Java:

Learning to Program with Robots

Chapter 06: Using Variables

Chapter Objectives

After studying this chapter, you should be able to:

- Add new instance variables to a simple version of the **Robot** class.
- Store the results of calculations in temporary variables and use those results later in the method.
- Write methods using parameter variables.
- Use constants to write more understandable code.
- Explain the differences between instance variables, temporary variables, parameter variables, and constants.
- Extend an existing class with new instance variables.

We'll learn about instance variables by considering a *simplified* version of the **Robot** class.

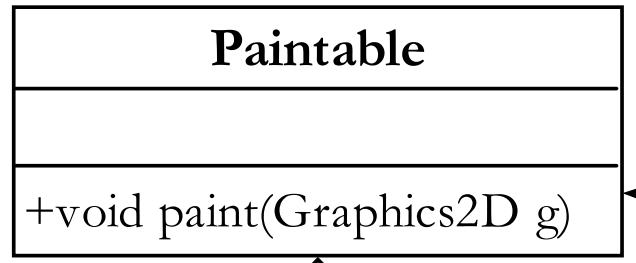
Instance variables are used to store information relevant to an entire object (its attributes). Examples:

- a robot's street, avenue, and direction
- a student's ID number, address, GPA, and list of current classes
- a song's track number, title, and duration.

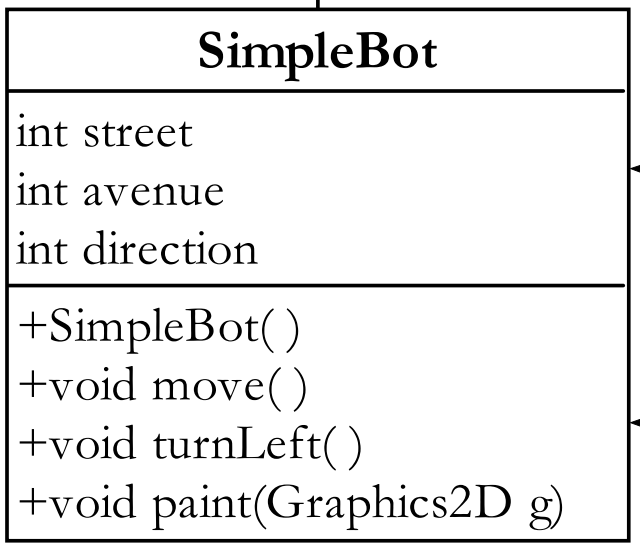
Instance variables have the following important properties:

- Each object has its own set of instance variables.
- The scope extends throughout the entire class.
- The lifetime of an instance variable is the same as the lifetime of the object to which it belongs.

A simplified version of **Robot**, called **SimpleBot**.



Override paint to determine the robot's appearance. Every object displayed in **SimpleCity** must do this.



Attributes (instance variables) to remember the street, avenue, and direction.

Methods that use and update the instance variables.

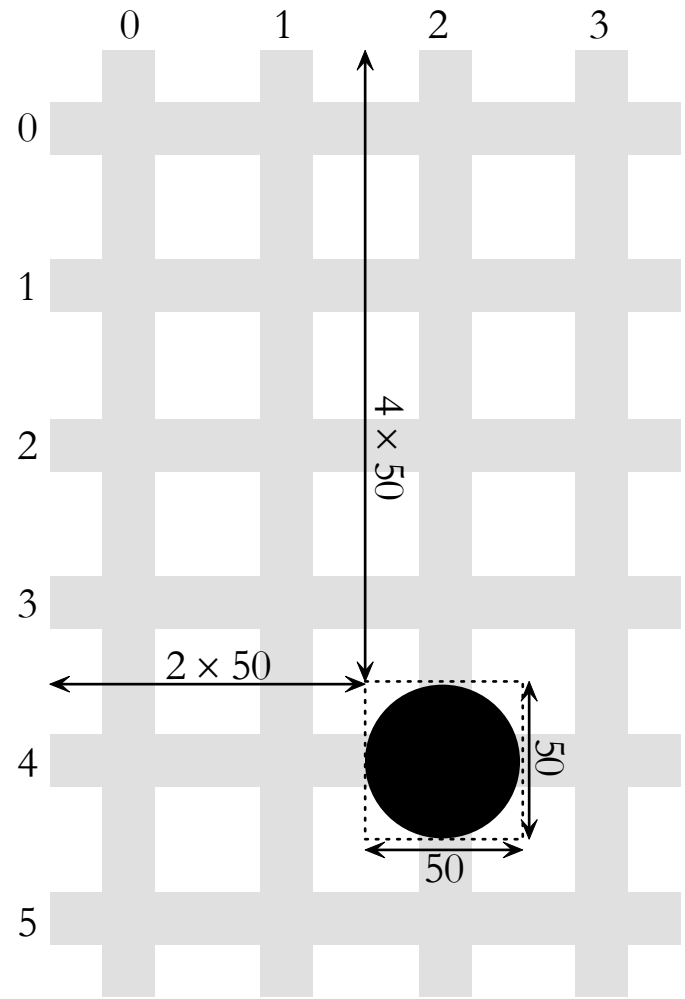
```
public class SimpleBot extends Paintable
{
   private int street = 4;       // Create space to store the robot's current street
   private int avenue = 2;       // Create space to store the robot's current avenue

   public SimpleBot()
   {  super();
   }

   // Incomplete class!
}
```

Four key parts to an instance variable declaration:

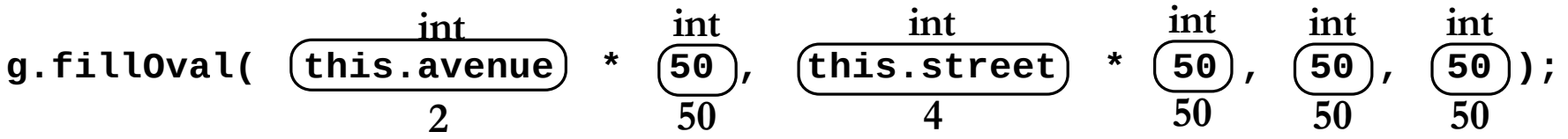
1. An access modifier; use **private** except in *rare* circumstances.
2. A type such as **int** to store integers or **String** to store a string of characters.
3. A descriptive name for the variable.
4. An initial value, placed after an equal sign.

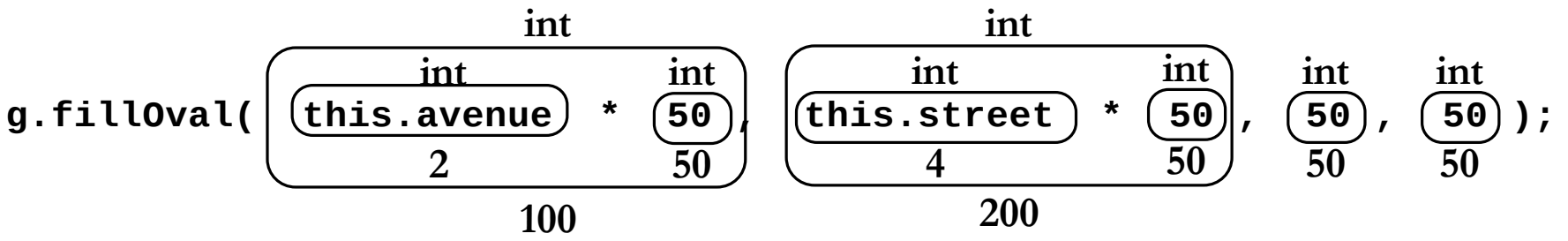
6.1.3: Accessing Instance Variables

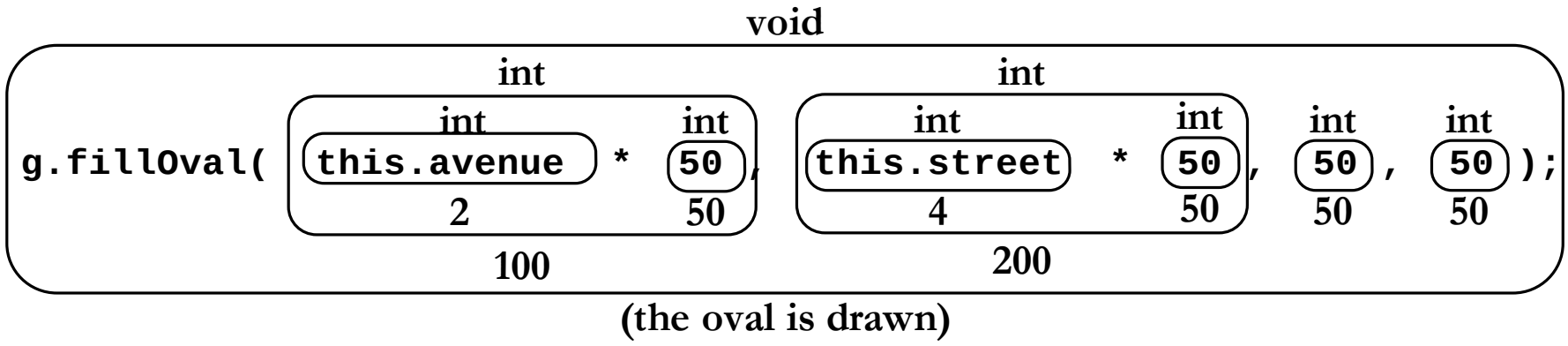
```
import java.awt.Graphics2D;
import java.awt.Color;

public class SimpleBot extends Paintable
{
   private int street = 4;          // Create space to store the robot's current street
   private int avenue = 2;          // Create space to store the robot's current avenue

   public SimpleBot()
   {  super();
   }

   public void paint(Graphics2D g)
   {  g.setColor(Color.BLACK);

      g.fillOval(100, 200, 50, 50);

      g.fillOval(2 * 50, 4 * 50, 50, 50);

      g.fillOval(this.avenue * 50,
                 this.street * 50, 50, 50);
   }
}
```



6.1.3: Accessing Instance Variables







(the oval is drawn)

6.1.4: Modifying Instance Variables

```
import java.awt.Graphics2D;
import java.awt.Color;

public class SimpleBot extends Paintable
{
   private int street = 4;          // Create space to store the robot's current street
   private int avenue = 2;          // Create space to store the robot's current avenue

   public SimpleBot()…

   public void paint(Graphics2D g)
   {  g.setColor(Color.BLACK);
      g.fillOval(this.avenue * 50, this.street * 50, 50, 50);
   }

   public void move()
   {  this.avenue = this.avenue + 1;       // Incomplete
   }

   public void turnLeft()
   { }
}
```

How does this move the robot?

SimpleCity contains a list of all the intersections, things, and **SimpleBot**s to show. It repaints the entire city about 20 times per second:

```
while (true)
{  paint everything in layer 0 (the intersections)
   paint everything in layer 1 (the things)
   paint everything in layer 2 (the robots)
   wait until 50 milliseconds have passed
}
```

When the robot moves, this code erases it from its old position and redraws it in its new position.

Problem: What if the robot moves several times within 50 milliseconds?

6.1.4: Modifying Instance Variables

```
import java.awt.Graphics2D;
import java.awt.Color;
import becker.util.Utilities;

public class SimpleBot extends Paintable
{
   private int street = 4;          // Create space to store the robot's current street
   private int avenue = 2;          // Create space to store the robot's current avenue

   public SimpleBot()…

   public void paint(Graphics2D g)
   {  g.setColor(Color.BLACK);
      g.fillOval(this.avenue * 50, this.street * 50, 50, 50);
   }

   public void move()
   {  this.avenue = this.avenue + 1;      // Incomplete
      Utilities.sleep(400);               // sleep for 400 milliseconds so user has
                                          // time to see the move
   }

   public void turnLeft()
   { }
}
```

6.1.5: Testing the SimpleBot Class

```
/** A main method to test the SimpleBot and related classes.
*
* @author Byron Weber Becker */
public class Main
{
   public static void main(String[ ] args)
   {  SimpleCity newYork = new SimpleCity();
      SimpleBot karel = new SimpleBot();
      SimpleBot sue = new SimpleBot();

      newYork.add(karel, 2);
      newYork.add(sue, 2);

      newYork.waitForStart();     // Wait for the user to press the start button.

      for(int i=0; i<4; i = i+1)
      {  karel.move();
         karel.move();
         karel.turnLeft();
      }

      sue.move();
   }
}
```

6.1.6: Adding Direction

```
…
public class SimpleBot extends Paintable
{  …
   private int direction = 0;          // Begin facing east
   …

   /** Turn the robot left 1/4 turn. */
   public void turnLeft()
   {  if (this.direction == 0)         // if facing east…
      {  this.direction = 3;           // face north
      } else
      {  this.direction = this.direction – 1;
      }
   }
}
```

6.1.6: Adding Direction

```
…
public class SimpleBot extends Paintable
{  …
   private int east = 0;
   private int south = 1;
   private int west = 2;
   private int north = 3;

   private int direction = this.east;          // Begin facing east
   …

   /** Turn the robot left 1/4 turn. */
   public void turnLeft()
   {  if (this.direction == this.east)         // if facing east…
      {  this.direction = this.north;          // face north
      } else
      {  this.direction = this.direction – 1;
      }
   }
}
```

6.1.6: Adding Direction

```
public class Constants
{
   public static final int EAST = 0;
   public static final int SOUTH = 1;
   public static final int WEST = 2;
   public static final int NORTH = 3;
}

public class SimpleBot extends Paintable
{  …

   private int direction = Constants.EAST;          // Begin facing east
   …

   /** Turn the robot left 1/4 turn. */
   public void turnLeft()
   {  if (this.direction == Constants.EAST)         // if facing east…
      {  this.direction = Constants.NORTH;          // face north
      } else
      {  this.direction = this.direction – 1;
      }
   }
}
```

6.1.6: Adding Direction

```
public class SimpleBot extends Paintable
{  …
   private int street = 4;
   private int avenue = 2;
   private int direction = Constants.EAST;          // Begin facing east
   …

   public void move()
   {  this.street = this.street + this.strOffset();
      this.avenue = this.avenue + this.aveOffset();
      Utilities.sleep(400);
   }
   private int strOffset()
   {  int offset = 0;
      if (this.direction == Constants.NORTH)
      {  offset = -1;
      } else if (this.direction == Constants.SOUTH)
      {  offset = 1;
      }
      return offset;
   }
   private int aveOffset()…
   public void turnLeft()…
}
```

An *accessor method* answers the question "What value does attribute *X* currently hold?"

In general:

```
public «typeReturned» get«Name»()
{  return this.«instanceVariable»;
}
```

For example:

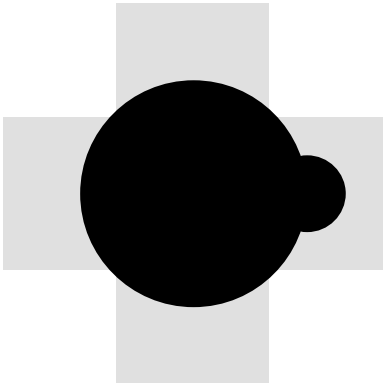
```
public class SimpleBot extends Paintable
{
   private int avenue = 2;
   …

   public int getAvenue()
   {  return this.avenue;
   }
   …
}
```

6.1.8: Instance Variables vs. Other Variables

Instance variables, temporary variables, and parameter (variables) all store information. Instance variables are different in the following ways.

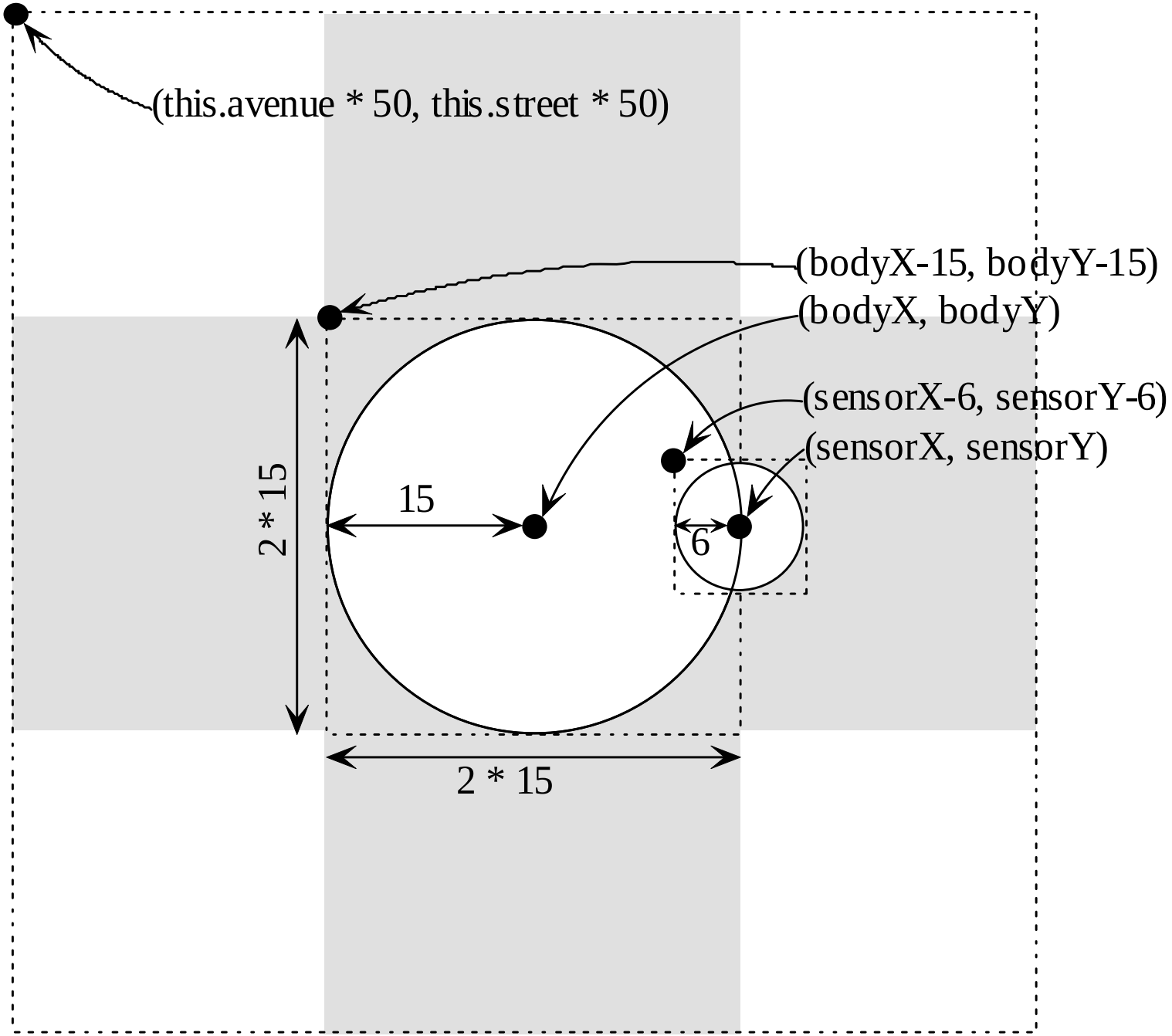
- Instance variables are declared inside a class but outside of all methods. Parameter and temporary variables are declared inside a method.
- Instance variables have a larger scope – the entire class. Parameter and temporary variables have a scope no larger than a method.
- Instance variables have a longer lifetime – the same as the object that contains them. Parameter and temporary variables disappear when their method finishes executing.

We need to enhance the **paint** method to show the direction the robot is facing. We'll do this by adding a "sensor" to the front of the robot.

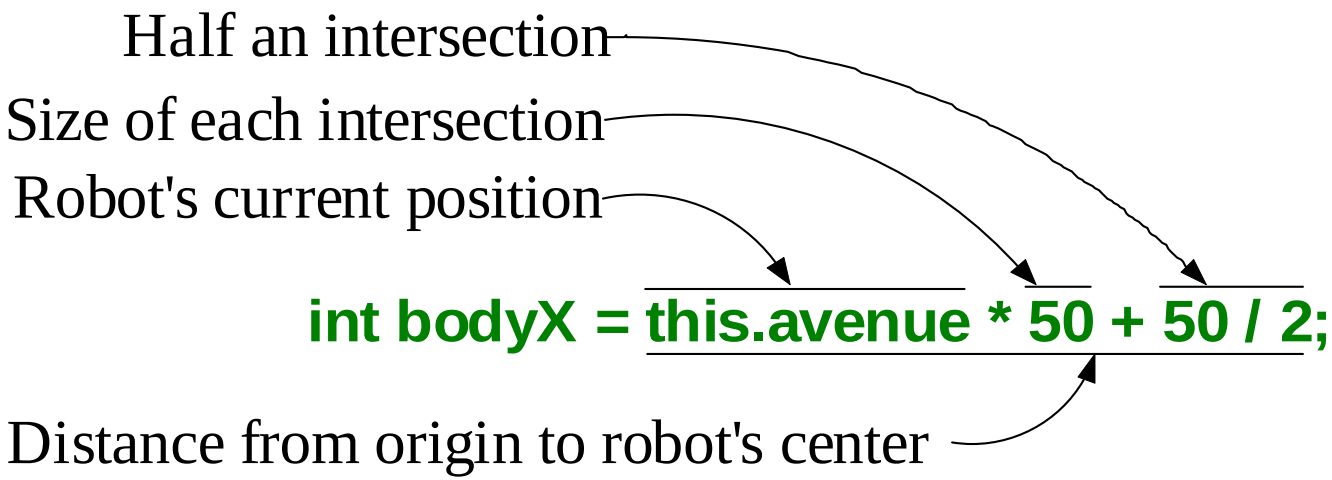


```
public void paint(Graphics2D g)
{  g.setColor(Color.BLACK);

   int bodyX = x coordinate of robot body's center
   int bodyY = y coordinate of robot body's center
   int sensorX = x coordinate of robot sensor's center
   int sensorY = y coordinate of robot sensor's center

   // Draw the robot's body
   g.fillOval(bodyX – 15, bodyY – 15, 2 * 15, 2 * 15);

   // Draw the robot's sensor
   g.fillOval(sensorX – 6, sensorY – 6, 2 * 6, 2 * 6);
}
```

Case Study 1: Calculating Values



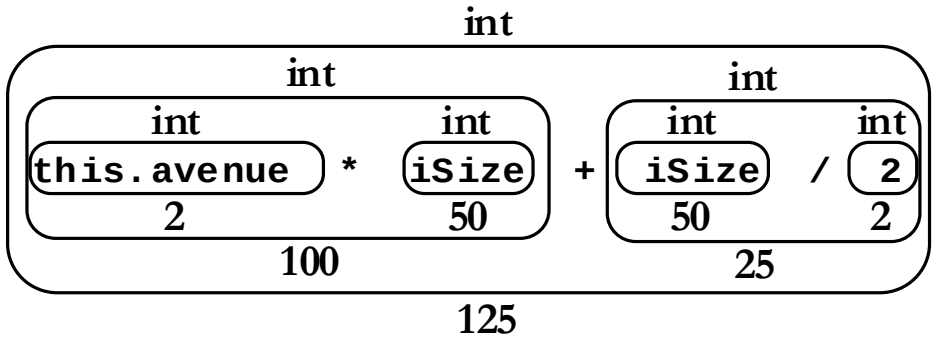
Case Study 1: Calculating bodyX, bodyY



Replace **50** with a more meaningful (but short) name:

```
int iSize = Constants.INTERSECTION_SIZE;
int bodyX = this.avenue * iSize + iSize / 2;
```

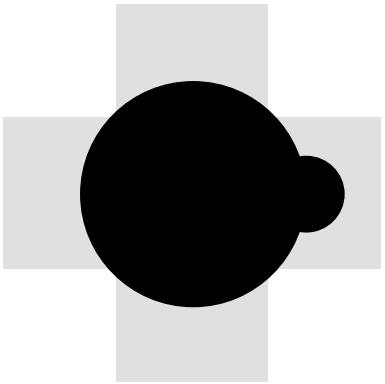
Assuming the robot is on Avenue 2, we have:



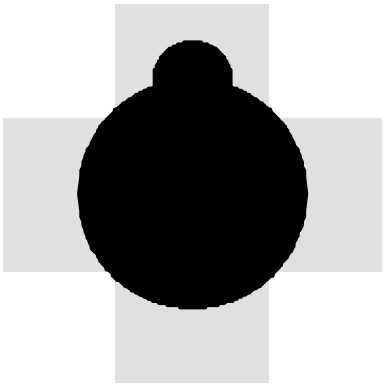
Case Study 1: Drawing the Body

```
public void paint(Graphics2D g)
{  g.setColor(Color.BLACK);

   int iSize = Constants.INTERSECTION_SIZE;
   int bodyX = this.avenue * iSize + iSize / 2;
   int bodyY = this.street * iSize + iSize / 2;
   int sensorX = x coordinate of robot sensor's center
   int sensorY = y coordinate of robot sensor's center

   // Draw the robot's body
   g.fillOval(bodyX – 15, bodyY – 15, 2 * 15, 2 * 15);

   // Draw the robot's sensor
   g.fillOval(sensorX – 6, sensorY – 6, 2 * 6, 2 * 6);
}
```

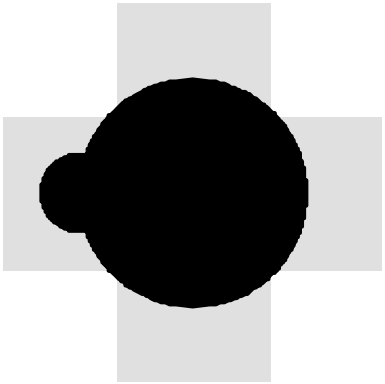
Case Study 1: Calculating sensorX, sensorY



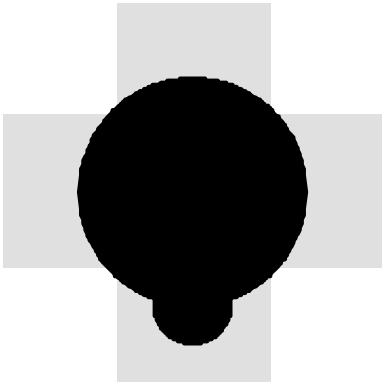
```
sensorX = bodyX + 15;
sensorY = bodyY;
```



```
sensorX = bodyX;
sensorY = bodyY – 15;
```



```
sensorX = bodyX - 15;
sensorY = bodyY;
```



```
sensorX = bodyX;
sensorY = bodyY + 15;
```

In general:

```
int sensorX = bodyX + this.aveOffset() * 15;
int sensorY = bodyY + this.strOffset() * 15;
```

Consider the following "family" of **move** methods:

```
public class SimpleBot extends Paintable
{
   private int street = 4;
   private int avenue = 2;
   …
   public void move()…

   public void moveFar()
   {  int howFar = 2;
      for(int i = 0; i < howFar; i = i + 1)
      {  this.move();
      }
   }
   public void moveReallyFar()
   {  int howFar = 3;
      for(int i = 0; i < howFar; i = i + 1)
      {  this.move();
      }
   }
}
```

Case Study 2: Implementing a Parameter

Without Parameters

```
public static void main(…)
{  SimpleBot r = new SimpleBot();
   …
   r.moveFar();
   r.moveReallyFar();
}
```

```
public class SimpleBot…
{  public void move()
   {  …
   }
   public void moveFar()
   {  int howFar = 2;
      for (int i = 0; i < howFar; i=i+1)
      {  this.move();
      }
   }
   public void moveReallyFar()
   {  int howFar = 3;
      for (int i = 0; i < howFar; i=i+1)
      …
   }
}
```

With Parameters

```
public static void main(…)
{  SimpleBot r = new SimpleBot();
   …
   r.moveFar(2);
   r.moveFar(3);
}
```

```
public class SimpleBot…
{  public void move()
   {  …
   }
   public void moveFar(int howFar)
   {
      for (int i = 0; i < howFar; i=i+1)
      {  this.move();
      }
   }
}
```

Case Study 2: Overloading Methods

```
public class Main
{
   public static void main(…)
   {  …
      SimpleBot sb = new Simple…;
      …

      sb.move();
      sb.move(5);
      sb.move(0, 0);
   }
}
```

```
public class SimpleBot…
{
   …
   public void move()
   {  …
   }

   public void move(int howFar)
   {  …
   }

   public void move(int str, int ave)
   {  …
   }
}
```

In each case, *we* can tell which **move** method to use. So can Java!

This is called *overloading*: when two or more methods have the same name and return type, but parameter lists (different number of parameters or different orders to the types).

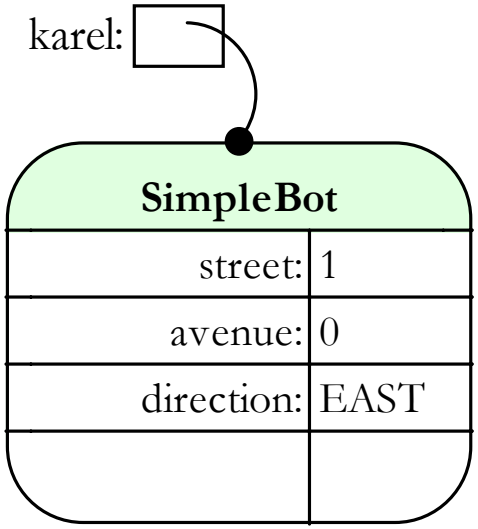
Case Study 2: Parameters in Constructors

Parameters are often used in constructors to initialize instance variables:

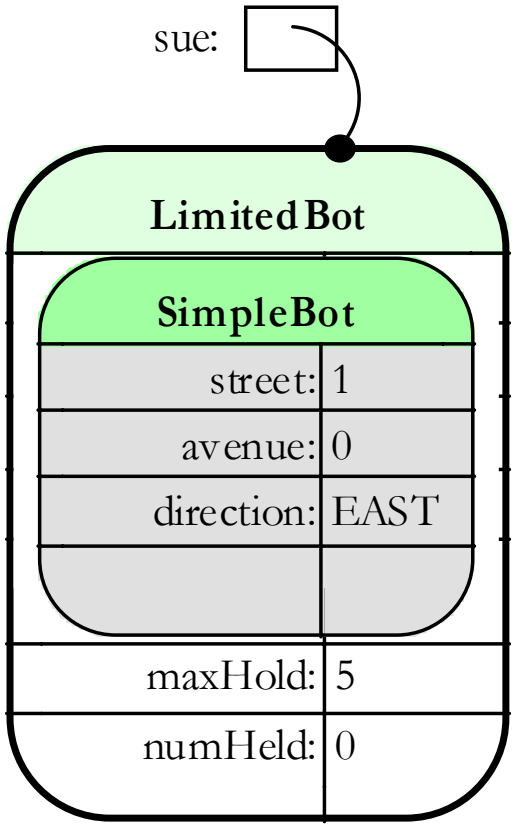
```
public class SimpleBot extends Paintable
{
   private int street = 4;
   private int avenue = 2;
   private int direction = Constants.EAST;

   public SimpleBot(int aStreet, int anAvenue, int aDirection)
   {  super();
      this.street = aStreet;
      this.avenue = anAvenue;
      this.direction = aDirection;
   }
}
```

Imagine a special kind of robot, called a **LimitedBot** that can hold only a limited number of things. Such a robot needs two additional pieces of information (attributes):

- How many things can I hold?
- How many things am I currently holding?



SimpleBot object



LimitedBot object

```
public class LimitedBot extends SimpleBot
{
   private int maxHold;          // How many things can this robot hold?
   private int numHeld = 0;      // How many things is this robot currently holding?

   public LimitedBot(City aCity, int aStr, int anAve, Direction aDir,
                     int maxCanHold)
   {  super(aCity, aStr, anAve, aDir);

      this.maxHold = maxCanHold;
   }
}
```

Must match the signature of a constructor in the superclass to initialize the instance variables in "Robot within this robot."

Initialize the instance variables in this object.

```
public class LimitedBot extends SimpleBot
{
   private int maxHold;          // How many things can this robot hold?
   private int numHeld = 0;      // How many things is this robot currently holding?

   public LimitedBot(City aCity, int aStr, int anAve, Direction aDir, int maxCanHold)
   {  super(aCity, aStr, anAve, aDir);
      this.maxHold = maxCanHold;
   }

   public void pickThing()
   {  if (this.numHeld == this.maxHold)
      {  this.breakRobot("Tried to pick up too many things.");
      } else
      {  super.pickThing();
         this.numHeld = this.numHeld + 1;
      }
   }

   public void putThing()
   {  super.putThing();
      this.numHeld = this.numHeld – 1;
   }
}
```

To create a robot with a limited carrying capacity, we could

- extend **SimpleBot** (as we did).
- modify the code for **SimpleBot**
- make a copy of **SimpleBot**, rename it, and modify that code

Which is best?

6.5.1: Comparing Kinds of Variables

All variables store a value. Differences are highlighted below.

	Instance Var.	**Temp. Var.**	**Param. Var.**	**Constant**
Are declared…	in a class but outside methods.	inside a method.	in the method's parameter list.	in a class but outside methods.
Can be used…	in any method in the class.	in the method where declared.	in the method where declared.	in any method in the class.
Are initialized …	in the declaration or the constructor.	in the declaration.	where the method is called.	in the declaration.
Values are stored…	until changed or the object is no longer used.	until changed or the block is finished executing.	until changed or the method is finished executing.	as long as the program is executing.
The visibility modifier…	should always be **private**.	is not applicable.	is not applicable.	may be **public** or **private**.

6.5.2: Comparing Kinds of Variables

If you…	Then…
need a value that never changes	use a **final** instance variable (constant).
need to store a value for use later in the method but then discarded	use a temporary variable.
have a method that needs a value provided by the client	use a parameter variable.
find yourself writing almost identical code several times	look for a way to put the code in a method, accounting for the differences with parameters.
need a value within many methods within a class	consider using an instance variable.
need to implement an attribute	use an instance variable or calculate it based on existing instance variables.
must store a value even with no service is being used	use an instance variable.

```
public class CounterBot1…
{  private int count = 0;

   public int numIntersections()
   {
      while (true)
      {  if (this.canPickThing())
         {  this.count = this.count + 1;
         }
         if (this.frontIsClear())
         {  break;
         }
         this.move();
      }
      return this.count;
   }
}
```

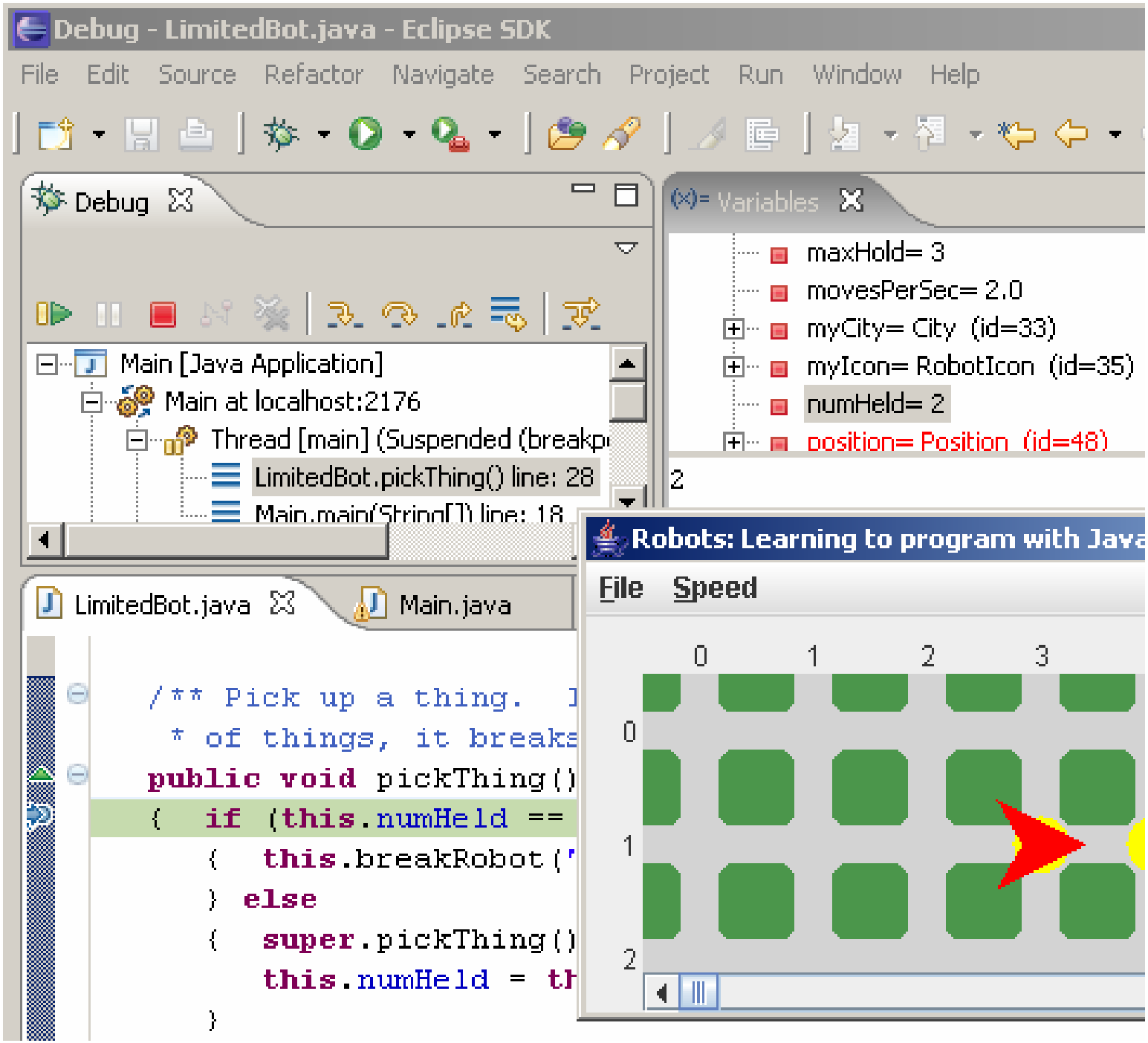
```
public class CounterBot2…
{

   public int numIntersections()
   {  int count = 0;
      while (true)
      {  if (this.canPickThing())
         {  count = count + 1;
         }
         if (this.frontIsClear())
         {  break;
         }
         this.move();
      }
      return count;
   }
}
```

Every temporary variable can be replaced with an instance variable. Does it matter which you choose? Why?

Printing the value of a variable or expression is often helpful while debugging.

```
public class LimitedBot extends SimpleBot
{  private int maxHold;          // How many things can this robot hold?
   private int numHeld = 0;      // How many things is this robot currently holding?
   ...

   public void pickThing()
   {  System.out.print("PickThing: numHeld=");
      System.out.println(this.numHeld);

      if (this.numHeld == this.m
      {  this.breakRobot("Tried to
      } else
      {  super.pickThing();
         this.numHeld = this.nur
      }
   }
}
```

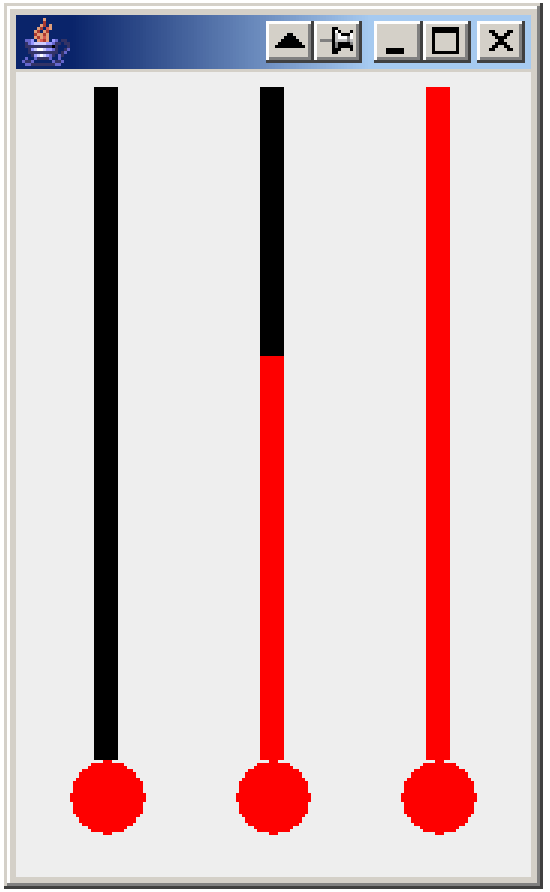
6.6.2: Using a Debugger



Application: Repainting

A graphical user interface often shows a graphical representation of a numerical value – such as the thermometer showing the temperature.

This frame shows three instance of **Thermometer**.



How might the following be used in this program?

- Instance variables
- Temporary variables
- Parameter variables
- Constants

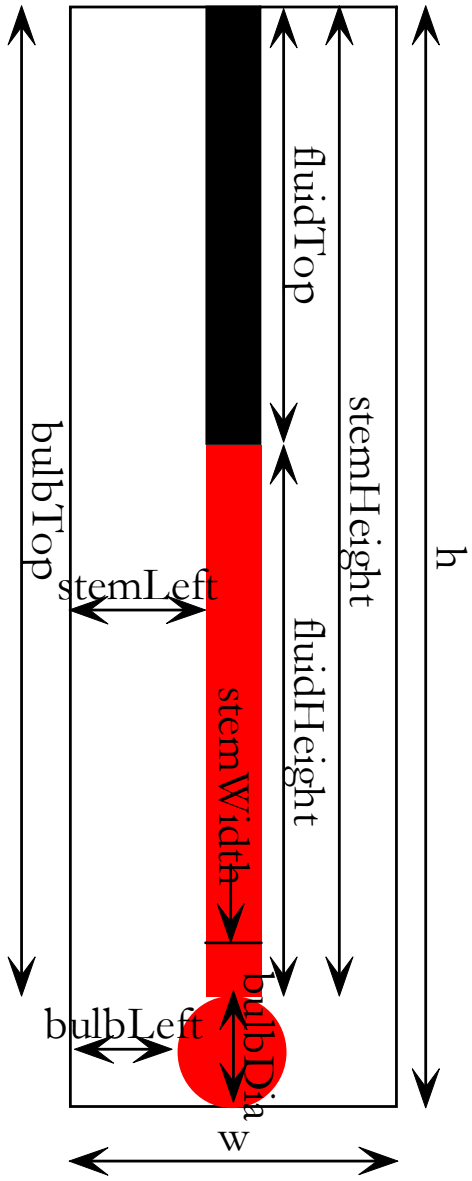
Application: main method

```
public static void main(String[ ] args)
{  // Create three thermometer components.
   Thermometer t0 = new Thermometer();
   Thermometer t1 = new Thermometer();
   Thermometer t2 = new Thermometer();

   // Create a panel to hold the thermometers.
   JPanel contents = new JPanel();
   contents.add(t0);
   contents.add(t1);
   contents.add(t2);

   // Set up the frame.
   JFrame f = new JFrame();
   f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   f.setContentPane(contents);
   f.pack();
   f.setVisible(true);

   // Set the temperature of each thermometer.
   t0.setTemperature(0);
   t1.setTemperature(30);
   t2.setTemperature(50);
}
```

```
public class Thermometer extends JComponent
{
   public final int MIN_TEMP = 0;
   public final int MAX_TEMP = 50;
   private int temp = MIN_TEMP;

   public void paintComponent(Graphics g)…

   /** Set the thermometer's temperature.
   *  @newTemp The new temperature. */
   public void setTemperature(                    )
   {
      
   }

   /** Get the thermometer's current temperature.
   *  @returns The thermometer's current temperature. */
   public int getTemperature()
   {
      
   }
}
```

Temporary Variables

```
public class Thermometer extends JComponent
{  public final int MIN_TEMP = 0;
   public final int MAX_TEMP = 50;
   private int temp = MIN_TEMP;

   public void paintComponent(Graphics g)
   {  super.paintComponent(g);

      final int w = this.getWidth();
      final int h = this.getHeight();

      final int bulbDia = h/10;
      final int bulbLeft = w/2 - bulbDia/2;
      final int bulbTop = h - bulbDia;

      final int stemWidth = bulbDia/3;
      final int stemLeft = w/2 - stemWidth/2;
      final int stemHeight = h - bulbDia;

      final int fluidHeight = stemHeight *
                  (this.temp - MIN_TEMP) / (MAX_TEMP - MIN_TEMP);
      final int fluidTop = stemHeight - fluidHeight;
      ...
```



Temporary Variables

```
        // paint the fluid
        g.setColor(Color.RED);
        g.fillOval(bulbLeft, bulbTop, bulbDia, bulbDia);
        g.fillRect(stemLeft, fluidTop, stemWidth, fluidHeight);

        // paint the stem above the fluid
        g.setColor(Color.BLACK);
        g.fillRect(stemLeft, 0, stemWidth, fluidTop);
    }

    public void setTemperature(                    )
    {
                                                    
        this.repaint();
    }
}
```

6.8.1: The Named Constant Pattern

Name: Named Constant

Context: A special, unchanging value that is known when the program is written is used one or more times in a program.

Solution: Use a named constant, for example:

```
private static final int DAYS_IN_WEEK = 7;
private static final int COST_PER_MOVE = 25;
```

In general,

```
«accessModifier» static final «type» «name» = «value»;
```

Consequences: Programs become more self-documenting when special values are given meaningful names.

Related Patterns: This pattern is a specialization of the Instance Variable pattern. When constants are used to distinguish a set of values, such as the four directions or **MALE** and **FEMALE**, the Enumeration pattern is often a better choice.

6.8.2: The Instance Variable Pattern

Name: Instance Variable

Context: An object needs to maintain a value. It must be remembered for longer than one method call and is usually required in more than one method.

Solution: Use an instance variable declared inside the class but outside of all methods. For example,

```
private int numMoves = 0;
private int currentAve;
```

An instance variable is declared with one of two general forms:

```
«accessModifer» «type» «name» = «initialValue»;
«accessModifer» «type» «name»;
```

where *«accessModifer»* is usually **private**. The *«type»* in these examples is **int** but may be others such as **double**, **boolean**, or a reference type.

Consequences: The variable stores a value for the lifetime of the object. It may be changed with an assignment statement.

Related Patterns: Temporary Variable; Named Constant

6.8.3: The Accessor Method Pattern

Name: Accessor Method

Context: A class has private instance variables to protect them from misuse. However, clients still need to access their values.

Solution: Provide a public query using the following template:

```
public «typeReturned» get«name»()
{ return this.«instanceVariable»;
}
```

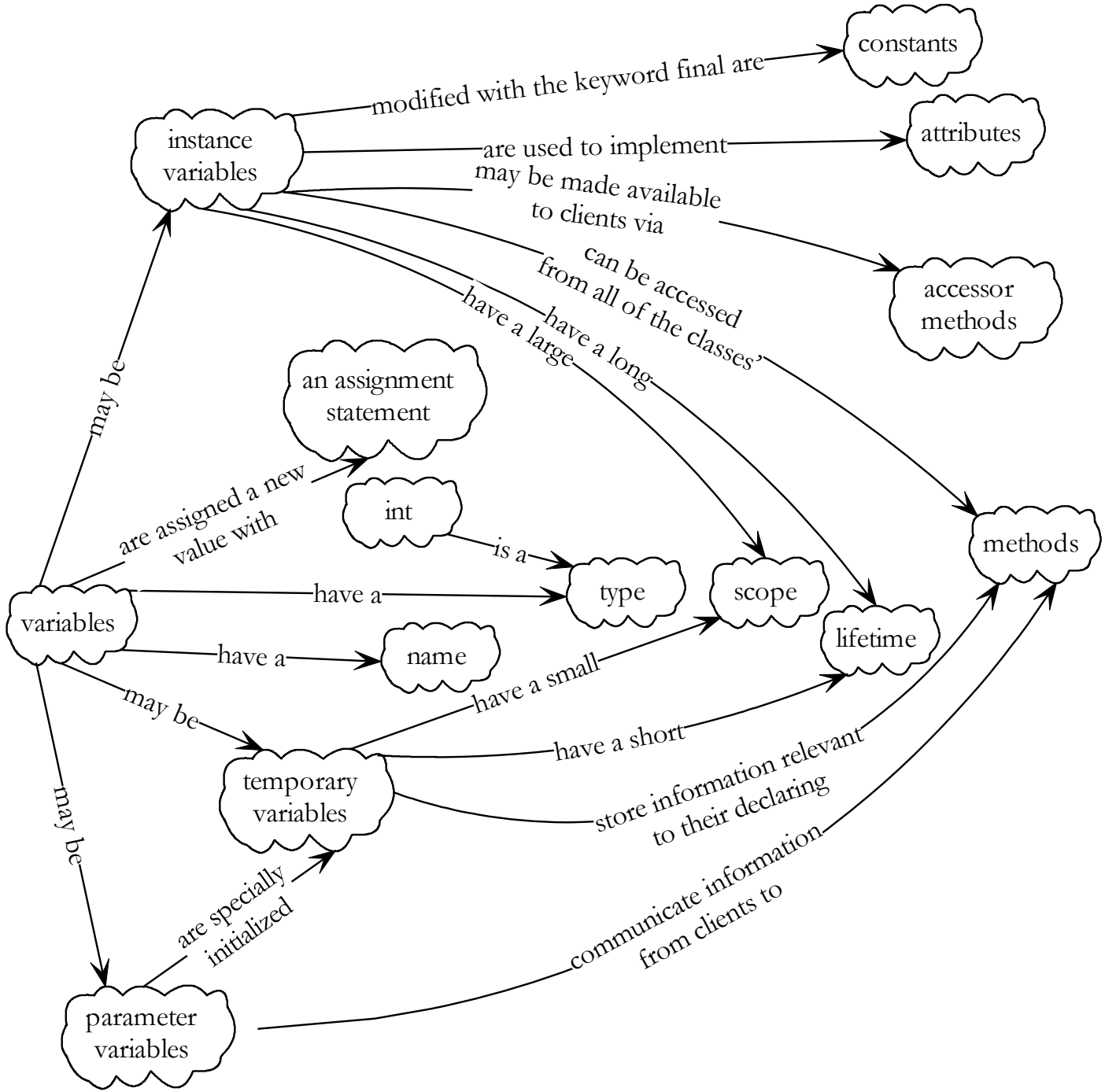
For example,

```
public class SimpleBot...
{ private int street;
  ...
  public int getStreet()
  { return this.street;
  }
}
```

Consequences: Restricted access is provided to an instance variable.

Related Patterns: This pattern is a specialization of the Query pattern.

6.9: Concept Map



Summary

We have learned:

- how to use instance variables to implement the attributes of a class.
- that instance variables are similar to temporary and parameter variables in that they all store values, but have important differences in purpose, lifetime, and scope.
- how to initialize instance variables where they are declared or in a constructor.
- that the **final** modifier makes the first value assigned to a variable the final value so that it can't be changed.
- that the **static** modifier allows a constant to be accessed with a class name rather than an object.
- how to extend a class with additional instance variables.
- how to print the value of a variable and view it with a debugger.
- that one must call **repaint** after changing an instance variable that affects a component's appearance.