Java:

Learning to Program with Robots

Chapter 04: Making Decisions

Chapter Objectives

After studying this chapter, you should be able to:

- Use an **if** statement to perform an action once or not at all.
- Use a **while** statement to perform an action zero or more times.
- Use an **if-else** statement to perform either one action or another action.
- Describe what conditions can be tested and how to write new tests.
- Write a method, called a predicate, that can be used in the test of an **if** or **while** statement.
- Use parameters to communicate values from the client to be used in the execution of a method.
- Use a **while** statement to perform an action a specified number of times.

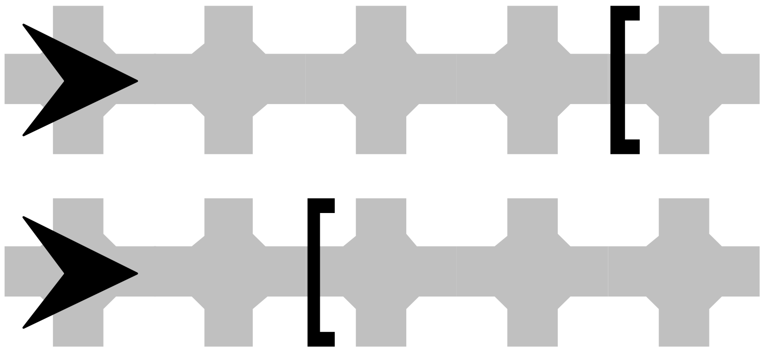
4.1: Understanding Two Kinds of Decisions

So far, programs have

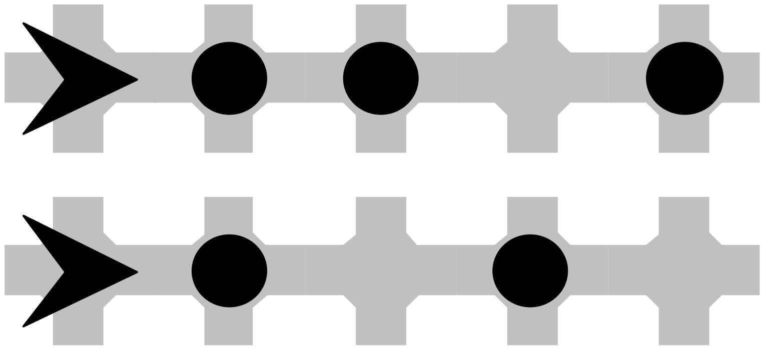
- executed one statement after another (sequential execution)
- executed all the statements in a method, and then returned

Examples of problems that can't be solved this way:

- Move to a wall when it's not known how far away the wall is.

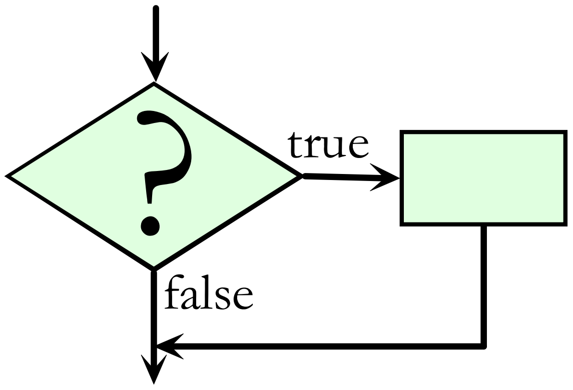
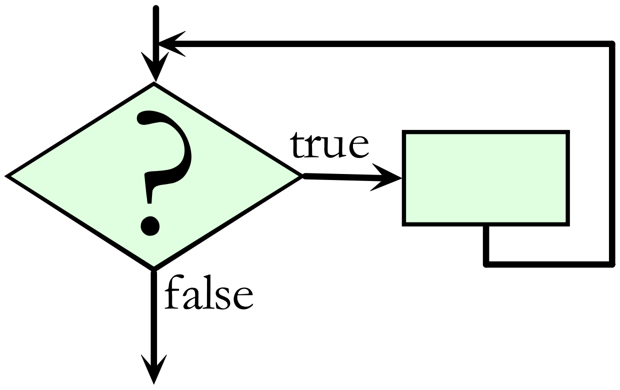


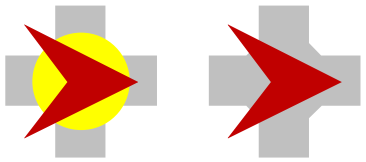
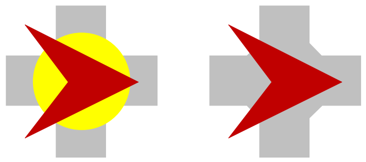
- Pick up all the **Thing**s in a row where some intersections have nothing on them.



In each case, use the *same* program to solve the different variations of the same problem. This requires our programs to make decisions.

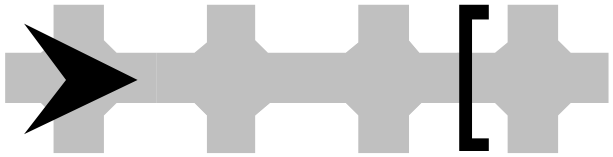
4.1: Understanding Two Kinds of Decisions

	if statement	**while** statement
Question:	Should this group of statements be executed *once or not at all*?	Should this group of statements be executed *again*?
Flow-chart:		

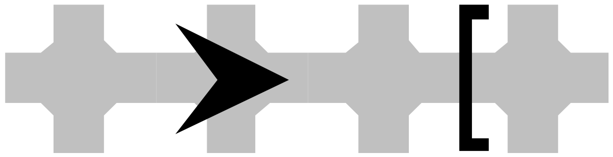
Example:	
	if (karel.canPickThing())
	{ karel.turnLeft();
	}
	karel.move();
		
		while (karel.canPickThing())
		{ karel.turnLeft();
		}
		karel.move();

4.1.2: Examining an if Statement (1/2)

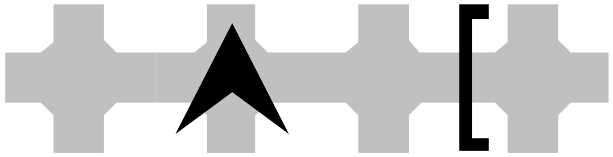
```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```



```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

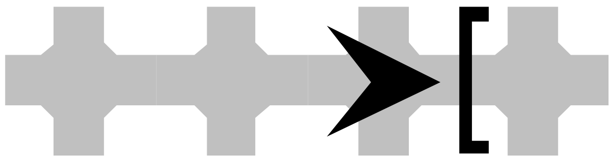


```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

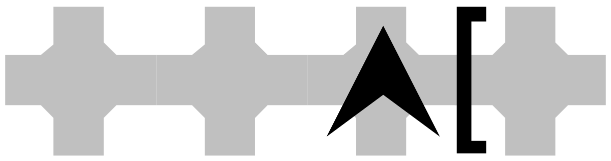


4.1.2: Examining an if Statement (2/2)

```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

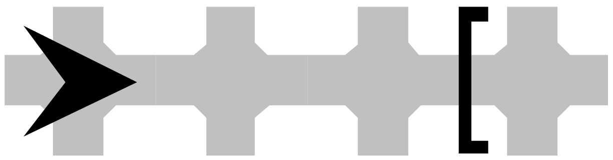


```
if (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

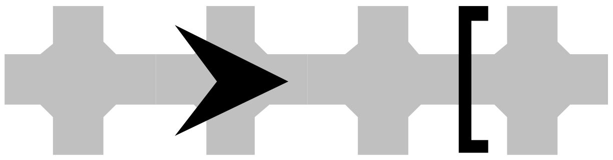


4.1.3: Examining a while Statement (1/2)

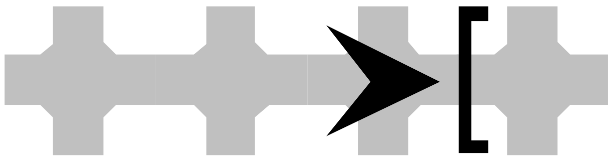
```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```



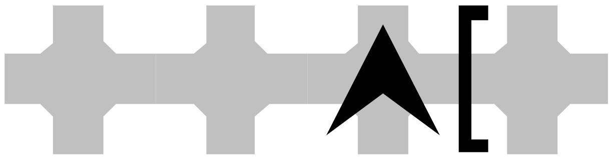
```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```



```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

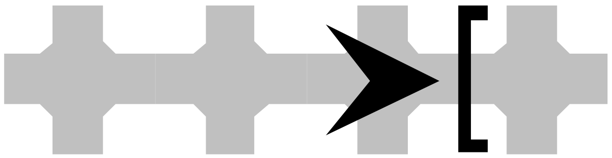


```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```

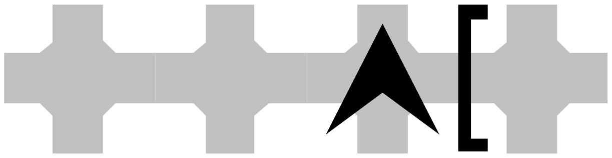


4.1.3: Examining a while Statement (2/2)

```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```



```
while (karel.frontIsClear())
{ karel.move();
}
karel.turnLeft();
```



4.1.4: The General Forms of if and while

The General Form of an **if** Statement:

```
if («test»)
{ «list of statements»
}
```

Examples:

```
if (karel.canPickThing())
{  karel.pickThing();
   karel.turnLeft();
}

if (this.frontIsClear())
{  this.move();
}
```

The General Form of a **while** Statement:

```
while («test»)
{ «list of statements»
}
```

Examples:

```
while (karel.canPickThing())
{  karel.pickThing();
   karel.turnLeft();
}

while (this.frontIsClear())
{  this.move();
}
```

4.2: Questions Robots Can Ask

Can I pick up a **Thing** from this intersection?

How many **Thing**s are in my backpack?

Is the path in front of me clear of obstructions (like walls)?

Which avenue am I on?

Which direction am I facing?

What string of characters is labelling me?

What is my speed?

What street am I on?

Questions with a **true** or **false** answer, like **canPickThing**, are called *predicates*.

Robot
int street int avenue Direction direction ThingBag backpack
+Robot(City aCity, int aStreet, int anAvenue, Direction aDirection) +boolean canPickThing( ) +int countThingsInBackpack( ) +boolean frontIsClear( ) +int getAvenue( ) +Direction getDirection( ) +String getLabel( ) +double getSpeed( ) +int getStreet( )

4.2.2: Negating Predicates

Negating a predicate gives it the opposite meaning.

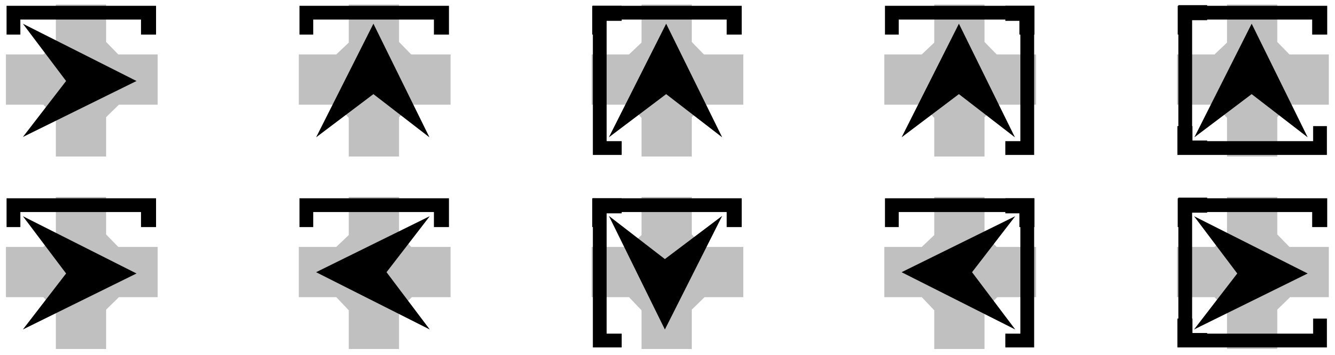
```
// in pseudocode
if (karel cannot pick a thing)
{  put a thing down
}
```

```
// in Java
if (!karel.canPickThing())
{  karel.putThing();
}
```

Exercise: Karel is moving through an area where it might be blocked by a wall. Each time it is blocked, it should turn left to find a direction in which it can move. There will always be at least one such direction. Here are some possible situations:

Initial Situations

Final Situations



Write a code fragment to implement this behavior.

4.2.3: Testing Integer Queries

Queries like **getAvenue** do not return **true** or **false**, they return an integer such as 0 or 3. Can they be used in an **if** or **while** statement?

```
if (karel.getStreet() == 1)
{  karel.turnAround();
}
```

- What happens if **karel** is on 2nd Street?
- What happens if **karel** is on 1st Street?

```
while (karel.countThingsInBackpack() < 4)
{  karel.pickThing();
}
```

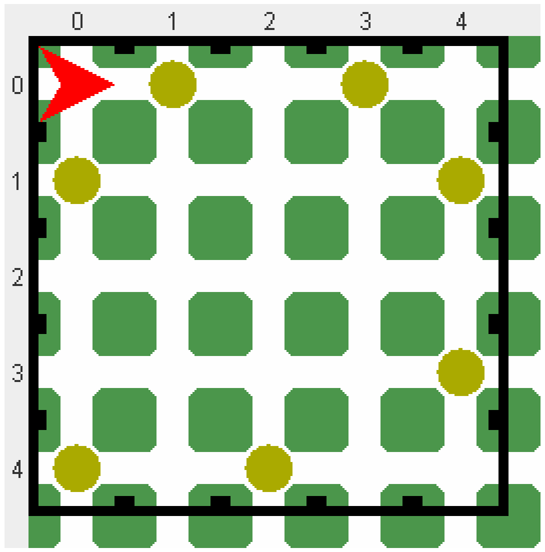
- What happens if **karel** already has 2 **Thing**s in its backpack?
- What happens if **karel** already has 6 **Thing**s in its backpack?

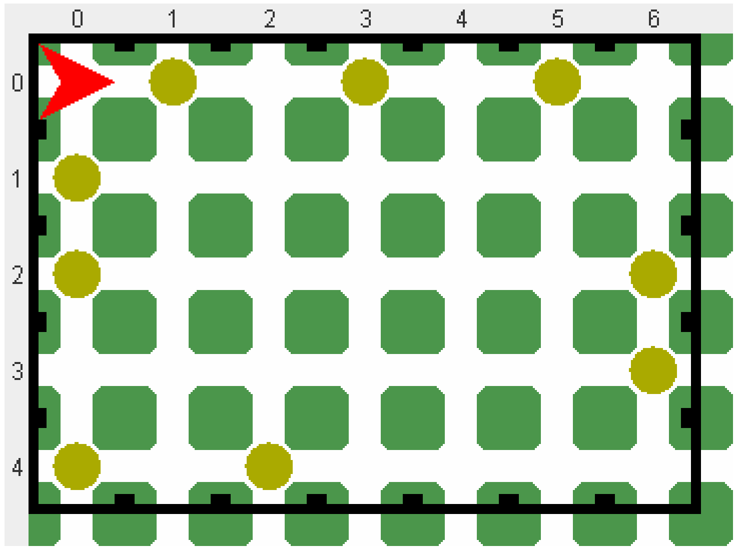
Comparison Operators:

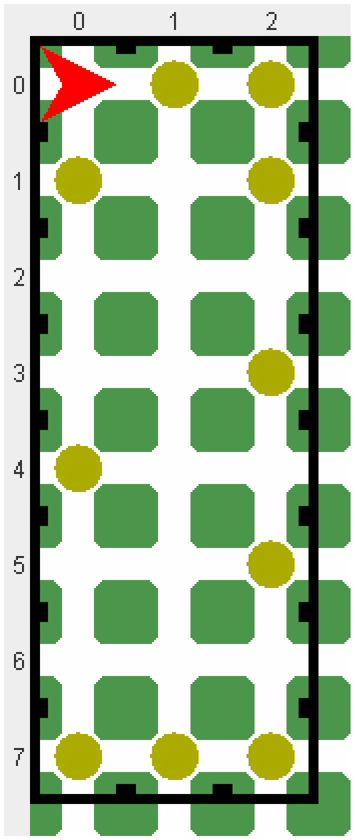
<	less than	**>**	greater than	**==**	equal
<=	less than or equal	**>=**	greater than or equal	**!=**	not equal

Case Study: Collecting Trash

Some irresponsible people have been pitching their trash over the fence and into your yard. Proud of your property's appearance, you develop a specialized robot to pick up the trash. Knowing your neighbors have similar problems, you design the robot to collect garbage from fenced yards with arbitrary dimensions. The only restriction is that they are rectangular and the northwest corner is at (0, 0). The trash is always beside the fence and consists of a single **Thing**.







Case Study: The main Method

```
import becker.robots.*;

/** Collect the trash from a rectangular fenced yard of arbitrary size.
*
* @author Byron Weber Becker */
public class CollectTrash
{
   public static void main(String[ ] args)
   {
   City yard = new City("yard1.txt");
   TrashBot karel = new TrashBot(yard);

   karel.collectTrash();
   }
}
```

The **City** class can read a file to tell where to put **Walls** and **Things**. This allows easy testing with several different yards.

We'll always create the **TrashBot** at (0, 0) facing East. All we need to provide here is the city.

Case Study: Configuring the City

```
# A City with a fenced yard and trash.

# Window title
Robots: Learning to program with Java
# first street, first avenue, num streets, num avenues showing
0 0 7 7
# intersection size (in pixels)
48

# North fence
becker.robots.Wall 0 0 NORTH
becker.robots.Wall 0 1 NORTH
becker.robots.Wall 0 2 NORTH
becker.robots.Wall 0 3 NORTH
becker.robots.Wall 0 4 NORTH

# Similar for South, East, and West fences

# Trash
becker.robots.Thing 0 1
becker.robots.Thing 0 3
becker.robots.Thing 4 2
becker.robots.Thing 1 4
becker.robots.Thing 3 4
```

Comments begin with #.

A title, information about which roads to show, and how large to make intersections is required.

Objects to add to the city. Each line has the complete class name of the object to create and values corresponding to one of its constructors. It is assumed the first argument to the constructor is the city.

Case Study: TrashBot (1/6)

```
import becker.robots.*;

/** A robot that collects trash along a rectangular fenced yard.
*
* @author Byron Weber Becker */
public class TrashBot extends RobotSE
{
      /** Construct a TrashBot at (0,0) facing East. */
   public TrashBot(City c)
   {  super(c, 0, 0, Direction.EAST);
   }

   /** Collect the trash along a fenced yard. */
   public void collectTrash()
   {
   }
}
```

Case Study: TrashBot (2/6)

```
import becker.robots.*;

public class TrashBot extends RobotSE
{
   public TrashBot(City c)…            // done

   /** Collect the trash along a fenced yard. */
   public void collectTrash()
   {  this.collectOneSide();
      this.collectOneSide();
      this.collectOneSide();
      this.collectOneSide();
   }

   /** Collect the trash along one side of the fence, stopping at the corner. */
   private void collectOneSide()
   {
   }
}
```

Case Study: TrashBot (3/x)

```
import becker.robots.*;

public class TrashBot extends RobotSE
{
   public TrashBot(City c)…            // done
   public void collectTrash()…         // done

   /** Collect the trash along one side of the fence, stopping at the corner. */
   private void collectOneSide()
   {  while (this robot is not blocked by the fence on the next side)
      {  pick up the trash on this intersection (if any)
         move to the next intersection
      }
      this.turnRight();
   }
}
```

Case Study: TrashBot (4/6)

```
import becker.robots.*;

public class TrashBot extends RobotSE
{
   public TrashBot(City c)…            // done
   public void collectTrash()…         // done

   /** Collect the trash along one side of the fence, stopping at the corner. */
   private void collectOneSide()
   {  while (this.frontIsClear())
      {  this.pickupTrash();
         this.move();
      }
      this.turnRight();
   }

   /** Pick the trash at this location. */
   private void pickTrash()
   {
   }
}
```

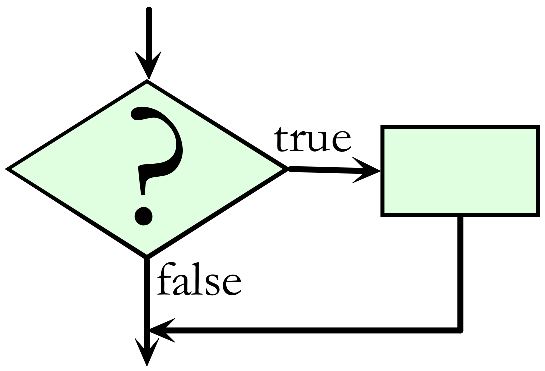
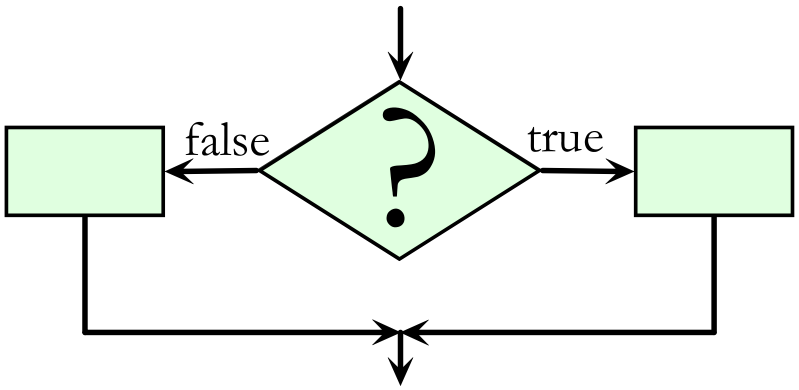
Case Study: TrashBot (5/6)

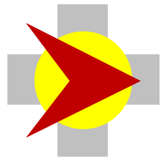
```
import becker.robots.*;

public class TrashBot extends RobotSE
{
   public TrashBot(City c)…             // done
   public void collectTrash()…          // done
   private void collectOneSide()…  // done

   /** Pick the trash at this location. */
   private void pickTrash()
   {  if (this robot is beside some trash)
      {  pick it up
      }
   }
}
```

Case Study: TrashBot (6/6)

```
import becker.robots.*;

public class TrashBot extends RobotSE
{
   public TrashBot(City c)…            // done
   public void collectTrash()…         // done

   /** Collect the trash along one side of the fence, stopping at the corner. */
   private void collectOneSide()
   {  while (this.frontIsClear())
      {  this.pickupTrash();
         this.move();
      }
      this.turnRight();
   }

   /** Pick the trash at this location. */
   private void pickTrash()
   {  if (this.canPickThing())
      {  this.pickThing();
      }
   }
}
```

4.4: Using the if-else Statement

	if statement	**if-else** statement
Question:	Should this group of statements be executed *once or not at all*?	Which group of statements should be executed once – this group or that group?
Flow-chart:		

Example:

 	
	`if (karel.canPickThing())`
	`{  karel.turnLeft();`
	`}`
	`karel.move();`
		`if (jess.canPickThing())`
		`{  jess.turnLeft();`
		`} else`
		`{  jess.turnRight();`
		`}`
		`jess.move();`

4.5: Writing Predicates

Well-named tests and avoiding *not* (!) in **if** and **while** statements make our code easier to read and understand. For example:

Harder:

```
if (!this.frontIsClear())
{  this.turnLeft();
}
```

Easier:

```
if (this.frontIsBlocked())
{  this.turnLeft();
}
```

We can add our own predicates to a subclass of **Robot**, just like we can add our own commands. Use the following template:

```
«accessModifier» boolean «predicateName»(«optionalParameters»)
{  return «booleanExpression»
}
```

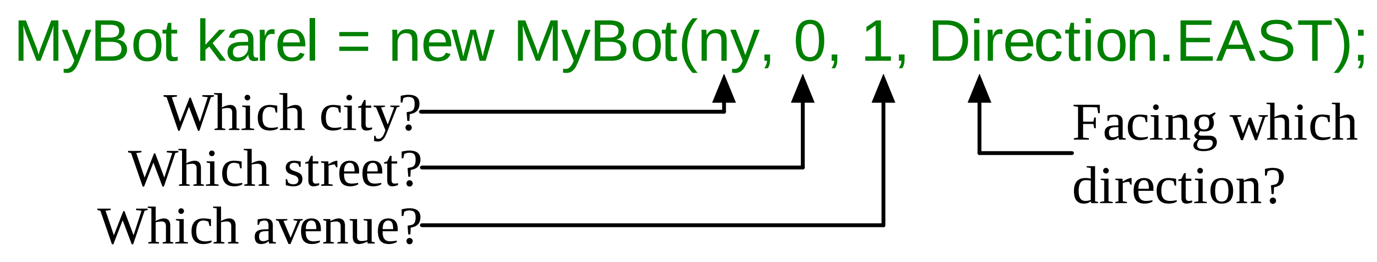
Exercises:

- Is this robot’s front blocked?
- Is this robot on fifth avenue?
- Is this robot facing south?

4.6: Parameters

Purpose: To provide additional information to a constructor or method.

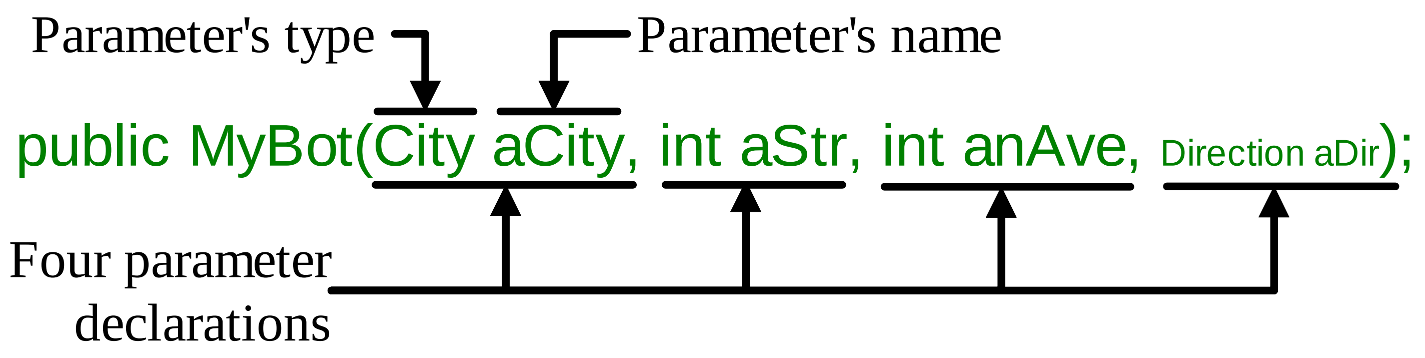
Usage Example:



```
MyBot karel = new MyBot(ny, 0, 1, Direction.EAST);
```

These values are called *arguments*. They are also called *actual parameters*.

Declaration Example:



```
public MyBot(City aCity, int aStr, int anAve, Direction aDir);
```

These pairs are called *formal parameters*, or simply *parameters*. Consecutive pairs are separated by commas.

Each parameter has the value of the corresponding argument.

Without parameters:

Usage Example:

```
if (this.isOnFifthAvenue())
{  this.turnAround();
}
```

Declaration:

```
public boolean isOnFifthAvenue()
{  return this.getAvenue() == 5;
}
```

With parameters:

Usage Example:

```
if (this.isOnAvenue(5))
{  this.turnAround();
}
```

Declaration:

```
public boolean isOnAvenue(int anAve)
{  return this.getAvenue() == anAve;
}
```

Use a parameter to generalize the following predicate:

```
public boolean isFacingSouth()
{  return this.getDirection() == Direction.SOUTH;
}
```

Write a predicate that will determine if a robot has at least *x* **Thing**s in its backpack, where *x* is specified as a parameter.

Usage examples:

```
if (this.hasAtLeast(5))  …
if (this.hasAtLeast(500)) …
```

4.6.1: Using a while Statement & a Parameter

Parameters may also be used with commands. Both of these examples assume **karel**'s current avenue is less than 50.

Example without parameters:

Usage Example: **karel.moveToAvenue50();**

Declaration:

```
public void moveToAvenue50()
{  while (this.getAvenue() < 50)
   {  this.move();
   }
}
```

Example with parameters:

Usage Example: **karel.moveToAvenue(50);**

Declaration:

```
public void moveToAvenue(int ave)
{  while (this.getAvenue() < ave)
   {  this.move();
   }
}
```

Write methods to implement the following commands:

karel.carryAtLeast(50) instructs the robot **karel** to pick up things from the current intersection so that it is carrying at least 50 things.

karel.face(Direction.EAST) causes the robot **karel** to turn left until it is facing East.

karel.moveToAvenue(5) causes the robot **karel** to move to avenue 5. Unlike the previous example, **karel**'s current avenue is unspecified. It may be less than, greater than, or even equal to the given argument. Simplify your solution by using **face**. Apply stepwise refinement.

4.6.2: Using Assignment with a Loop

Consider the following ill-advised method:

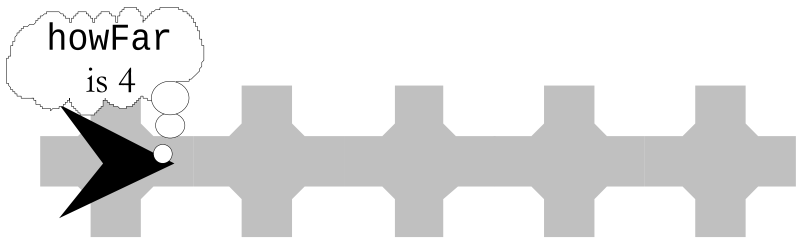
```
public void step(int howFar)
{  while (howFar > 0)
   {  this.move();
   }
}
```

If we could implement the pseudocode addition, what would the method do?

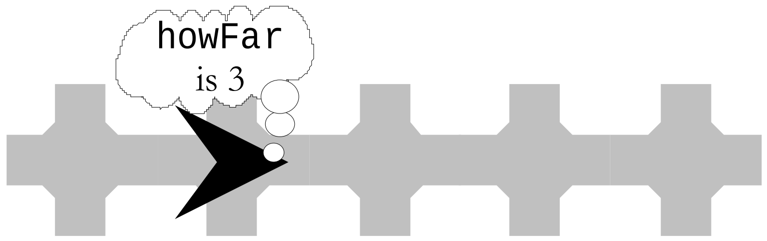
```
public void step(int howFar)
{  while (howFar > 0)
   {  this.move();
      make howFar one less than it is now
   }
}
```

4.6.2: Tracing step(int howFar)

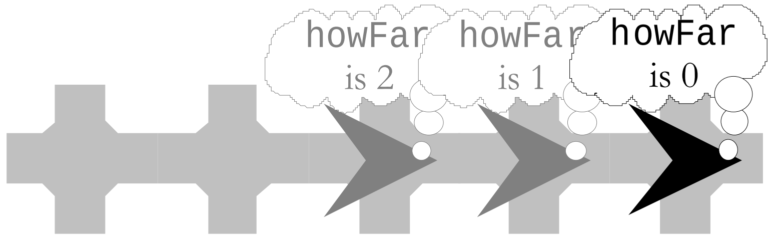
```
while (howFar > 0)
{ this.move();
```
make howFar one less than it is now
```
}
```



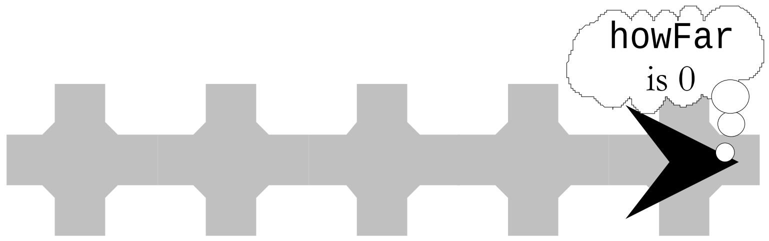
```
while (howFar > 0)
{ this.move();
```
make howFar one less than it is now
```
}
```



```
while (howFar > 0)
{ this.move();
```
make howFar one less than it is now
```
}
```



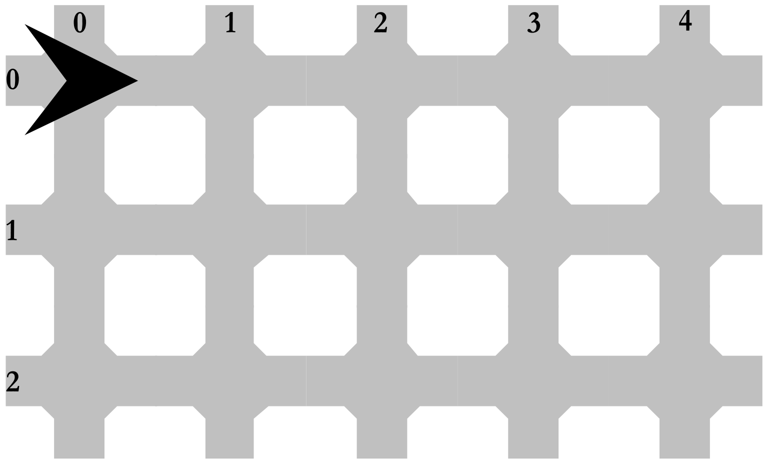
```
while (howFar > 0)
{ this.move();
```
make howFar one less than it is now
```
}
```

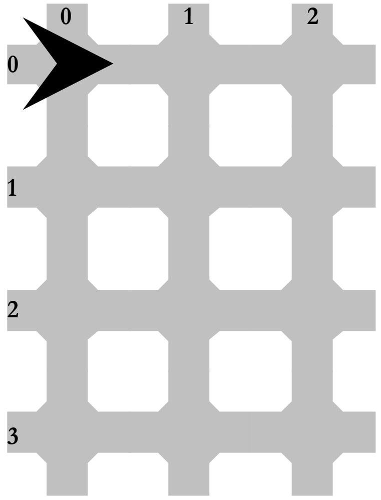


Case Study: RectanglePlanter (1/5)

Develop a new kind of robot that can plant flowers in a rectangle, the size of which is specified using parameters. Two examples are:

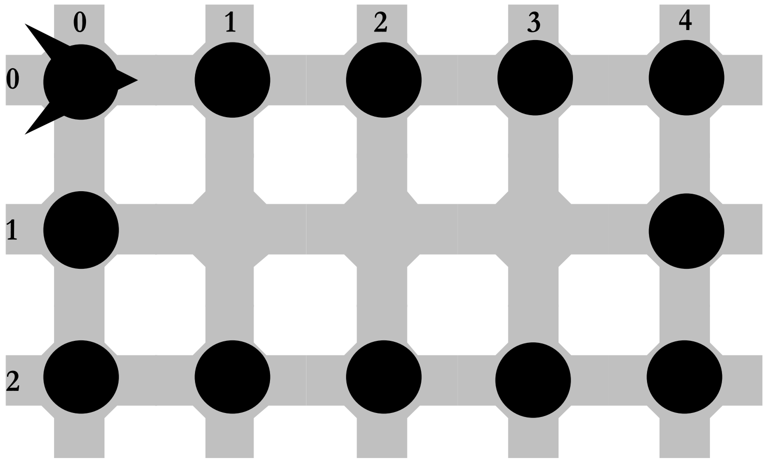
Initial Situation

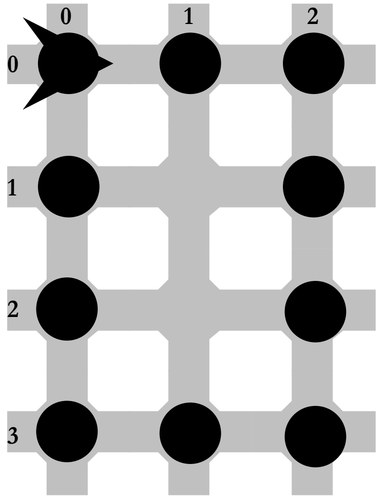




Command **karel.plantRect(5, 3);** **karel.plantRect(3, 4);**

Final Situation



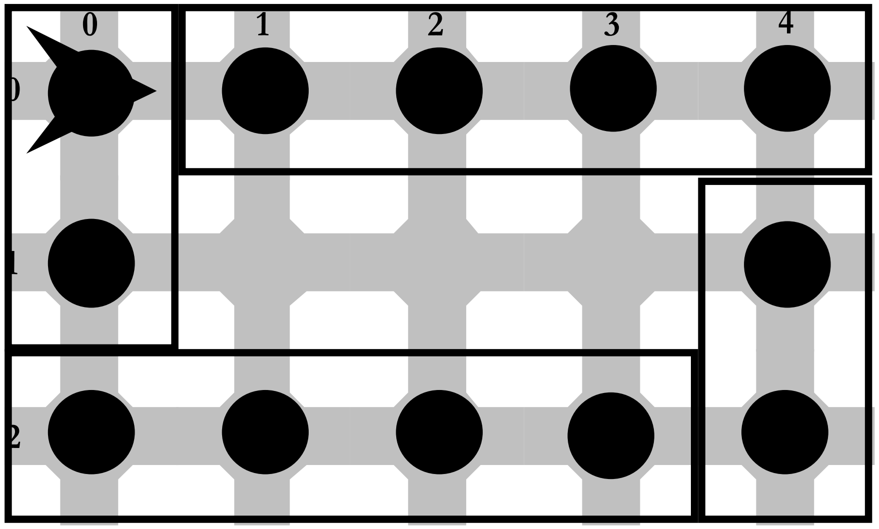


Case Study: RectanglePlanter (2/5)

```
import becker.robots.*;

/** A class of robots that plants Things in the form of a hollow rectangle.
*
* @author Byron Weber Becker */
public class RectanglePlanter extends RobotSE
{
    /** Create a new rectangle planter.
     * @param aCity The robot's city.
     * @param aStreet The robot's initial street.
     * @param anAvenue The robot's initial avenue.
     * @param aDir The robot's initial direction.
     * @param numThings The number of things initially in the robot's backpack. */
    public RectanglePlanter(City aCity, int aStreet, int anAvenue,
                            Direction aDir, int numThings)
    {  super(aCity, aStreet, anAvenue, aDir, numThings);
    }

    /** Plant a hollow rectangle of Things.  The robot must be positioned in the
    * rectangle's upper-left corner facing east.
    * @param width The number of avenues wide.
    * @param height The number of streets high. */
    public void plantRect(int width, int height)
    {
    }
}
```

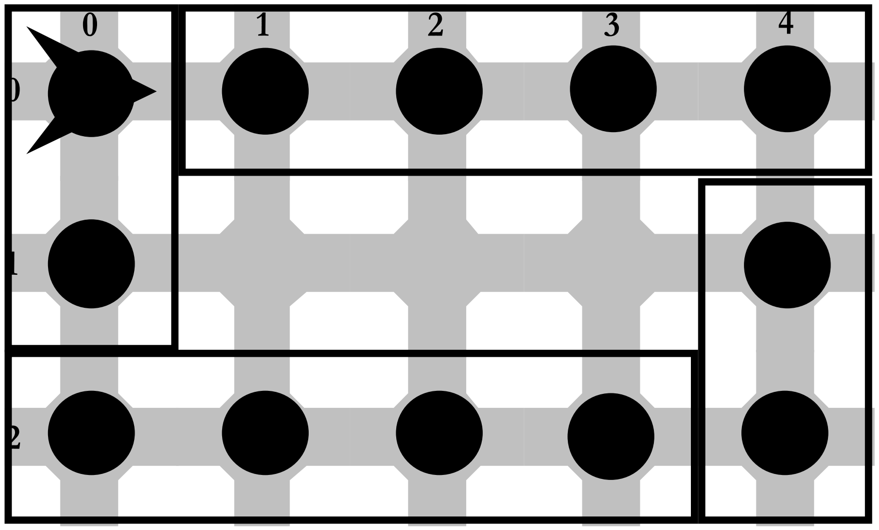
Case Study: RectanglePlanter (3/5)

```
import becker.robots.*;

public class RectanglePlanter extends RobotSE
{
   public RectanglePlanter(…)…    // done

   public void plantRect(int width, int height)
   {  this.plantSide(width);
      this.plantSide(height);
      this.plantSide(width);
      this.plantSide(height);
   }

   /** Plant one side of the rectangle;  make the side ‘length’ intersections long, but plant
   *   one less Thing to avoid duplicationg at
   *   the corners. */
   protected void plantSide(int length)
   {
   }
}
```



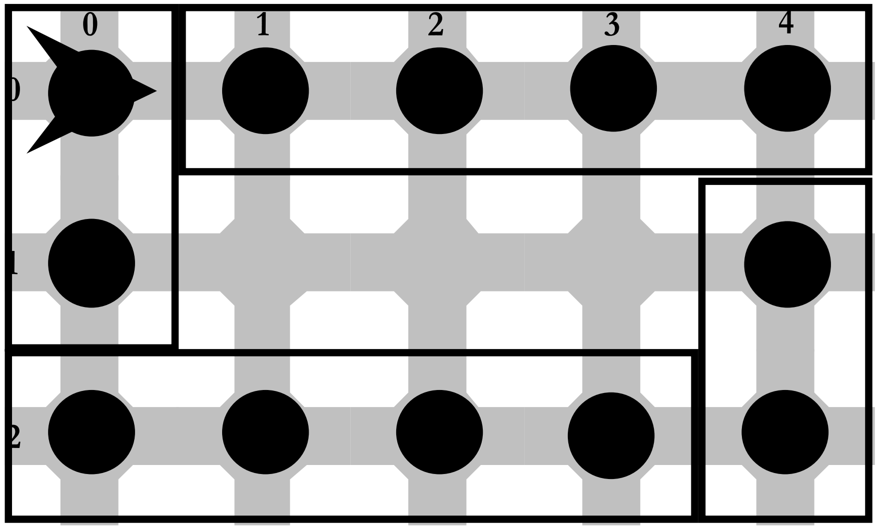
Case Study: RectanglePlanter (4/5)

```
import becker.robots.*;

public class RectanglePlanter extends RobotSE
{
   public RectanglePlanter(…)…                         // done
   public void plantRect(int width, int height)        // done

   /** Plant one side of the rectangle;  make the side ‘length’ intersections long, but plant
   *   one less Thing to avoid duplicationg at
   *   the corners. */
   protected void plantSide(int length)
   {  length = length – 1;
      this.plantLine(length);
      this.turnRight();
   }

   /** Plant one side of the rectangle with Things,
   *   beginning with the next intersection. */
   protected void plantLine(int len)
   {
   }
}
```

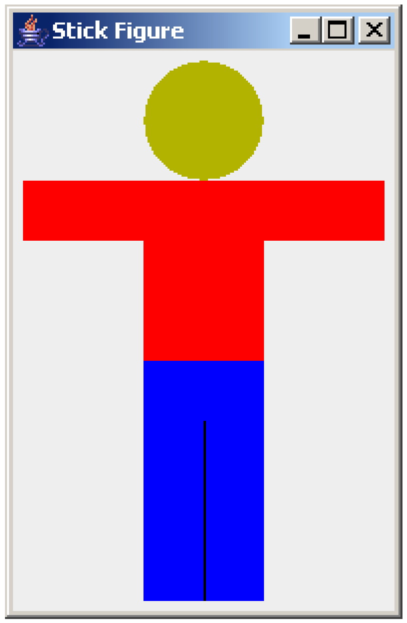


Case Study: RectanglePlanter (5/5)

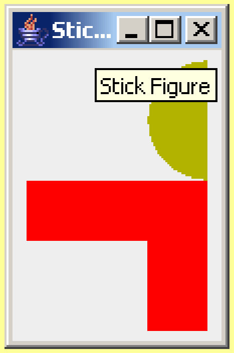
```
public class RectanglePlanter extends RobotSE
{
   public RectanglePlanter(…)…                              // done
   public void plantRect(int width, int height)      // done
   protected void plantSide(int length)               // done
   {  length = length – 1;
      this.plantLine(length);
      this.turnRight();
   }

   /** Plant one side of the rectangle with Things,
   *   beginning with the next intersection. */
   protected void plantLine(int len)
   {  while (len > 0)
      {  this.move();
         this.plantIntersection();
         len = len – 1;
      }
   }
   protected void plantIntersection()
   {  this.putThing();
   }
}
```



```
public class StickFigure extends JComponent
{  public StickFigure ()
   {  super ();
      Dimension prefSize = new Dimension(180, 270);
      this.setPreferredSize(prefSize);
   }

   // Paint a stick figure.
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);

      g.setColor(Color.YELLOW);     // head
      g.fillOval(60, 0, 60, 60);

      g.setColor(Color.RED);        // shirt
      g.fillRect(0, 60, 180, 30);
      g.fillRect(60, 60, 60, 90);

      g.setColor(Color.BLUE);       // pants
      g.fillRect(60, 150, 60, 120);
      g.setColor(Color.BLACK);
      g.drawLine(90, 180, 90, 270);
   }
}
```

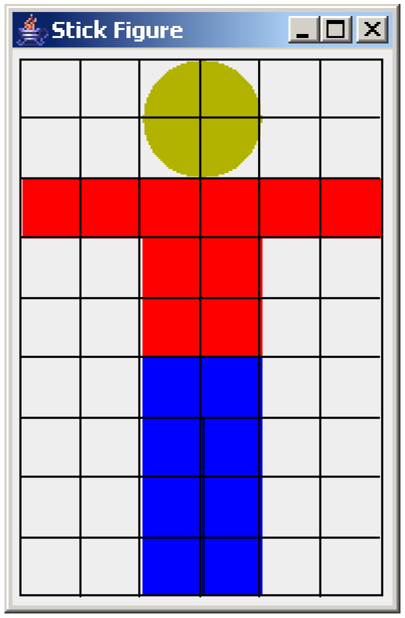


Changing the preferred size of the component to (90, 135) results in:



Approach:

1. Use a grid to identify significant points in the drawing.



Think of this grid as a new coordinate system. The head is enclosed in a square with (2,0) as the upper left corner and (4,2) as the lower right corner.

These coordinates are independent of the actual size of the component.

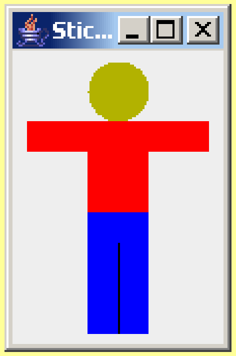
2. Use **this.getWidth()** and **this.getHeight()** to obtain the actual size of the component (in pixels).

3. Combine 1 and 2 to specify where to draw:

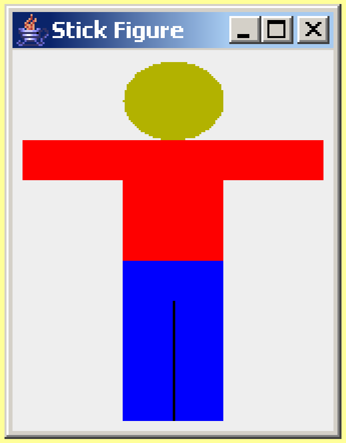
```
g.fillOval(this.getWidth()*2/6,  // x coord of head is 2/6 of comp's width
           this.getHeight()*0/9, // y coord of head is 0/9 of comp's height
           this.getWidth()*2/6,  // head is 2/6 of component's width
           this.getHeight()*2/9);// head is 2/9 of component's height
```

4.7.2: Scaling an Image

```
public class StickFigure extends JComponent
{  // Constructor omitted
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);

      // standard stuff to scale the image
      Graphics2D g2 = (Graphics2D)g;
      g2.scale(this.getWidth()/6, this.getHeight()/9);
      g2.setStroke(new BasicStroke(1.0F/this.getWidth()));

      g2.setColor(Color.YELLOW.);        // head
      g2.fillOval(2, 0, 2, 2);

      g2.setColor(Color.RED);            // shirt
      g2.fillRect(0, 2, 6, 1);
      g2.fillRect(2, 2, 2, 3);

      g2.setColor(Color.BLUE);           // pants
      g2.fillRect(2, 5, 2, 4);
      g2.setColor(Color.BLACK);
      g2.drawLine(3, 6, 3, 9);
   }
}
```



90 x 135



150 x 180

4.8.1: The Once or Not at All Pattern

Name: Once or Not at All

Context: A group of statements must be executed once or not at all, based on the value of a Boolean expression.

Solution: Use an **if**-statement, for example:

```
if (this.canPickThing())
{  this.pickThing();
}
```

```
if (this.numStudentsInCourse() < 100)
{  this.addStudentToCourse();
}
```

In general,

```
if («test»)
{ «list of statements»
}
```

Consequences: Programs are able to respond differently, depending on the situation.

Related Patterns: The Either This or That pattern executes one of two groups of statements. The Zero or More Times pattern is used if statements must be executed repeatedly rather than once or not at all.

4.8.2: The Zero or More Times Pattern

Name: Zero or More Times

Context: A group of statements must be executed an unknown number of times: as few as zero, possibly many times. Whether to execute the statements again can be determined with a Boolean expression.

Solution: Use a **while**-statement, for example:

```
while (this.frontIsClear())      while (this.countThingsInBackpack() < 4)
{  this.turnLeft();              {  this.pickThing();
}                                }
```

In general,

```
while («test»)
{ «list of statements»
}
```

Consequences: The *«list of statements»* may be executed as few as zero times or many more, depending on the outcome of the *«test»*.

Related Patterns: The Once or Not At All and Either This or That patterns execute a group of statements once or not at all.

4.8.3: The Either This or That Pattern

Name: Either This or That

Context: One of two groups of statements, either this group or that group, must be executed. Which group to execute depends on the outcome of a Boolean expression.

Solution: Use an **if-else** statement, for example:

```
if (this.frontIsClear())
{  this.move();
} else
{  this.turnLeft();
}
```

In general,

```
if («test»)
{ «statementGroup1»
} else
{ «statementGroup2»
}
```

Consequences: Exactly one group of statements is executed once.

Related Patterns: Once or Not at All is a specialization of this pattern; Zero or More Times can execute statements repeatedly.

4.8.4: The Simple Predicate Pattern

Name: Simple Predicate

Context: A Boolean expression is not easy to read or understand.

Solution: Put the Boolean expression in a method with a meaningful name. Return the value of the Boolean expression. Such a method is called a predicate. For example,

```
public boolean frontIsBlocked()
{  return !this.frontIsClear();
}
```

In general,

```
«accessModifier» boolean «predicateName»(«optParameters»)
{  return «a Boolean expression»
}
```

Consequences: Statements that use the predicate are easier to understand. The predicate can be easily reused.

Related Patterns: This pattern is often used to define predicates for use in Once or Not at All and Zero or More Times patterns. This pattern is a specialization of the more general Predicate pattern discussed in lesson 05.

The Parameterized Method Pattern

Name: Parameterized Method

Context: A method might do many variations of its task if it only had some information from its client to say which variation to perform.

Solution: Use one or more parameters to communicate information from the client to the method. In general,

```
  «accessModifier» void «methodName»(«paramType1» «paramName1»,
                                     «paramType2» «paramName2»,
                                     ...
                                     «paramTypeN» «paramNameN»)
  { «list of statements, at least some using paramNameX»
  }
```

Consequences: A parameterized method is more flexible than similar unparameterized methods.

Related Patterns: The Parameterless Command and Helper Method patterns are simplifications of this pattern.

4.8.5: The Count-Down Loop Pattern

Name: Count-Down Loop

Context: You must execute an action a specified number of times. The number is often given via a parameter.

Solution: Write a **while** statement that uses a variable, often a parameter, to count down to zero.

```
while («variable» > 0)
{ «list of statements»
  «variable» = «variable» - 1;
}
```

```
public void plantLine(int length)
{ while (length > 0)
  { this.move();
    this.putThing();
    length = length – 1;
  }
}
```

Consequences: This pattern performs an action a specified number of times, even if the number is large.

Related Patterns: This pattern is a special case of the Zero or More Times pattern.

Name: Scale an Image

Context: An image drawn by the **paintComponent** method in a subclass of **JComponent** needs to scale to different sizes.

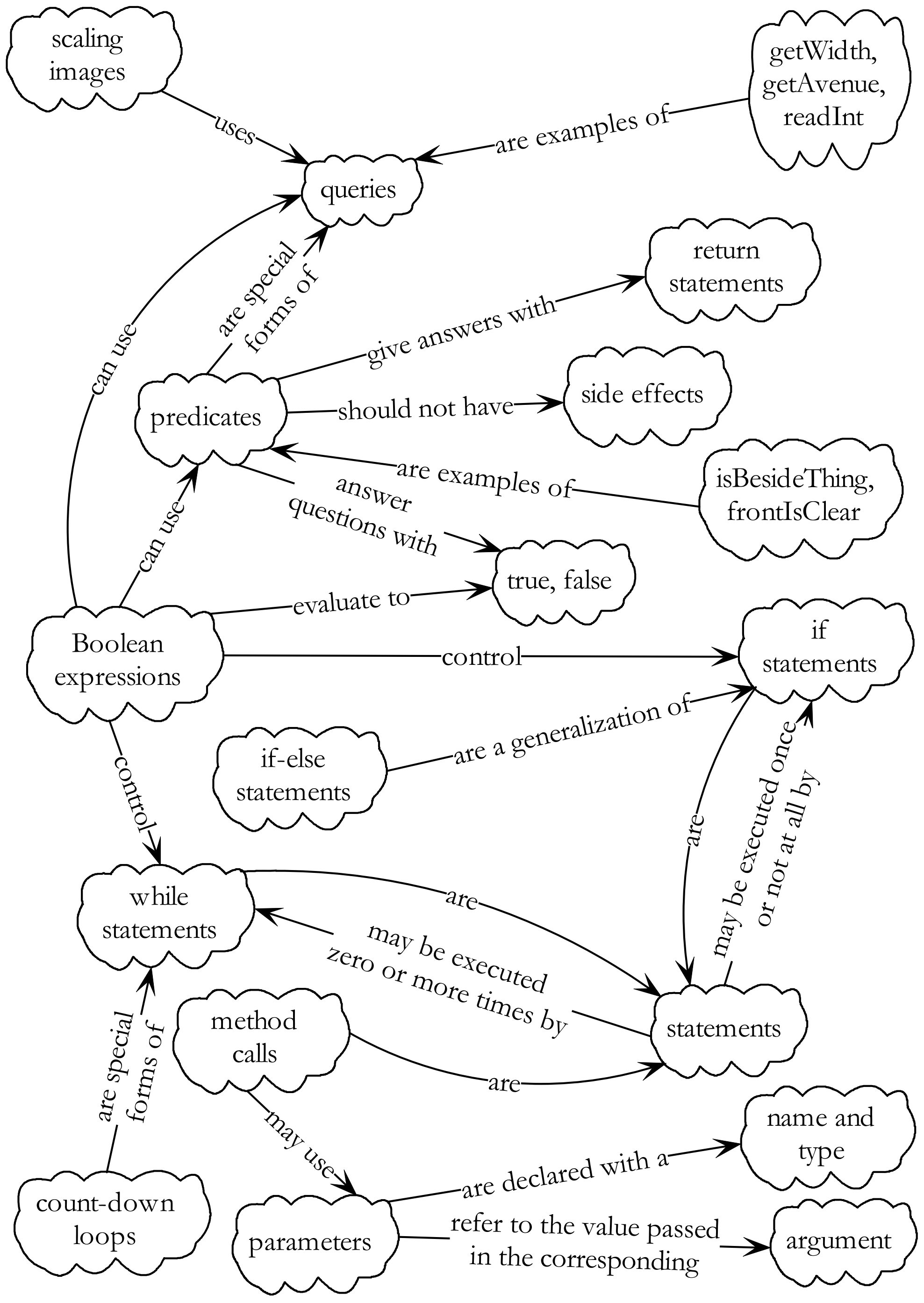
Solution: Draw the image based on a predefined grid for the coordinates. Then use the following code template to use that coordinate grid while drawing.

```
public void paintComponent(Graphics g)
{  super.paintComponent(g);
   // Standard code to scale the image
   Graphics2D g2 = (Graphics2D)g;
   g2.scale(this.getWidth()/«gridWidth»,
                 this.getHeight()/«gridHeight»);
   g2.setStroke(new BasicStroke(1.0F/this.getWidth()));

   «statements using g2 to draw the image»
}
```

Consequences: Parameters used by methods such as **fillRect** are multiplied by appropriate constants to scale the image.

Related Patterns: This is a specialization of Draw a Picture pattern.

4.9: Concept Map



Summary

We have learned:

- how to control which statements are executed, using the following techniques:
  - the **if**-statement, implementing the Once or Not At All pattern.
  - the **while**-statement, implementing the Zero or More Times pattern.
  - the **if-else** statement, implementing the Either This or That pattern.
- that **if**, **while**, and **if-else** statements are controlled using Boolean expressions.
- that predicates such as **frontIsClear** are Boolean expressions and that we can write our own predicates.
- how to use parameters to make methods more flexible.
- how to use parameters in **while** statements.
- how to scale an image.