Java:
Learning to Program with Robots

Chapter 06: Using Variables
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.

**Paintable**

| +void paint(Graphics2D g) |

**SimpleBot**

| int street |
| int avenue |
| int direction |

| +SimpleBot( ) |
| +void move( ) |
| +void turnLeft( ) |
| +void paint(Graphics2D g) |

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.
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
6.1.3: Accessing Instance Variables

```java
void (the oval is drawn)
```
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:

```java
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?
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() { }
}

/** 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.move();
            karel.turnLeft();
        }

        sue.move();
    }
}
... 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;
     }
  }
}
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;
            }
        }
    }
}
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;
        }
    }
}
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:

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

For example:

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

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

```java
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

- (this.avenue * 50, this.street * 50)
- (bodyX, bodyY)
- (sensorX-6, sensorY-6)
- (sensorX, sensorY)

Mathematical Expressions:
- 2 * 15
- 2 * 15
Case Study 1: Calculating bodyX, bodyY

Half an intersection
Size of each intersection
Robot's current position
Distance from origin to robot's center

int bodyX = this.avenue * 50 + 50 / 2;

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:

\[
\begin{array}{c|c}
\text{int} & \text{int} \\
\hline
\text{this.avenue} & \text{iSize} \\
2 & 50 \\
\hline
100 & 25 \\
\hline
125 &
\end{array}
\]
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 - 15;
sensorY = bodyY;
```

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

```
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:

```java
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();
        }
    }
}
```
<table>
<thead>
<tr>
<th>Without Parameters</th>
<th>With Parameters</th>
</tr>
</thead>
</table>
| 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)
    { ...  
    }
  }
} | 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();
    }
  }
} |
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).
Parameters are often used in constructors to initialize instance variables:

```java
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

<table>
<thead>
<tr>
<th>street</th>
<th>avenue</th>
<th>direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>EAST</td>
</tr>
</tbody>
</table>

**LimitedBot** object

<table>
<thead>
<tr>
<th>street</th>
<th>avenue</th>
<th>direction</th>
<th>maxHold</th>
<th>numHeld</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>EAST</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
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?
All variables store a value. Differences are highlighted below.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Are declared…</td>
<td>in a class but outside methods.</td>
<td>inside a method.</td>
<td>in the method’s parameter list.</td>
<td>in a class but outside methods.</td>
</tr>
<tr>
<td>Can be used…</td>
<td>in any method in the class.</td>
<td>in the method where declared.</td>
<td>in the method where declared.</td>
<td>in any method in the class.</td>
</tr>
<tr>
<td>Are initialized …</td>
<td>in the declaration or the constructor.</td>
<td>in the declaration.</td>
<td>where the method is called.</td>
<td>in the declaration.</td>
</tr>
<tr>
<td>Values are stored…</td>
<td>until changed or the object is no longer used.</td>
<td>until changed or the block is finished executing.</td>
<td>until changed or the method is finished executing.</td>
<td>as long as the program is executing.</td>
</tr>
<tr>
<td>The visibility modifier…</td>
<td>should always be <strong>private</strong>.</td>
<td>is not applicable.</td>
<td>is not applicable.</td>
<td>may be <strong>public</strong> or <strong>private</strong>.</td>
</tr>
<tr>
<td>If you…</td>
<td>Then…</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>need a value that never changes</td>
<td>use a <strong>final</strong> instance variable (constant).</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>need to store a value for use later in the method but then discarded</td>
<td>use a temporary variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>have a method that needs a value provided by the client</td>
<td>use a parameter variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>find yourself writing almost identical code several times</td>
<td>look for a way to put the code in a method, accounting for the differences with parameters.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>need a value within many methods within a class</td>
<td>consider using an instance variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>need to implement an attribute</td>
<td>use an instance variable or calculate it based on existing instance variables.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>must store a value even with no service is being used</td>
<td>use an instance variable.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

```java
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.maxHold) {
            this.breakRobot("Tried to pick up too many things.");
        } else {
            super.pickThing();
            this.numHeld = this.numHeld + 1;
        }
    }
}
```
6.6.2: Using a Debugger

```java
/** Pick up a thing. If the robot has already picked up a thing, it breaks.
 * of things, it breaks.
 */

public void pickThing()
{
    if (this.numHeld ==
    {
        this.breakRobot('t');
    } else
    {
        super.pickThing();
        this.numHeld = t;
    }
}
```
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
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) {
        // Paint the thermometer
    }

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

    /** Get the thermometer's current temperature. */
    public int getTemperature() {
    }
}
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;
        ...
}
// 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);
g.fillRect(stemLeft, 0, stemWidth, fluidTop);

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

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

In general,

```java
<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.
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,

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

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

```java
<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
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:

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

For example,

```java
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.
instance variables

variables

temporary variables

parameter variables

may be made available to clients via temporary variables methods

may be used to implement communication information from clients to parameter variables

may be accessed from all of the classes' variable methods

have a large lifetime

may be initialized

have a small scope

have a short lifetime

have a long lifetime

have a name

have a type

int is a type

an assignment statement

are assigned a new value

are specially initialized

modified with the keyword final are constants

have a parameter

int is a parameter

are used to implement accessor methods

store information relevant to their declaring

communicate information from clients to

are used to implement accessor methods
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.