

§9.10-9.22: Polymorphism

7 March 2007

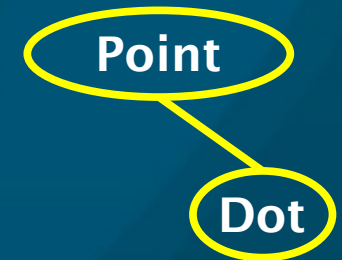
CMPT167

Dr. Sean Ho

Trinity Western University

Review last time

- **Inheritance** for software reusability
 - “has a” vs. “is a kind of”
- Subclass/superclass **constructors**
 - `super()`
- Subclass/superclass **references**
 - **Downcasting**



What's on for today

■ Polymorphism

- Dynamic method **binding**
- **final** keyword for **classes** and **methods**
- **Abstract** and **concrete** classes
 - ◆ **abstract** keyword for classes and methods

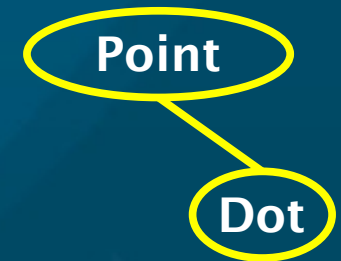
■ Interfaces

- vs. abstract superclasses

■ **Type-wrapper** classes for the primitive types

Polymorphism

- Think carefully about **class hierarchy** in program design
- Write programs/**algorithms** to operate on **superclass** objects
 - As **generic** as possible
- Instances of **subclasses** can be operated on by the algorithms without need for modification
- **Dynamic method binding**:
 - Java chooses correct method (e.g., **toString()**) from subclass



final: methods/classes

- We've seen **final** on **variables**: set as **constant**
- **final** on a **method** prevents subclasses from **overriding**
- **final** on a **class** means it cannot be extended
 - (Other classes cannot **inherit** from it)

Abstract vs. concrete classes

■ Abstract classes:

- Too **generic** to define a real object
 - ◆ e.g., **TwoDimensionalShape**
- **Not** intended to be directly instantiated
 - ◆ Java can **enforce** this: use **abstract** keyword
 - ◆ **abstract** classes can have **abstract methods**:
 - No **body** defined; each **subclass** must implement

■ Concrete classes:

- **Subclass** of an abstract class, meant to be instantiated
 - ◆ e.g., **Square, Circle, Triangle**

Example: TwoDimensionalShape

- Abstract superclass: TwoDimensionalShape
 - Abstract method: draw()
- Concrete subclasses: Circle, Square, Triangle
 - Each provide own implementation of draw()

```
abstract public class TwoDimensionalShape {  
    abstract public void draw();    // no body
```

```
public class Circle extends TwoDimensionalShape {  
    public void draw() { drawOval( x, y, r, r ); }  
}
```

```
public class Square extends TwoDimensionalShape {  
    public void draw() { drawRect( x, y, w, h ); }  
}
```

Interfaces

- Define a **set** of abstract methods

```
public interface drawableShape {  
    public abstract void draw();  
    public abstract double area();  
}
```

- Classes **implement** these methods

```
public class Circle implements drawableShape {  
    public void draw() { drawOval( x, y, r, r ); }  
    public double area() { return 2 * Math.PI * r * r; }  
}
```

- We've already been using the **actionListener** interface

Abstract classes vs. interfaces

- Abstract **superclasses** declare **identity**:
 - “**Circle** is a kind of **TwoDimensionalShape**”
 - Each class can have only **one** superclass
 - ◆ No **multiple inheritance** in Java
 - Inherit methods, attributes; get **protected** access
- **Interfaces** declare **capability**:
 - “**Circles know how** to be **drawableShapes**”
 - May implement **multiple** interfaces
 - Interfaces are not **ADTs** (abstract data types)

Primitive type-wrapper classes

- Eight **primitive** types in Java
 - Primitives are **not** really **objects**
- **Type-wrapper** classes for each of the eight:
 - **Character**, **Byte**, **Integer**, **Boolean**, etc.
 - Enable us to **represent** primitives as **Object**
 - Can then process them **polymorphically**
- Type-wrapper classes declared **final**
 - Many methods declared **static**
 - ◆ e.g., **Integer.parseInt(String)**

TODO

- Lab4 due next week Wed 14Mar
 - OO concepts (sets and vectors)