Java:

Learning to Program with Robots

Chapter 02: Extending Classes with Services

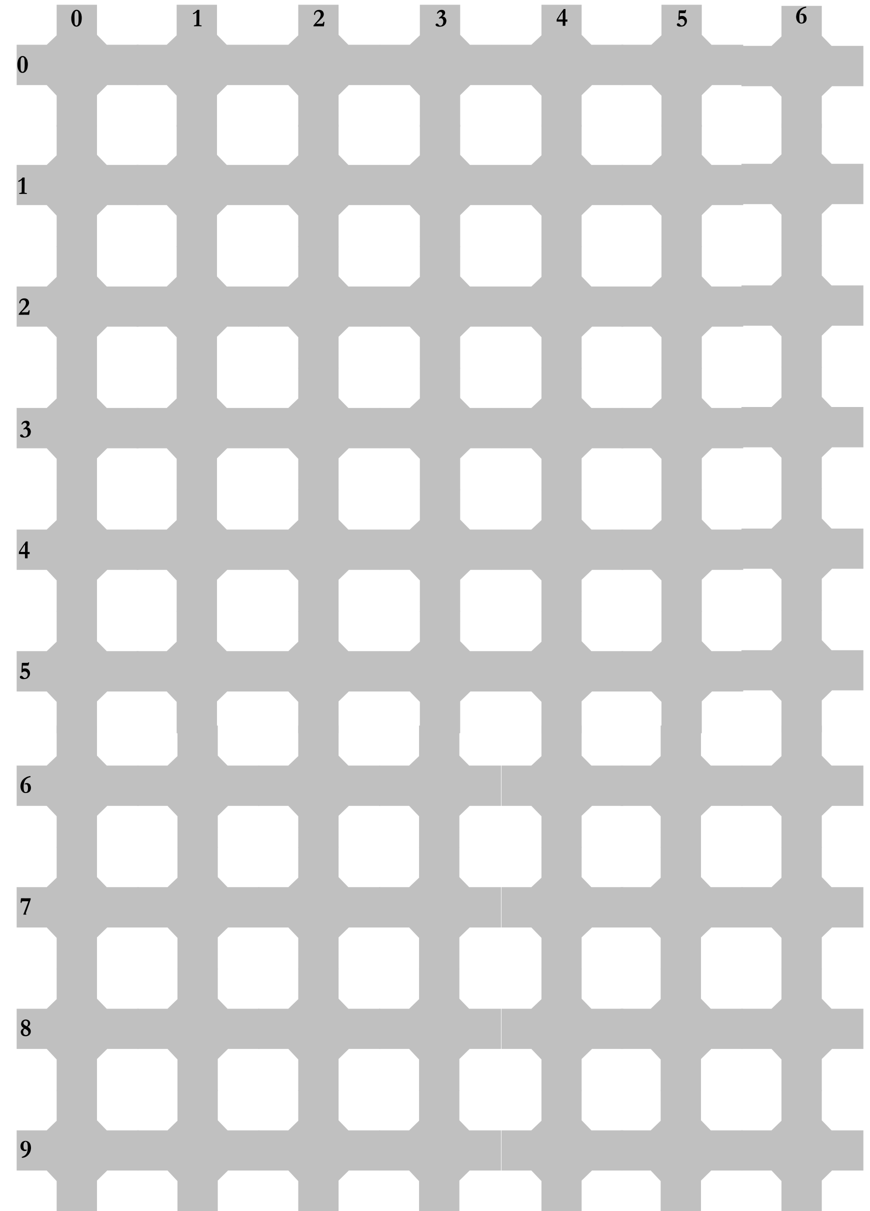
Chapter Objectives

After studying this chapter, you should be able to:

- Extend an existing class with new commands.
- Explain how a message sent to an object is resolved to a particular method.
- Use inherited services in an extended class.
- Override services in the superclass to provide different functionality.
- Follow important stylistic conventions for Java programs.
- Extend a graphical user interface component to draw a scene.
- Add new kinds of **Thing**s and **Robot**s to the robot world.

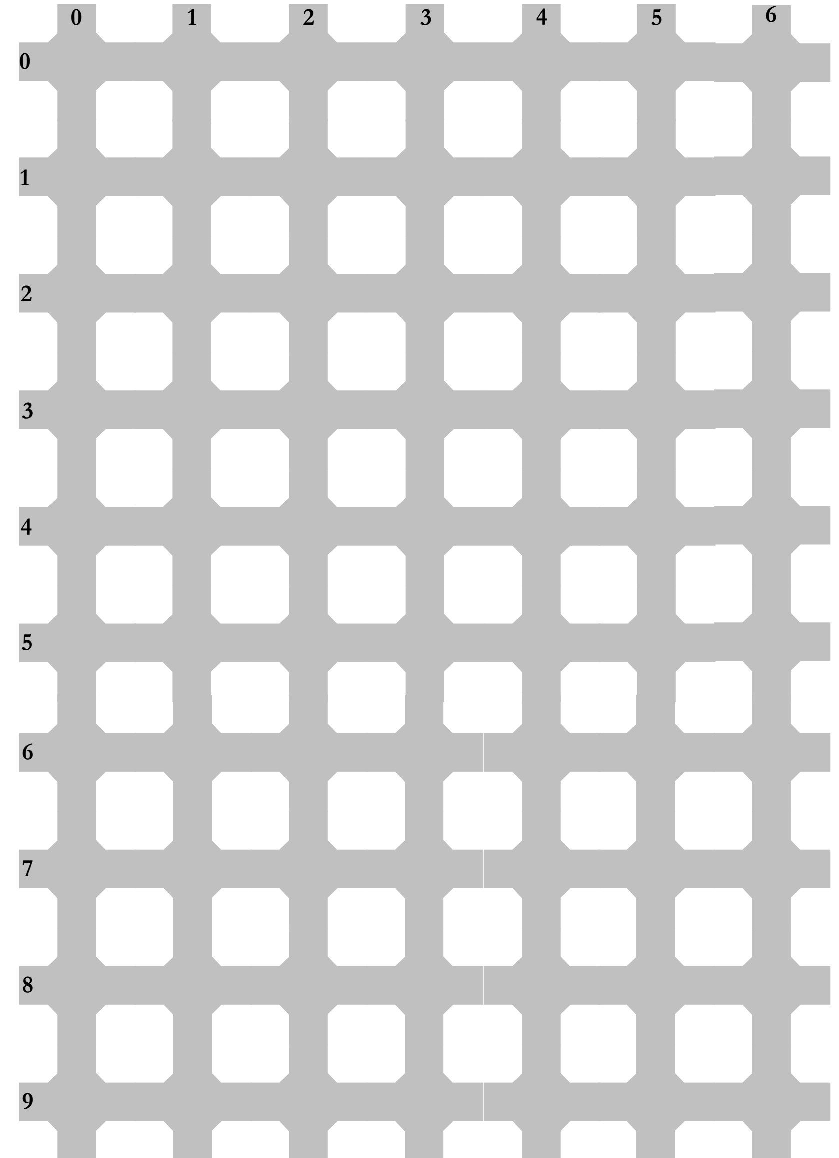
2.1: An Experiment, Program 1

```
5    public static void main(String[ ] args)
6    {  City austin = new City();
7       Robot lisa = new Robot(austin, 3, 3, Direction.EAST);
9       lisa.move();
10      lisa.move();
11      lisa.move();
12      lisa.turnLeft();
13      lisa.turnLeft();
14      lisa.turnLeft();
15      lisa.move();
16      lisa.move();
17      lisa.move();
18      lisa.turnLeft();
19      lisa.turnLeft();
20      lisa.move();
21      lisa.move();
22      lisa.move();
23      lisa.turnLeft();
24      lisa.move();
25      lisa.move();
26      lisa.move();
27      lisa.turnLeft();
28      lisa.turnLeft();
     }
```



2.1: An Experiment, Program 2

```
5    public static void main(String[ ] args)
6    {  City austin = new City();
7       ExperimentRobot lisa =
8              new ExperimentRobot(austin, 3, 2, Direction.SOUTH);
9
10      lisa.move3();
11      lisa.turnRight();
12      lisa.move3();
13      lisa.turnAround();
14      lisa.move3();
15      lisa.turnLeft();
16      lisa.move3();
17      lisa.turnAround();
18   }
```



The idea in software:

- Start with an existing class, such as **Robot**.
- Extend it with new capabilities to perform related, but new, services such as **turnRight**.

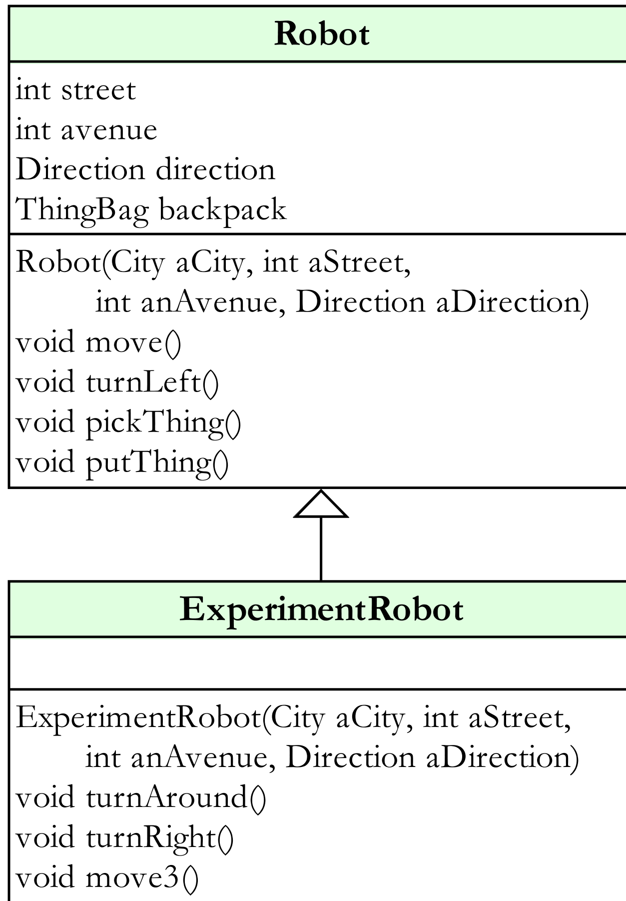
Benefits:

- Make use of existing functionality easily.
- Customize it for your particular application.

The idea in manufacturing:

- Start with an existing product, such as a delivery van.
- Extend it with new features for a niche market, such as camping.

2.2.1: Vocabulary of Extending Classes



Superclass

Subclass

"**ExperimentRobot** extends **Robot**"

"**ExperimentRobot** inherits from **Robot**"

```
1 import «importedPackage»;
2
3 public class «className» extends «superClass»
4 {
5   «list of attributes used by this class»
6   «list of constructors for this class»
7   «list of services provided by this class»
8 }
```

- To extend the **Robot** class, import **becker.robots.***
- The list of attributes will be empty until Chapter 6

2.2.3: Implementing a Constructor (1/2)

ExperimentRobot

Robot

street:	
avenue:	
direction:	
backpack:	

Robot(City aCity, int aStreet, int anAvenue, Direction aDirection)
void move( )
void turnLeft( )
void pickThing( )
void putThing( )

ExperimentRobot(City aCity, int aStreet, int anAvenue, Direction aDirection)
void turnAround( )
void turnRight( )
void move3( )

2.2.3: Implementing a Constructor (2/2)

```
import becker.robots.*;

public class ExperimentRobot extends Robot
{
   // No new attributes.

   // A constructor to initialize the ExperimentRobot and the Robot-inside-the-ExperimentRobot.
   public ExperimentRobot(City aCity, int aStreet,
                          int anAvenue, Direction aDirection)
   {  super(aCity, aStreet, anAvenue, aDirection);
   }

   // Another constructor to initialize the ExperimentRobot to be in a standard position.
   public ExperimentRobot(City aCity)
   {  super(aCity, 0, 0, Direction.EAST);
   }

   // The new services offered by an ExperimentRobot will be inserted here.
}
```

Usage:

```
ExperimentRobot lisa = new ExperimentRobot(
                                 austin, 3, 2, Direction.SOUTH);
ExperimentRobot larry = new ExperimentRobot(austin);
```

2.2.4: Adding a Service

```java
import becker.robots.*;

public class ExperimentRobot extends Robot
{
   // No new attributes.

   // A constructor to initialize the ExperimentRobot and the Robot-inside-the-ExperimentRobot.
   public ExperimentRobot(City aCity, int aStreet,
                          int anAvenue, Direction aDirection)
   {  super(aCity, aStreet, anAvenue, aDirection);
   }

   // Another constructor to initialize the ExperimentRobot to be in a standard position.
   public ExperimentRobot(City aCity)
   {  super(aCity, 0, 0, Direction.EAST);
   }

   // A new service to turn the robot around.
   public void turnAround()
   {  this.turnLeft();
      this.turnLeft();
   }
}
```

The *flow of control* is the sequence in which statements are executed.

Calling a method such as **turnAround** alters the flow of control to execute the statements contained in the method.

```
public ... main(...)
{   ...
    lisa.turnAround();
    lisa.move();
    ...
}
```

```
public void turnAround()
{   this.turnLeft();
    this.turnLeft();
}
public void move()
{   ...
}
```

Quick Quiz

Implement the other methods used in the experiment at the beginning of the lesson. That is:

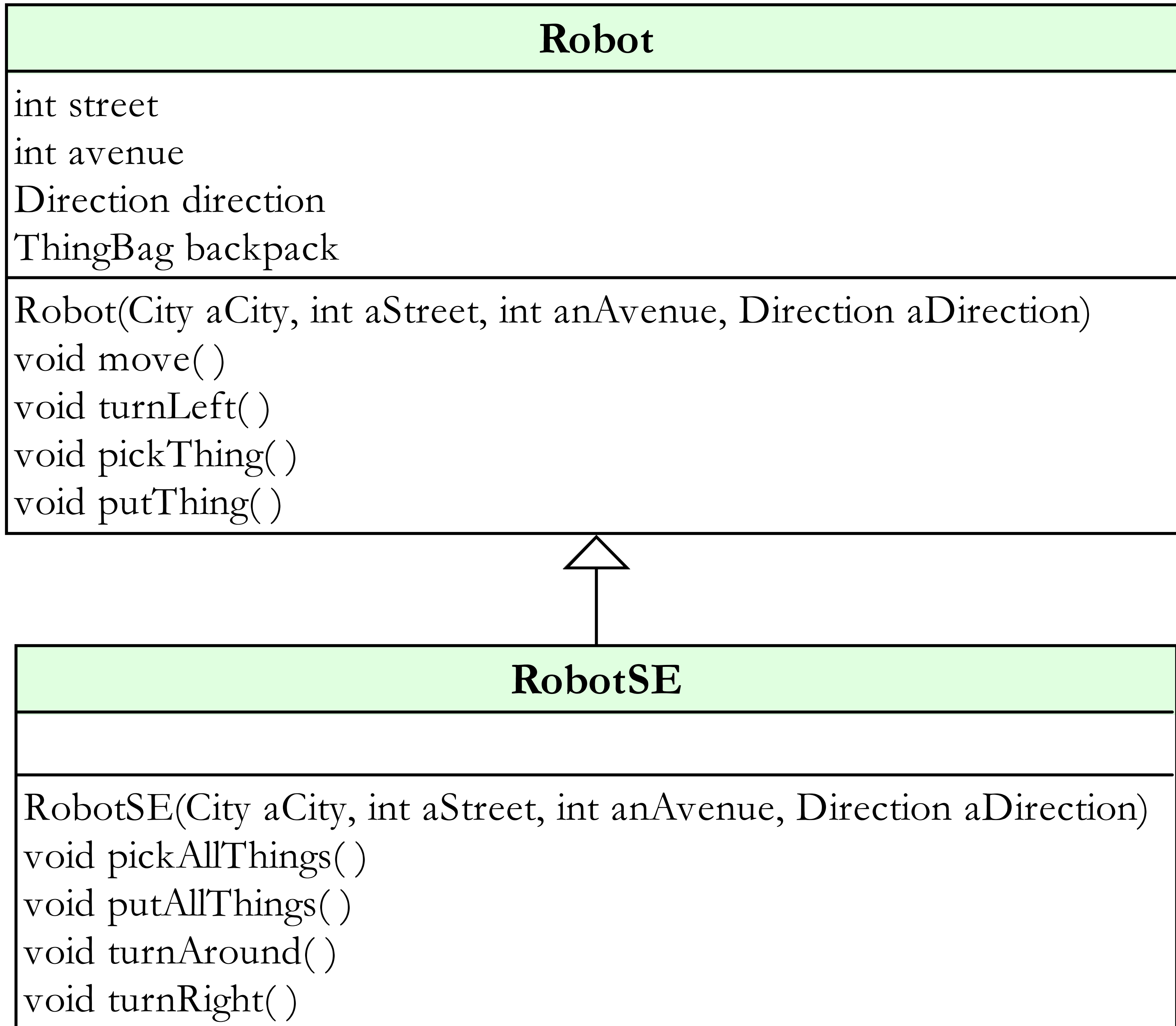
1. Implement **move3**, which moves the robot forward 3 times.
2. Implement **turnRight**, which turns the robot left three times (equivalent to turning right).
3. Bonus question: Implement **turnRight** differently than your answer to the previous question.

Quick Quiz Solutions

```
import becker.robots.*;

public class ExperimentRobot extends Robots
{   public ExperimentRobot(City aCity, int aStreet, int anAvenue, Direction aDirection)
    {   super(aCity, aStreet, anAvenue, aDirection);
    }

    public void move3()
    {  this.move();
       this.move();
       this.move();
    }

    public void turnRight()
    {  this.turnLeft();
       this.turnLeft();
       this.turnLeft();
    }

    public void turnAround()
    {   this.turnLeft();
        this.turnLeft();
    }
}
```

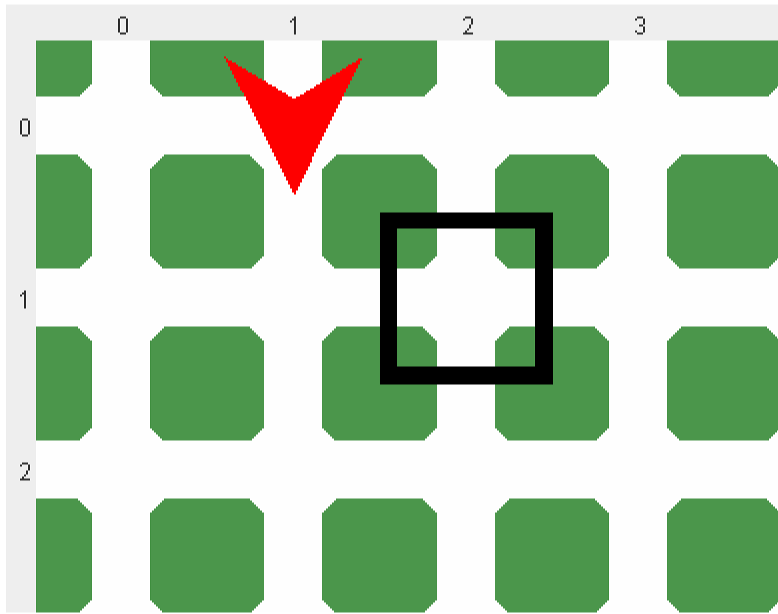
```
//  Alternate solution
public void turnRight()
{  this.turnLeft();
   this.turnAround();
}
```

2.2.7: RobotSE

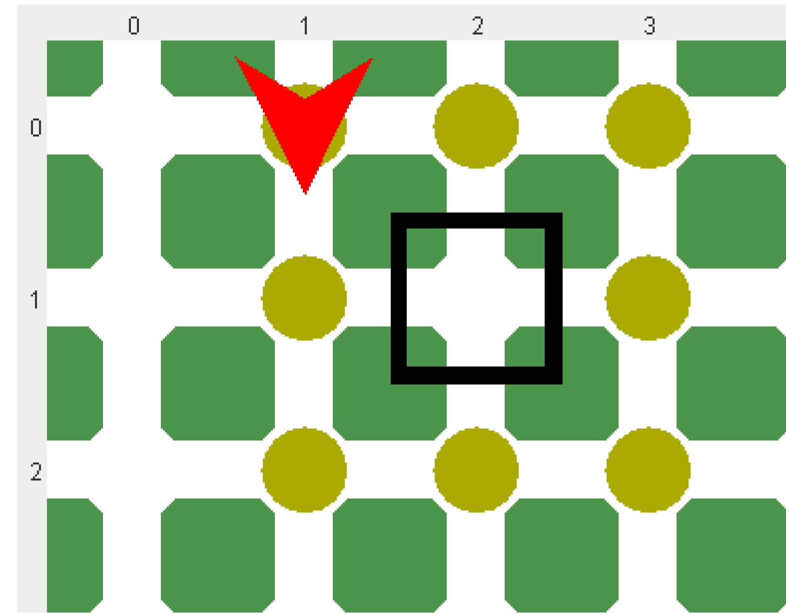


Case Study: Planting Flowers (revisited)

In the previous lesson we instructed a robot to plant flowers around a square wall:



Initial Situation



Final Situation

Create and use a **GardenerBot** that contains a service named **plantFlowers**.

Create an extended version of **City** named **Garden** that already contains a wall as shown in the initial situation.

Case Study: Setting up the main method

```
import becker.robots.*;

// Plant flowers around a square garden wall.
public class PlantFlowers
{
   public static void main(String[ ] args)
   {
      // Code to create the initial situation goes here.
      // This is to be replaced with a Garden object that has the walls already built.
      City berlin = new City();
      Wall eWall = new Wall(berlin, 1, 2, Direction.EAST);
      Wall nWall = new Wall(berlin, 1, 2, Direction.NORTH);
      Wall wWall = new Wall(berlin, 1, 2, Direction.WEST);
      Wall sWall = new Wall(berlin, 1, 2, Direction.SOUTH);

      // Create a robot with 8 things already in its backpack.
      Gardener karel = new Gardener(berlin, 0, 1, Direction.SOUTH, 8);

      // Code to plant the flowers goes here.
      karel.plantFlowers();
   }
}
```

Case Study: Gardener Class (Step 1)

```
import becker.robots.*;

public class Gardener extends Robot
{
   // Create a new Gardener robot.
   public Gardener(City aCity, int aStreet, int anAvenue,
                              Direction aDirection, int numThings)
   {  super(aCity, aStreet, anAvenue, aDirection, numThings);
   }

}
```

Case Study: Gardener Class (Step 2)

```
import becker.robots.*;

public class Gardener extends Robot
{
    // Create a new Gardener robot.
    public Gardener(City aCity, int aStreet, int anAvenue,
                                    Direction aDirection, int numThings)
    {   super(aCity, aStreet, anAvenue, aDirection, numThings);
    }

    public void plantFlowers()
    {  this.plantOneSide();
       this.plantOneSide();
       this.plantOneSide();
       this.plantOneSide();
    }

    public void plantOneSide()
    {  this.move();
       this.putThing();
       this.move();
       this.putThing();
       this.turnLeft();
    }
}
```

Case Study: The Garden Class

```
import becker.robots.*;

// Plant flowers around a square garden wall.
public class PlantFlowers
{
   public static void main(String[ ] args)
   {
      // Code to create the initial situation goes here.
      City berlin = new City();
      Wall eWall = new Wall(berlin, 1, 2, Direction.EAST);
      Wall nWall = new Wall(berlin, 1, 2, Direction.NORTH);
      Wall wWall = new Wall(berlin, 1, 2, Direction.WEST);
      Wall sWall = new Wall(berlin, 1, 2, Direction.SOUTH);

      Garden berlin = new Garden();
      berlin.buildWalls();

      // Create a robot with 8 things already in its backpack.
      Gardener karel = new Gardener(berlin, 0, 1, Direction.SOUTH, 8);

      // Code to plant the flowers goes here.
      karel.plantFlowers();
   }
}
```

Case Study: The Garden Class (version 1)

```
import becker.robots.*;

public class Garden extends City
{
   public Garden()
   {
      super();
   }

   // Add walls to this garden.
   public void buildWalls()
   {  Wall eWall = new Wall(this, 1, 2, Direction.EAST);
      Wall nWall = new Wall(this, 1, 2, Direction.NORTH);
      Wall wWall = new Wall(this, 1, 2, Direction.WEST);
      Wall sWall = new Wall(this, 1, 2, Direction.SOUTH);
   }

}
```

Case Study: The Garden Class (version 2)

```
import becker.robots.*;

public class Garden extends City
{
   public Garden()
   {
      super();

      // Add walls to this garden.
      Wall eWall = new Wall(this, 1, 2, Direction.EAST);
      Wall nWall = new Wall(this, 1, 2, Direction.NORTH);
      Wall wWall = new Wall(this, 1, 2, Direction.WEST);
      Wall sWall = new Wall(this, 1, 2, Direction.SOUTH);
   }
}
```

2.4.1: White Space and Indentation

```
import becker.robots.*; public class ExperimentRobot extends Robot {
public ExperimentRobot(City aCity, int aStreet, int anAvenue, Direction
aDirection) { super(aCity, aStreet, anAvenue, aDirection);}  public void
turnAround() { this.turnLeft(); this.turnLeft(); } public void move3(){
this.move(); this.move(); this.move();  } public void turnRight() {
this.turnAround(); this.turnLeft(); }}
```

White space and appropriate indentation make programs easier to read.

- Begin each statement on a new line.
- Include at least one blank line between blocks of code with different purposes.
- Line up curly braces so that the closing brace is directly beneath the opening brace.
- Indent everything inside a pair of braces by a consistent number of spaces.

- Java reserves symbols such as **{**, **}**, **;**, **(**, **)**, **+**, …. They can't be used as programmer-chosen names (identifiers).
- Java reserves keywords such as **public**, **class**, **void**, **import**, **extends**, etc. They can't be used as identifiers.

Identifier	Conventions	Examples
Class	A descriptive singular noun, beginning with an uppercase letter. If the name is composed of several words, each word begins with a capital letter.	**Robot** **Wall** **Gardener** **MazeCity**
Method	A descriptive verb or verb phrase, beginning with a lowercase letter. If the method name is composed of several words, each word, except the first, begins with a capital letter.	**move** **pickThing** **turnAround** **plantFlowers**
Variable	A descriptive noun or noun phrase, beginning with a lowercase letter. If the name is composed of several words, each word, except the first, begins with a capital letter.	**karel** **berlin** **northWall**

2.4.3: Comments

```java
import becker.robots.*;

/** A new kind of robot that can turn around, turn right, and move
*   forward three intersections at a time.
*   @author Byron Weber Becker **/
public class ExperimentRobot extends Robot
{
   /**Initialize this ExperimentRobot to start at the origin (0, 0)
   *   facing EAST.
   *   @param aCity The city in which this robot is to appear. */
   public ExperimentRobot(City aCity)
   {  super(aCity, 0, 0, Direction.EAST);
   }

   public void turnRight()
   {  /* One strategy for turning right is to first turn around
         and then turn left one more time. */
      this.turnAround();
      this.turnLeft();          // The last step in turning right.
   }
}
```

2.4.3: Comments

	Single-Line	Multi-Line	Documentation
Extent	// to end of the line	From /* to */	From /** to */
Purpose	Short comment, possibly at end of line of code	Extended documentation Comment-out several lines of code	Generate external documentation (e.g. web pages) for reference
Nestable	Inside multi-line and doc comments	No	No
Uses tags	No	No	Yes
Appears	Anywhere	Anywhere	Just before classes and methods

Consider trying to understand the following code (from the experiment performed earlier):

```
10        lisa.move3();
11        lisa.turnRight();
12        lisa.move3();
13        lisa.turnAround();
14        lisa.move3();
15        lisa.turnLeft();
16        lisa.move3();
17        lisa.turnAround();
```

After a little more looking, you discover that **move3** is defined as

```
public void move3()
{  this.turnLeft();
   this.pickThing();
   this.turnLeft();
}
```

Some people "show off" when they move. They might be loud or swagger or sway their hips or …. It seems like they can't move in any other way. If they move, they show off.

Create a **ShowOff** robot that turns magenta each time it moves. When it is standing still or turning (that is, not **move**-ing) it should continue to be red.

```
import becker.robots.*;

public class ShowOffMain
{
  public static void main(String[ ] args)
  {
    City ny = new City();
    ShowOff karel = new ShowOff(ny, 2, 3, Direction.SOUTH);

    karel.turnLeft();      // red
    karel.move();          // magenta
    karel.move();          // magenta
    karel.turnLeft();      // red
    karel.move();          // magenta
    …
```

Example:

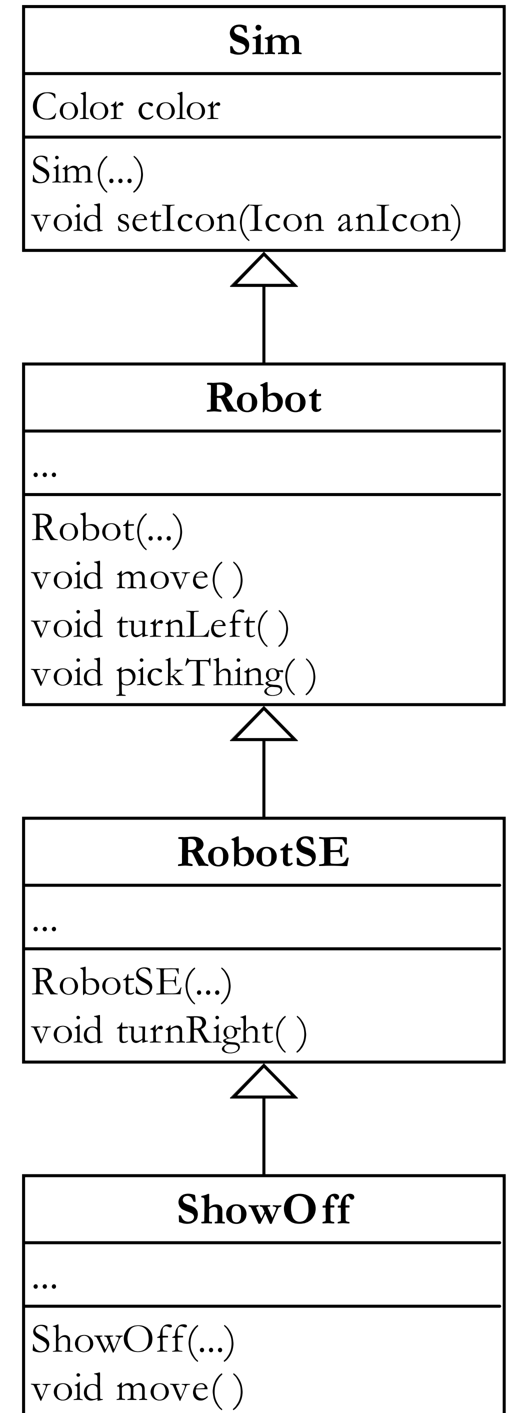
```
import becker.robots.*;
import java.awt.Color;

public class ColorExample
{
   public static void main(String[ ] args)
   {
      City ny = new City();
      ShowOff karel = new ShowOff(ny, 2, 3, Direction.SOUTH);

      karel.setColor(Color.RED);
      karel.turnLeft();

      karel.setColor(Color.MAGENTA);
      karel.move();
      karel.move();

      karel.setColor(Color.RED);
      karel.turnLeft();
   }
}
```

This is NOT the desired solution! **karel** should turn magenta automatically every time it moves. The programmer using the **ShowOff** robot shouldn't have to remember to do it each time.

2.6: Overriding a Method

```
import becker.robots.*;
import java.awt.Color;

/** A kind of robot that shows off by turning color when it moves.
*   @author Byron Weber Becker */
public class ShowOff extends RobotSE
{
    public ShowOff(City aCity, int aStreet,
                   int anAvenue, Direction aDir)
    {   super(aCity, aStreet, anAvenue, aDir);
    }

    /** Override move so this robot, turns magenta each time it
    *   moves.  Make it red again after moving.  */
    public void move()
    {  this.setColor(Color.MAGENTA);
       super.move();
       this.setColor(Color.RED);
    }
}
```

```
    public void static main(String[ ] args)
    {  …
       ShowOff karel = new ShowOff(…);
       karel.turnLeft();   // What happens?
       karel.move();       // What happens?
    }
```



2.8.1: The Extended Class Pattern

Name: Extended Class

Context: A new kind of object with customized services is required.

Solution:

```
import «importedPackage»; // may have 0 or more import statements
public class «className» extends «superClass»
{ «list of attributes used by this class»
  «list of constructors for this class»
  «list of services provided by this class»
}
```

For example,

```
import becker.robots.*;
public class Gardener extends Robot
{ public Gardener(…) {   super(…);  }
  public void plantFlowers() { …   }
}
```

Consequences: Objects inherit services from the superclass and can either override them or supplement them with new services.

Related Patterns: Constructor and Parameterless Command patterns both occur within an instance of the Extended Class pattern.

2.8.2: The Constructor Pattern

Name: Constructor

Context: Instances of a class must be initialized when constructed.

Solution: Add a constructor to the class to carry out initialization, including initializing the object's superclass.

```
/**«Description of what this constructor does.»
   @param «parameterName» «Description of parameter»
*/
public «className»(«parameter list»)
{ super(«arguments»);
   «list of Java statements»
}
```

The *«parameter list»* contains zero or more parameters, each with a type and name such as **City aCity**. There should be an **@param** line in the documentation comment for each parameter.

Consequences: Instances of the class are consistently initialized.

Related Patterns: This pattern always occurs within a pattern for a class, such as the Extended Class pattern.

2.8.3: The Parameterless Command Pattern

Name: Parameterless Command

Context: A new service is needed in a class. It does not need any additional information to carry out its purpose.

Solution:

```
/** «Description of the command.»
*/
public void «commandName»()
{ «list of statements»
}
```

Example:

```
/** Turn the robot around to face the opposite direction. */
public void turnAround()
{ this.turnLeft();
  this.turnLeft();
}
```

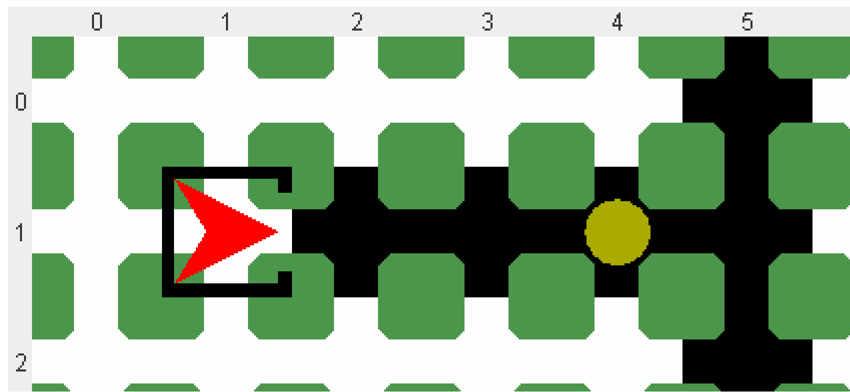
Consequences: The new service is available to clients of the class.

Related Patterns: This pattern always occurs within a pattern for a class. The Extended Class pattern is one such pattern.

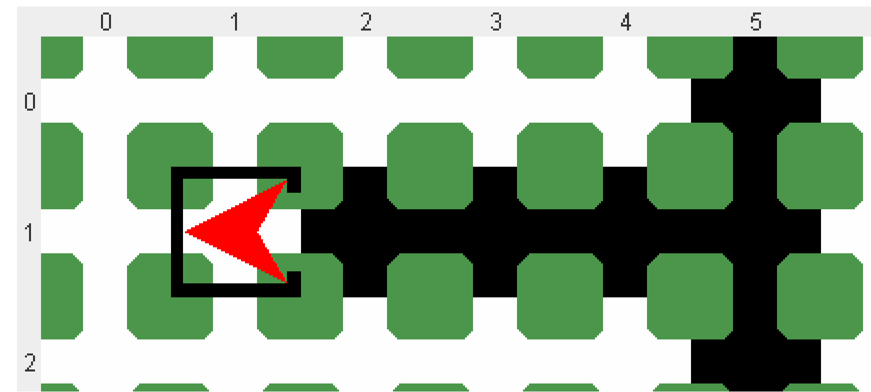
Case Study: Fetching the Newspaper

A pyjama-clad robot is fetching the morning paper (a **Thing**). Unfortunately, the deliverybot left it at the end of the driveway. The robot is rather shy and doesn't want to be seen in its pyjamas, but also wants the paper. It decides to venture forth to get the paper, but can't help itself from looking both ways **every time** it moves.

Write a program with a **main** method that sets up the scenario. Implement new kind of robot, a **ShyBot**, that looks both ways before moving. Implement a new kind of **City** that includes a house (3 walls).



Initial Situation



Final Situation

Bonus: "Pave" the driveway and street by coloring the intersections black.

Hint: Research the **getIntersection** method in the **City** class.

Case Study: The main method

```
import becker.robots.*;

/** Fetch the paper with a shy robot.
*   @author Byron Weber Becker */
public class FetchPaper1
{  public static void main(String[] args)
   {
      House house = new House();       // House is a special kind of City
      ShyBot pat = new ShyBot(house, 1, 1, Direction.EAST);
      Thing paper = new Thing(house, 1, 4);

      // Go fetch the paper at the end of the driveway, looking both ways every time pat moves.
      pat.move();
      pat.move();
      pat.move();
      pat.pickThing();
      pat.turnAround();
      pat.move();
      pat.move();
      pat.move();
   }
}
```

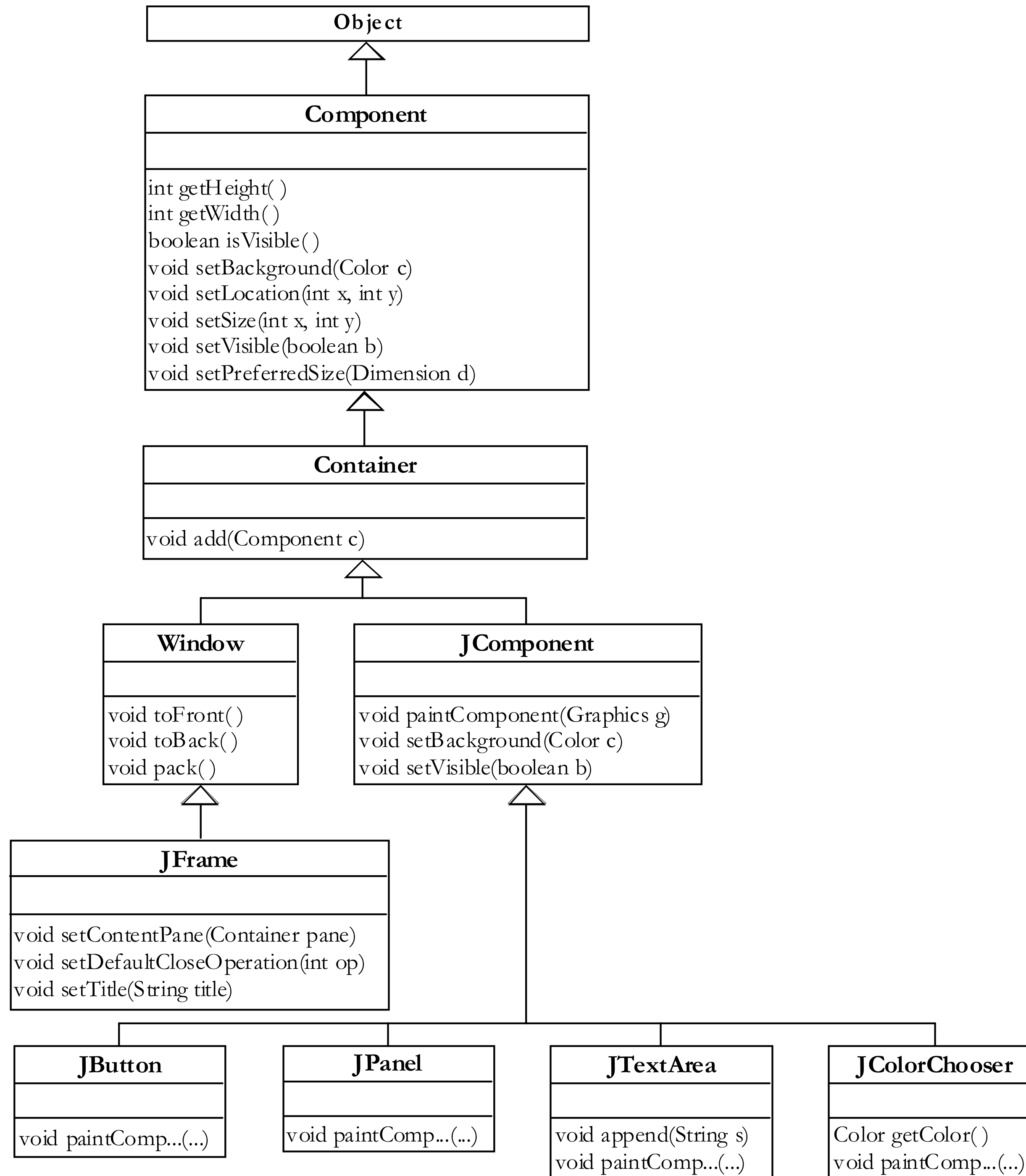
Case Study: The ShyBot Class

```
import becker.robots.*;

/** A shy robot that always looks both ways before moving.
*   @author Byron Weber Becker */
public class ShyBot extends RobotSE
{
   /** Construct a shy robot at the given intersection.
    *  @param  aCity    The city in which this robot appears.  Remaining @param tags omitted. */
   public ShyBot(City aCity, int aStreet, int anAvenue, Direction aDir)
   {  super(aCity, aStreet, anAvenue, aDir);
   }

   /** Look both ways and then move to the next intersection. */
   public void move()
   {  this.lookBothWays();
      super.move();
   }

   /** Look both ways (to the right and then to the left). */
   public void lookBothWays()
   {  this.turnRight();
      this.turnLeft();
      this.turnLeft();
      this.turnRight();
   }
}
```

Case Study: The House Class

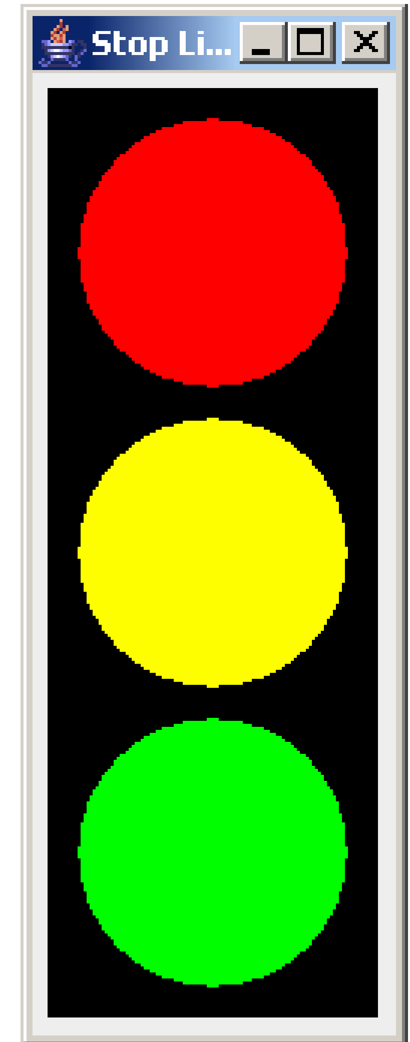
```java
import becker.robots.*;
import java.awt.Color;

/** The house and driveway for a shy robot.
*   @author Byron Weber Becker */
public class House extends City
{
   public House()
   {  super();

      // build the house
      Wall north = new Wall(this, 1, 1, Direction.NORTH);
      Wall west = new Wall(this, 1, 1, Direction.WEST);
      Wall south = new Wall(this, 1, 1, Direction.SOUTH);

      // "pave" the driveway
      this.getIntersection(1, 2).setColor(Color.BLACK);
      this.getIntersection(1, 3).setColor(Color.BLACK);
      this.getIntersection(1, 4).setColor(Color.BLACK);

      // "pave" the street
      this.getIntersection(0, 5).setColor(Color.BLACK);
      this.getIntersection(1, 5).setColor(Color.BLACK);
      this.getIntersection(2, 5).setColor(Color.BLACK);
   }
}
```

Application: The Swing Inheritance Hierarchy



Application: A Stop Light Component

Write a program to display a stop light, as shown on the right.



Hints:

- Start with the **FramePlay** program from Chapter 1.
- Extend **JComponent** and override **paintComponent** to display a stoplight.
- Research the **Graphics** class to find ways to paint circles and rectangles.

Application: Modifying FramePlay (1/2)

```
import javax.swing.*;
public class FramePlay
{
   public static void main(String[] args)
   {  // declare the objects to show
      JFrame frame = new JFrame();
      JPanel contents = new JPanel();
      JButton saveButton = new JButton("Save");
      JTextArea textDisplay = new JTextArea(5, 10);

      // set up the contents
      contents.add(saveButton);
      contents.add(textDisplay);

      // set the frame's contents to display the panel
      frame.setContentPane(contents);

      // set up and show the frame
      frame.setTitle("FramePlay");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setLocation(250, 100);
      frame.setSize(150, 200);
      frame.setVisible(true);
   }
}
```

Unmodified **FramePlay** program from Chapter 01.

Application: Modifying FramePlay (2/2)

```
import javax.swing.*;

/** Display a frame containing a customized component.
*   @author Byron Weber Becker */
public class StopLightMain
{
   public static void main(String[] args)
   {  // Declare the objects to show.
      JFrame frame = new JFrame();
      JPanel contents = new JPanel();
      StopLight light = new StopLight();

      // Add the stop light to the contents.
      contents.add(light);

      // Display the contents in a frame.
      frame.setContentPane(contents);
      frame.setTitle("Stop Light");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setLocation(250, 100);
      frame.pack();        // Calculate the size of the frame based on the component size(s).
      frame.setVisible(true);
   }
}
```

Application: Extending JComponent (1/5)

```
import javax.swing.*;
import java.awt.*;

/** A new kind of component that displays a stop light.
*   @author Byron Weber Becker */
public class StopLight extends JComponent
{
   public StopLight()
   {  super();
   }

   /** Paint a stoplight. */
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);        // Let the superclass do it's work

   }
}
```

The program runs at this stage, but the size is completely wrong and (of course) there is no stop light painted yet:

Overriding a method such as **paintComponent** requires that the new method have exactly the same *signature* as the original: the same name, parameters, and return type.

Application: Extending JComponent (2/5)

JComponent (Java 2 Platform SE 5.0) - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

file:///C:/java/jdk1.5/docs/api/index.html Go

java.awt.color
java.awt.datatrar
java.awt.dnd

JComboBox
JComboBox.Ke
JComponent
JdbcRowSet
JDesktopPane
JDialog
JEditorPane
JFileChooser

setPreferredSize

```
public void setPreferredSize(Dimension preferredSize)
```

Sets the preferred size of this component. If `preferredSize` is `null`, the UI will be asked for the preferred size.

Overrides:

`setPreferredSize` in class `Component`

Parameters:

`preferredSize` - The new preferred size, or null

See Also:

Done

Dimension (Java 2 Platform

File Edit View Go Bookma

file:///C:/java/jdk1.5/docs/api/index.htr Go

java.awt.color
java.awt.datat
java.awt.dnd

Dictionary
DigestExcepti
DigestInputSt
DigestOutputS
Dimension
Dimension2D
DimensionUIF
DirContext
DirectColorM

Constructor Summary

`Dimension``()` Creates an instance of `Dimension` with a width of zero and a height of zero.
`Dimension``(Dimension d)` Creates an instance of `Dimension` whose width and height are the same as for the specified dimension.
`Dimension``(int width, int height)` Constructs a `Dimension` and initializes it to the specified width and specified height.

file:///C:/java/jdk1.5/docs/api/java/awt/Dimension.html

Application: Extending JComponent (3/5)

Graphics (Java 2 Platform SE 5.0) - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

file:///C:/java/jdk1.5/docs/api/index.html Go

java.awt.event
java.awt.font
java.awt.geom
java.awt.im
java.awt.im.spi

GraphicAttribute
Graphics
Graphics2D
GraphicsConfigT
GraphicsConfigu
GraphicsDevice
GraphicsEnviron
GrayFilter
GregorianCalenc
GridBagConstrai
GridBagLayout
GridLayout
Group

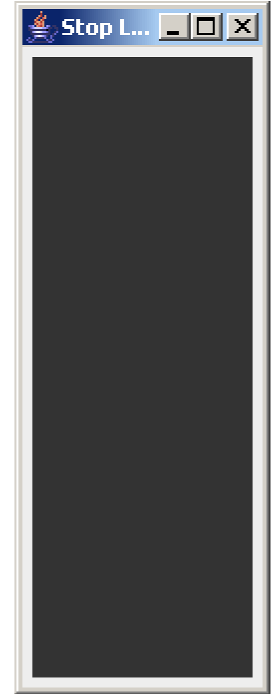
void	int startAngle, int arcAngle) Fills a circular or elliptical arc covering the specified rectangle.
abstract void	**fillOval**(int x, int y, int width, int height) Fills an oval bounded by the specified rectangle with th current color.
abstract void	**fillPolygon**(int[] xPoints, int[] yPoints, int nPoints) Fills a closed polygon defined by arrays of *x* and *y* coordinates.
void	**fillPolygon**(Polygon p) Fills the polygon defined by the specified Polygon obje with the graphics context's current color.
abstract void	**fillRect**(int x, int y, int width, int height) Fills the specified rectangle.
abstract	fillRoundRect(int x, int y, int width,

Find: Graphics Find Next Find Previous Highlight all Match case

Done

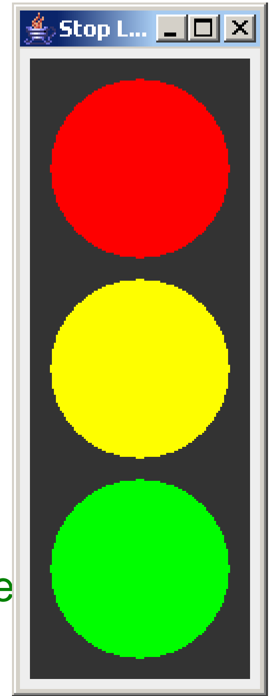
Application: Extending JComponent (4/5)

```
import javax.swing.*;
import java.awt.*;

/** A new kind of component that displays a stop light.
*   @author Byron Weber Becker */
public class StopLight extends JComponent
{
   public StopLight()
   {  super();
      Dimension prefSize = new Dimension(110, 310);
      this.setPreferredSize(prefSize);
   }

   /** Paint a stoplight.   Assume each light is 90 pixels in diameter;  10 pixels between lights.*/
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);

      g.fillRect(0, 0, 110, 310);     // The housing for the stop light
   }
}
```

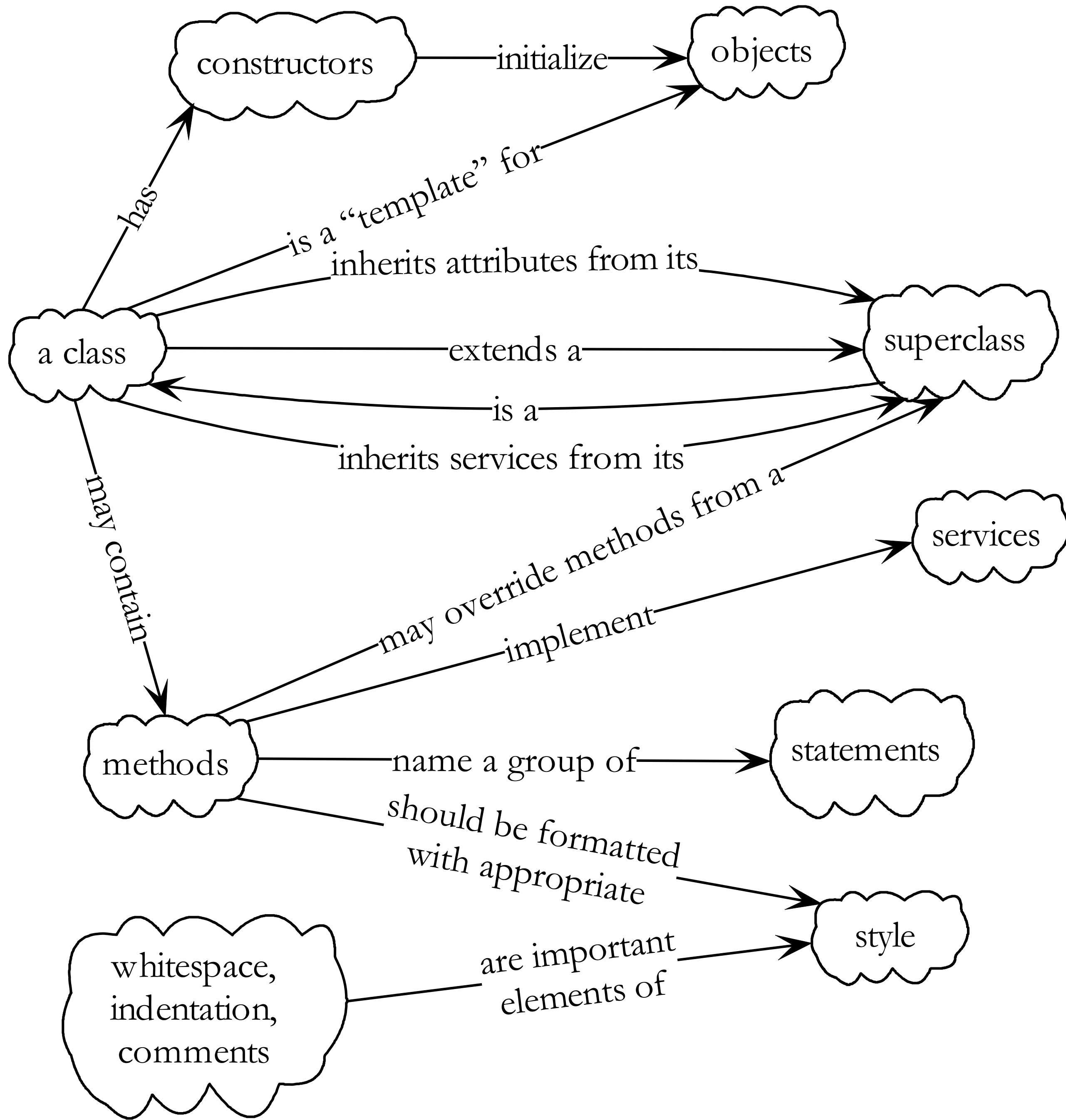


Application: Extending JComponent (5/5)

```
import javax.swing.*;
import java.awt.*;

/** A new kind of component that displays a stop light.*/
public class StopLight extends JComponent
{
    public StopLight()
    {   super();
        Dimension prefSize = new Dimension(110, 310);
        this.setPreferredSize(prefSize);
    }

    /** Paint a stoplight.   Assume each light is 90 pixels in diameter;  10 pixels betwe
    public void paintComponent(Graphics g)
    {  super.paintComponent(g);

        g.fillRect(0, 0, 110, 310);     // The housing for the stop light

        g.setColor(Color.RED);
        g.fillOval(10, 10, 90, 90);

        g.setColor(Color.YELLOW);
        g.fillOval(10, 110, 90, 90);

        g.setColor(Color.GREEN);
        g.fillOval(10, 210, 90, 90);
    }
}
```



2.9: Concept Map



Summary

We have learned:

- how to extend an existing class (**Robot**, **JComponent**) to create a similar class but with new capabilities (**Gardener**, **StopLight**).
- how to write a constructor to initialize the objects.
- how to write new services (methods).
- how to override a method in a superclass.

We have also learned:

- that when a method is called, its statements are executed before execution continues with the statement after the method call.
- that “superclass” refers to the class being extended and that “subclass” refers to the new class.
- how to include the relationship between subclass and its superclass in a class diagram.
- that style – whitespace, comments, identifier names – are important for understanding a program.