the event handlers
  3. Specifically implement the **`actionPerformed`** methods

Nested Classes

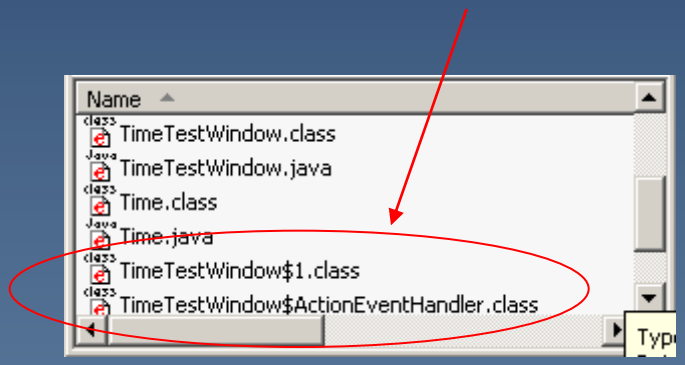
- An inner class is allowed to directly access its inner class's variables and methods
- When `this` used in an inner class
  - Refers to current inner-class object
- To access outer-class using `this`
  - Precede `this` with outer-class name

Nested Classes

- Anonymous inner class
  - Declared inside a method of a class
  - Has no name
- Inner class declared inside a method
  - Can access outer class's members
  - Limited to local variables of method in which declared
- Note use of anonymous inner class, Figure 10.23

Notes on Nested Classes

- Compiling class that contains nested class
  - Results in separate `.class` file
  - **`OuterClassName$InnerClassName.class`**



- Inner classes with names can be declared as
  - `public`, `protected`, `private` or package access

Notes on Nested Classes

- Access outer class's **this** reference

  ***OuterClassName*.this**

- Outer class is responsible for creating inner class objects

```
OuterClassName.InnerClassName innerRef =
        ref.new InnerClassName();
```

- Nested classes can be declared **static**
  - Object of outer class need not be created
  - Static nested class does not have access to outer class's non-static members

Type-Wrapper Classes for Primitive Types

- Each primitive type has one
  - **`Character`**, **`Byte`**, **`Integer`**, **`Boolean`**, etc.
- Enable to represent primitive as `Object`
  - Primitive values can be processed polymorphically using wrapper classes
- Declared as **`final`**
- Many methods are declared **`static`**

Invoking Superclass Methods from Subclass Objects

- Recall the point and circle hierarchy
  - Note :
    Assignment of superclass reference to superclass variable
  - Assignment of subclass reference to subclass variable
- No surprises … but the "is-a" relationship makes possible
  - Assignment of subclass reference to superclass variable
- See Figure 10.1

Using Superclass References with Subclass-Type Variables

- Suppose we have a superclass object

```
Point3 point = new Point3 (30, 40);
```

- Then a subclass reference

```
Circle4 circle;
```

- It would be illegal to have the subclass reference aimed at the superclass object

```
circle = point;
```

Subclass Method Calls via Superclass-Type Variables

- Consider aiming a subclass reference at a superclass object
  - If you use this to access a subclass-only method it is illegal
- Note Figure 10.3
  - Lines 22 - 23

Summary of Legal Assignments between Super- and Subclass Variables

- Assigning superclass reference to superclass-type variable OK
- Assigning subclass reference to subclass-type variable OK
- Assigning subclass object's reference to superclass type-variable, safe
  - A circle "is a" point
  - Only possible to invoke superclass methods
- Do NOT assign superclass reference to subclass-type variable
  - You can cast the superclass reference to a subclass type (explicitly)

Polymorphism Examples

Suppose designing video game

- Superclass **`SpaceObject`**
  - Subclasses **`Martian, SpaceShip, LaserBeam`**
  - Contains method **`draw`**
- To refresh screen
  - Send **`draw`** message to each object
  - Same message has "many forms" of results
- Easy to add class **`Mercurian`**
  - Extends **`SpaceObject`**
  - Provides its own implementation of **`draw`**
- Programmer does not need to change code
  - Calls **`draw`** regardless of object's type
  - **`Mercurian`** objects "plug right in"

Polymorphism

- Enables programmers to deal in generalities
  - Let execution-time environment handle specifics
- Able to command objects to behave in manners appropriate to those objects
  - Don't need to know type of the object
  - Objects need only to belong to same inheritance hierarchy

Abstract Classes and Methods

- Abstract classes not required, but reduce client code dependencies
- To make a class abstract
  - Declare with keyword **`abstract`**
  - Contain one or more *abstract methods*

```
public abstract void draw();
```

  - Abstract methods
    - No implementation, must be overridden

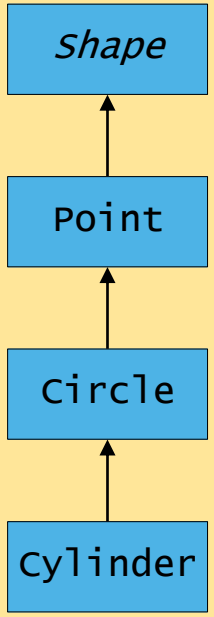
Inheriting Interface and Implementation

Make abstract superclass **`Shape`**

- Abstract method (must be implemented)
  - **`getName`**, **`print`**
  - Default implementation does not make sense
- Methods may be overridden
  - **`getArea`**, **`getVolume`**
    - Default implementations return `0.0`
  - If not overridden, uses superclass default implementation
- Subclasses **`Point`**, **`Circle`**, **`Cylinder`**

Polymorphic Interface For The Shape Hierarchy Class

	getArea	getVolume	getName	toString
Shape	0.0	0.0	abstract	default Object implement
Point	0.0	0.0	"Point"	[x,y]
Circle	πr^2	0.0	"Circle"	center=[x,y]; radius=r
Cylinder	$2\pi r^2 + 2\pi rh$	$\pi r^2 h$	"Cylinder"	center=[x,y]; radius=r; height=h



Creating and Using Interfaces

- All the same "is-a" relationships apply when an interface inheritance is used
- Objects of any class that extend the interface are also objects of the interface type
  - **`Circle`** extends **`Point`**, thus it is-a **`Shape`**
- Multiple interfaces are possible
  - Comma-separated list used in declaration