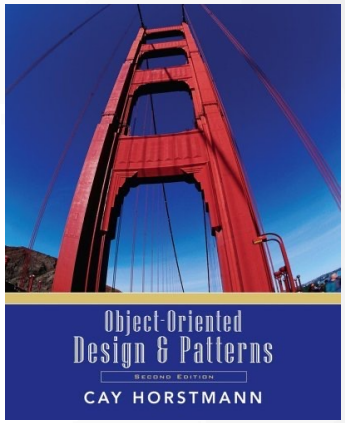
Object-Oriented Design & Patterns

2nd edition

Cay S. Horstmann



Chapter 6: Inheritance and Abstract Classes

CPSC 2100

Software Design and Development

Chapter Topics

- The Concept of Inheritance.
- Graphics Programming with Inheritance.
- Abstract Classes.
- The TEMPLATE METHOD Pattern.
- Protected Interfaces.
- The Hierarchy of Swing Components.
- The Hierarchy of Standard Geometrical Shapes.
- The Hierarchy of Exception Classes.
- When Not to Use Inheritance.

Chapter Objective

- Discuss the important class relationship of *inheritance*.
- Examine how inheritance is used in Java class libraries.

Inheritance

- Used to model relationship between classes.
  - One class represents a more general concept (Superclass)
  - Another class represents a more specialized concept (Subclass).

Modeling Specialization

- Start with simple Employee class

```
public class Employee
   {
      public Employee(String aName) { name = aName; }
      public void setSalary(double aSalary) { salary =
   aSalary; }
      public String getName() { return name; }
      public double getSalary() { return salary; }

      private String name;
      private double salary;
   }
```

- Manager is a subclass

Modeling Specialization

- Manager class adds new method: setBonus
- Manager class *overrides* existing method: getSalary
  - Adds salary and bonus

```
public class Manager extends Employee
   {
      public Manager(String aName) { ... }

    // new method
      public void setBonus(double aBonus)
    {
       bonus = aBonus;
    }

    // overrides Employee method
      public double getSalary() { ... }

      private double bonus; // new field
   }
```

Modeling Specialization

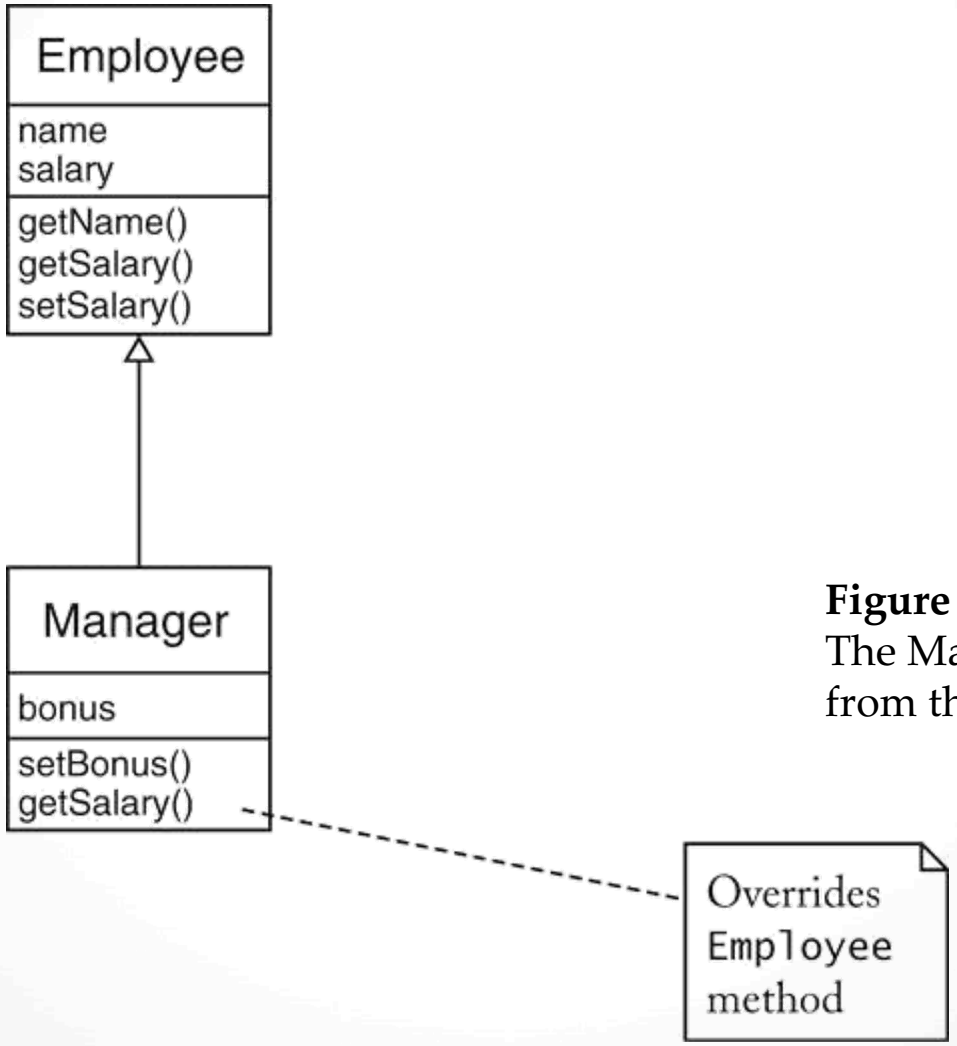


Figure 1:
The Manager Class Inherits from the Employee Class

Manager Methods and Fields

- methods `setSalary`, `getname` (inherited from Employee).
- method `getSalary` (overridden in Manager).
- method `setBonus` (defined in Manager).
- fields `name` and `salary` (defined in Employee).
- field `bonus` (defined in Manager).

The Super/Sub Terminology

- Why is Manager a **subclass**?
- Isn't a Manager **superior**?
- Doesn't a Manager object have more fields?
- The set of managers is a *subset* of the set of employees

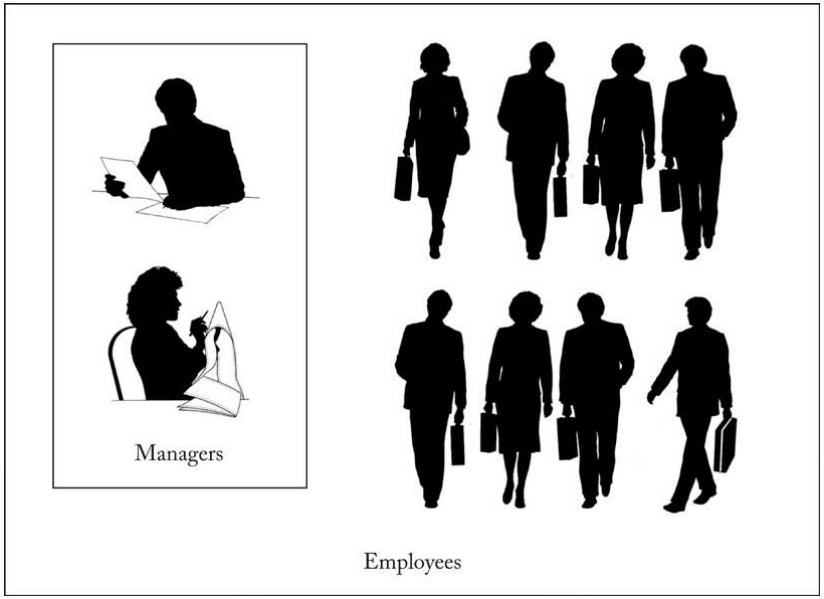


Figure 2: The Set of Managers is a Subset of the Set of Employee

Inheritance Hierarchies

- Real world: Hierarchies describe **general/specific relationships:**
  - **General** concept at root of tree.
  - More **specific** concepts are children.

- Programming: Inheritance hierarchy
  - **General** **superclass** at root of tree.
  - More **specific** **subclasses** are children.

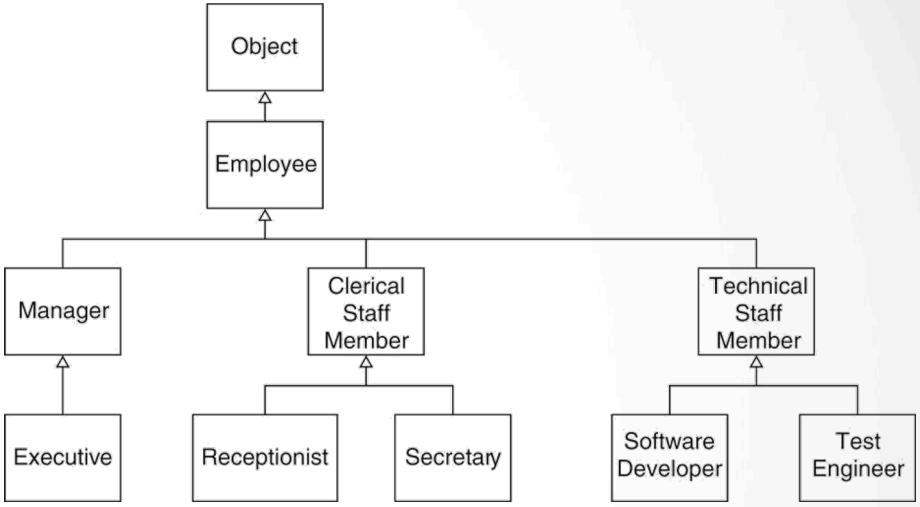


Figure 3:
A Hierarchy of Employee Classes

Liskov Substitution Principle

- Formulated by **Barbara Liskov.**
  - *You can use a subclass object whenever a superclass object is expected.*

- Example:

```
Employee e;
     ...
System.out.println("name=" + e.getName());
System.out.println("salary=" + e.getSalary());
```

- Can set `e` to `Manager` reference.

```
e = new Manager("Barbara");
e.getName();
e.getSalary(); // JVM Polymorphism
```

- **Polymorphism**: Correct `getSalary` method is invoked.

Invoking Superclass Methods

- Can't access private fields of superclass.

```
public class Manager extends Employee
{
   public double getSalary()
   {
      return salary + bonus;  // ERROR--private
field
   }
   ...
}
```

- Be careful when calling superclass method.

```
public double getSalary()
   {
      return getSalary() + bonus; //ERROR--recursive call
   }
```

Invoking Superclass Methods

- Use super keyword

```
public double getSalary()
{
   return super.getSalary() + bonus;
}
```

- super is *not* a reference.
- super turns off polymorphic call mechanism

⇩

enforces the superclass method to be called.

Invoking Superclass Constructors

- Use super keyword in subclass constructor:

```
public Manager(String aName)
{
   super(aName); //calls super constructor
   bonus = 0;
}
```

- Call to super must be *first* statement in subclass constructor.

- If subclass constructor doesn't call super, superclass must have constructor without parameters.

Preconditions

- Precondition of redefined method *at most as strong*

```
public class Employee
{
   /**
      Sets the employee salary to a given value.
      @param aSalary the new salary
      @precondition aSalary > 0
   */
    public void setSalary(double aSalary) { ... }
}
```

- Can we redefine `Manager.setSalary` with precondition salary > 100000?

- No--Could be defeated:

```
Manager m = new Manager();
Employee e = m;
e.setSalary(50000);
```

Postconditions, Visibility, Exceptions

- Postcondition of redefined method *at least as strong*.
- Example: `Employee.setSalary` promises not to decrease salary.
  - Then `Manager.setSalary` must fulfill postcondition.
- Redefined method cannot be more private.
  (Common error: omit public when redefining)
- Redefined method cannot throw more checked exceptions than are already declared in the superclass method.

Graphic Programming with Inheritance

- Chapter 4: Create drawings by implementing Icon interface type.
- Now: Form subclass of JComponent

```
public class MyComponent extends JComponent
{
   public void paintComponent(Graphics g)
   {
      drawing instructions go here
   }
   ...
}
```

- Advantage: Inherit behavior from JComponent.
- Example: Can attach mouse listener to JComponent.

Mouse Listeners

- Attach mouse listener to component.
- Can listen to mouse events (clicks) or mouse motion events.

```
public interface MouseListener
{
   void mouseClicked(MouseEvent event);
   void mousePressed(MouseEvent event);
   void mouseReleased(MouseEvent event);
   void mouseEntered(MouseEvent event);
   void mouseExited(MouseEvent event);
}

public interface MouseMotionListener
{
   void mouseMoved(MouseEvent event);
   void mouseDragged(MouseEvent event);
}
```

Mouse Adapters

- To simplify the implementation of listeners
  - ⇨ MouseAdaptor.
  - ⇨ MouseMotionAdaptor.

- What if you just want to listen to mousePressed?

```
public class MouseAdapter implements MouseListener
{
   public void mouseClicked(MouseEvent event) {}
   public void mousePressed(MouseEvent event) {}
   public void mouseReleased(MouseEvent event) {}
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
}
```

Mouse Adapters

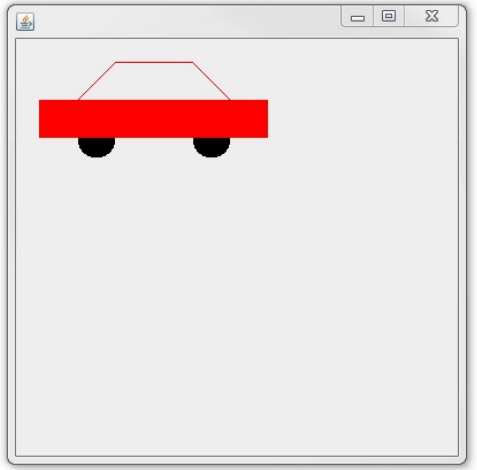
- Component constructor adds listener:

```
addMouseListener(new MouseAdapter()
   {
      public void mousePressed(MouseEvent event)
      {
         mouse action goes here
      }
   });
```

Car Mover Program

- Use the mouse to drag a car shape.
- Car panel has mouse + mouse motion listeners.
- mousePressed remembers point of mouse press.
- mouseDragged translates car shape.

CarComponent.java
CarMover.java
CarShape.java



Car Mover Program

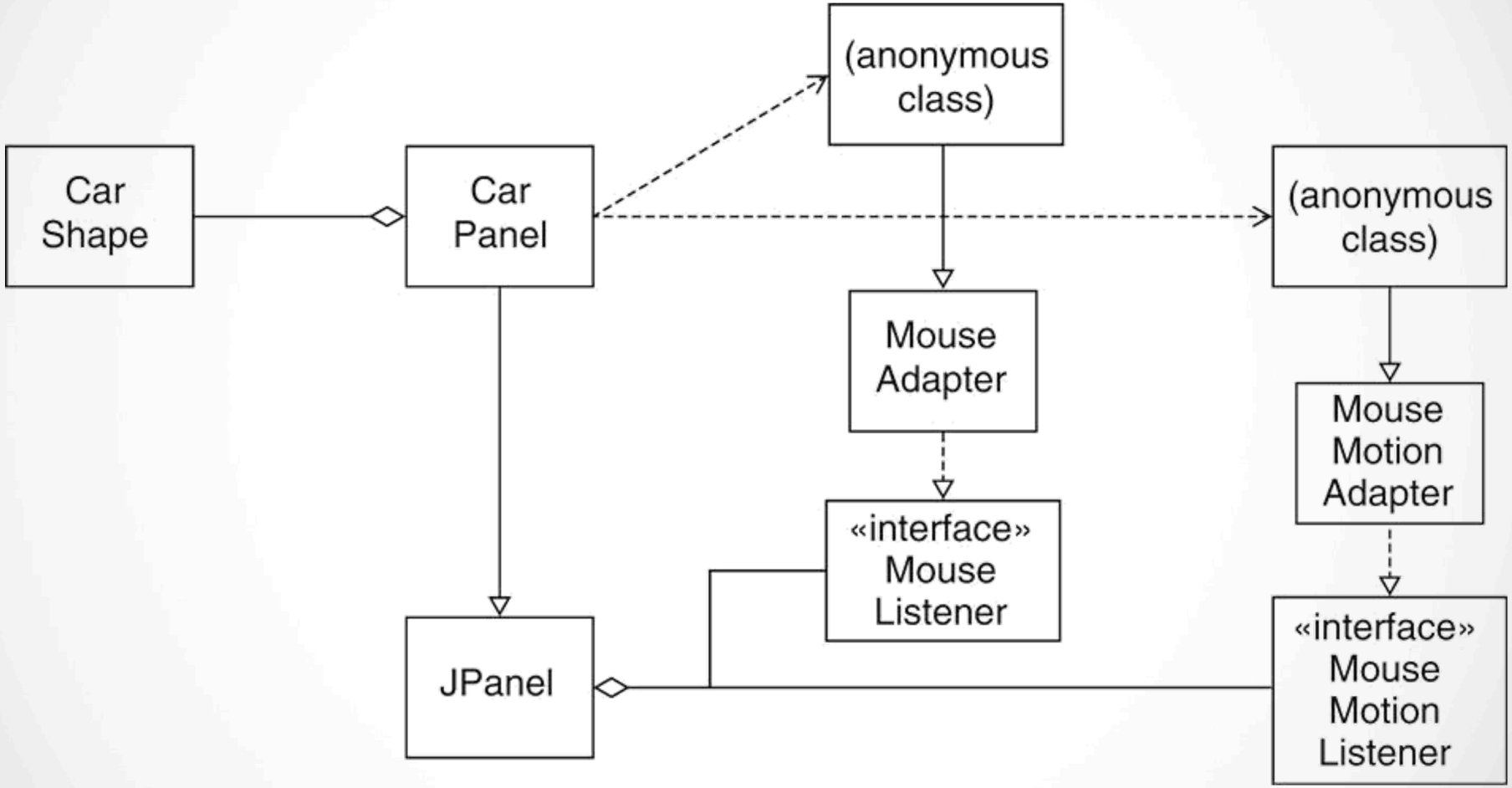


Figure 4:
The Classes of the Car Mover Program

Scene Editor

- Draws various shapes.
- User can add, delete, move shapes.
- User *selects* shape with mouse.
  - Selected shape is highlighted (filled in).

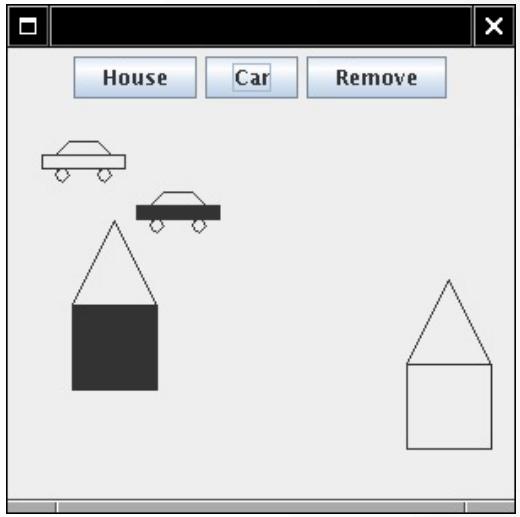


Figure 5:
The Scene Editor

The SceneShape Interface Type

- keep track of selection state.
- draw plain or selected shape.
- move shape.
- *testing*: is a point (e.g. mouse position) inside?

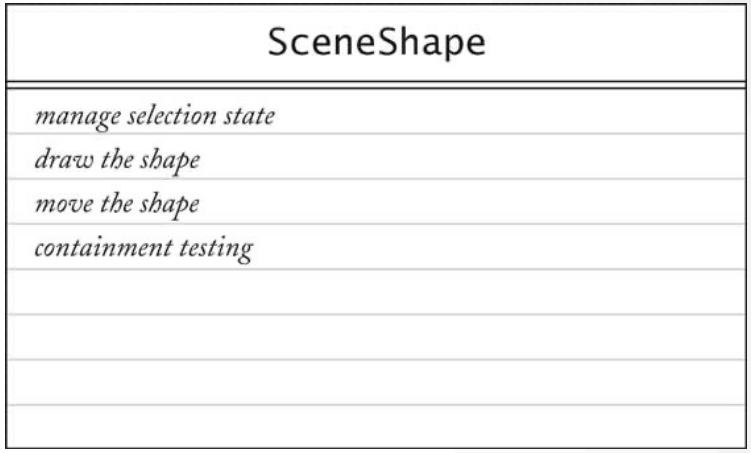


Figure 6:
A CRC Card of the SceneShape Interface Type

The SceneShape Interface Type

```
public interface SceneShape
{
   void setSelected(boolean b);
   boolean isSelected();
   void draw(Graphics2D g2);
   void drawSelection(Graphics2D g2);
   void translate(int dx, int dy);
   boolean contains(Point2D aPoint);
}
```

CarShape and HouseShape Classes

```
public class CarShape implements SceneShape
{
   ...
   public void setSelected(boolean b) { selected = b; }
   public boolean isSelected() { return selected; }
   private boolean selected;
}

public class HouseShape implements SceneShape
{
   ...
   public void setSelected(boolean b) { selected = b; }
   public boolean isSelected() { return selected; }
   private boolean selected;
}
```

Abstract Classes

- It is better idea to design a class that expresses this commonality.
- Factor out common behavior (setSelected, isSelected)
- Subclasses inherit common behavior
- Some methods still undefined (draw, drawSelection, translate, contains)

Abstract Classes

```
public abstract class SelectableShape implements
SceneShape
{
   public void setSelected(boolean b) { selected = b; }
   public boolean isSelected() { return selected; }

   private boolean selected;
}
```

- Problem with the SelctableShape class !!!
  - `SelectableShape` doesn't define all `SceneShape` methods.

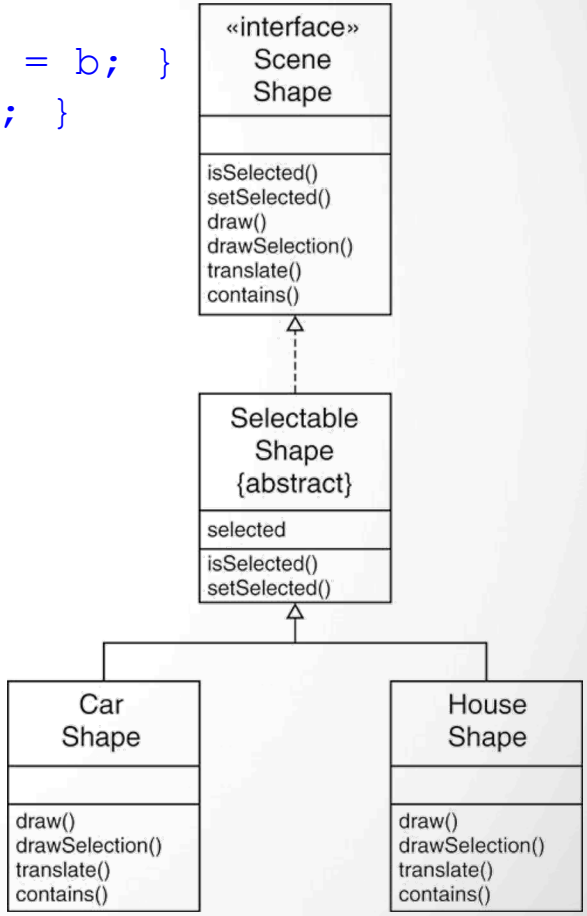


Figure 7:
Relationship Between SelectableShape Types

Abstract Classes

- `HouseShape` and `CarShape` are ***concrete*** *subclasses that define the remaining methods.*

- Can't instantiate abstract class:

```
SelectableShape s = new SelectableShape(); // ERROR
```

- Ok to have variables of abstract class type:

```
SelectableShape s = new HouseShape(); // OK
```

Abstract Classes and Interface Types

Abstract class	Interface
Abstract classes can have fields.	Interface types can only have constants (`public static final`).
Abstract classes can **define** methods.	Interface types can only **declare** methods.
In Java, a class can **extend** ONLY one other class.	A class can **implement** any number of interface types.

Scene Editor

- Mouse listener selects/unselects item
- Mouse motion listener drags item
- Remove button removes selected items

SceneComponent.java

SceneEditor.java

HouseShape.java

Uniform Highlighting Technique

- **Old approach:** each shape draws its selection state.
  - Inconsistent.
- **Better approach:** shift, draw, shift, draw, restore to original position.
- Define in `SelectableShape`

```
public void drawSelection(Graphics2D g2)
   {
      translate(1, 1);
      draw(g2);
      translate(1, 1);
      draw(g2);
      translate(-2, -2);
   }
```

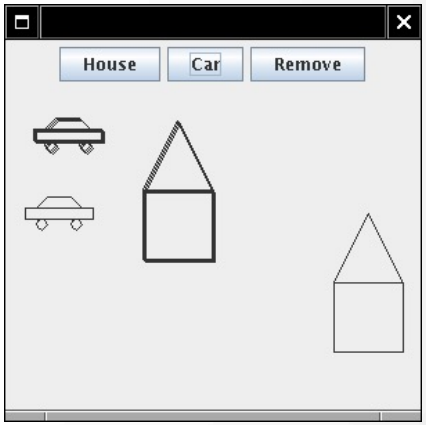


Figure 8:
Highlighting a Shape

Template Method

SelectableShape.java
HouseShape.java

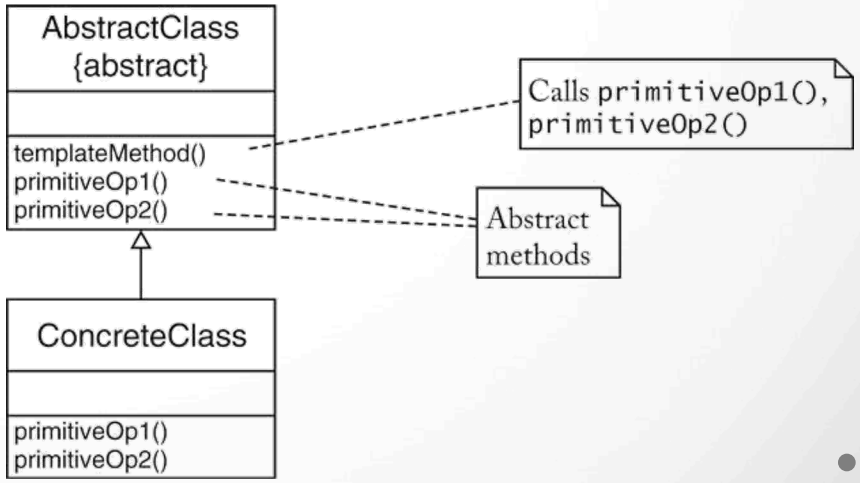
- **draw** Defined in CarShape, HouseShape
- drawSelection method calls draw.
- drawSelection doesn't know *which* methods – polymorphism
- drawSelection is a **TEMPLATE method.**

TEMPLATE METHOD Pattern

- **Context:**
  1. An algorithm is **applicable for multiple types.**
  2. The algorithm can be broken down into *primitive operations*. The primitive operations can be different for each type.
  3. The order of the primitive operations doesn't depend on the type.

TEMPLATE METHOD Pattern

- **Solution:**
  1. Define a superclass that has a method for the algorithm and abstract methods for the primitive operations.
  2. Implement the algorithm to call the primitive operations in the appropriate order.
  3. Do not define the primitive operations in the superclass, or define them to have appropriate default behavior.
  4. Each subclass defines the primitive operations but not the algorithm.



TEMPLATE METHOD Pattern

Name in Design Pattern	Actual Name (Selectable shapes)
AbstractClass	SelectableShape
ConcreteClass	CarShape, HouseShape
templateMethod()	drawSelection
primitiveOp1(), primitiveOp2()	translate, draw

Compound Shapes

- **GeneralPath:** sequence of shapes.

  `java.awt.geom.GeneralPath`

```
GeneralPath path = new GeneralPath();
path.append(new Rectangle(...), false);
path.append(new Triangle(...), false);
g2.draw(path);
```

- Advantage: Containment test is free

  `path.contains(aPoint);`

CompoundShape.java

HouseShape.java

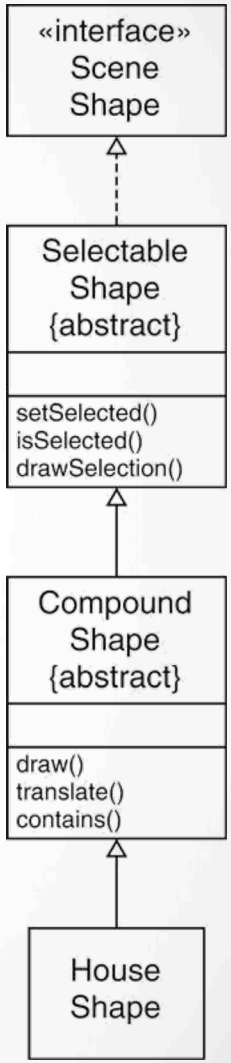


Figure 9: Inheritance Diagram of the HouseShape Class

When Not to Use Inheritance

- **Inheritance** is used to model an *is-a* relationship.
  - Car is a Vehicle.
- **Aggregation** is used to modes *has- a* relationship.
  - Car has a tire.

When Not to Use Inheritance

- From a tutorial for a C++ compiler:

```
public class Point
{
   public Point(int anX, int aY) { ... }
   public void translate(int dx, int dy) { ... }
   private int x;
   private int y;
}

public class Circle extends Point // DON'T
{
   public Circle(Point center, int radius) { ... }
   public void draw(Graphics g) { ... }
   private int radius;
}
```

When Not to Use Inheritance

- Huh? A circle isn't a point.
- By accident, inherited translate works for circles
- Same tutorial makes Rectangle a subclass of Point:

```
public class Rectangle extends Point // DON'T
{
   public Rectangle(Point corner1, Point corner2)        { ... }
   public void draw(Graphics g) { ... }
   public void translate(int dx, int dy) { ... }

   private Point other;
}
```

- That's even weirder:

```
public void translate(int dx, int dy)
{
   super.translate(dx, dy);
   other.translate(dx, dy);
}
```

Remedy: Use aggregation.

Circle, Rectangle classes *have* points

When Not to Use Inheritance

- Java standard library:

```
public class Stack<T> extends Vector<T> // DON'T
{
    T pop() { ... }
    void push(T item) { ... }
    ...
}
```

- Bad idea: Inherit all Vector methods
- Can insert/remove in the middle of the stack
- Remedy: Use aggregation

```
public class Stack<T>
{
    ...
    private ArrayList<T> elements;
}
```

End of Chapter 6