Java:

Learning to Program with Robots

Chapter 05: More Decision Making

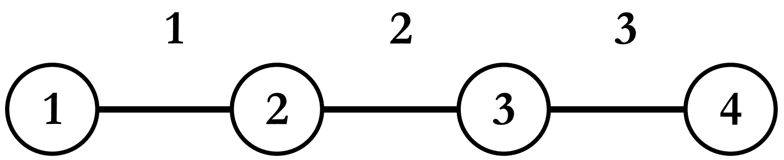
Chapter Objectives

After studying this chapter, you should be able to:

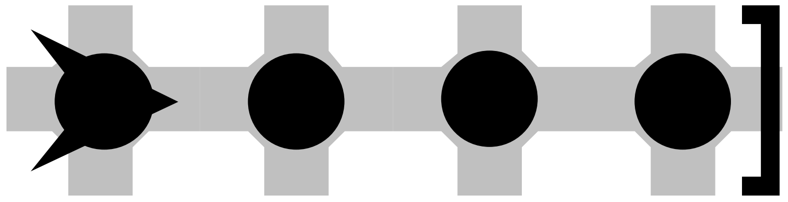
- Follow a process for constructing **while** loops with fewer errors.
- Avoid common errors encountered with **while** loops.
- Use temporary variables to remember information within methods.
- Nest statements inside other statements.
- Manipulate Boolean expressions.
- Perform an action a predetermined number of times using a **for** statement.
- Write **if** and **while** statements with appropriate style.

The Fence-Post Problem

A fence with three sections requires four fence posts.

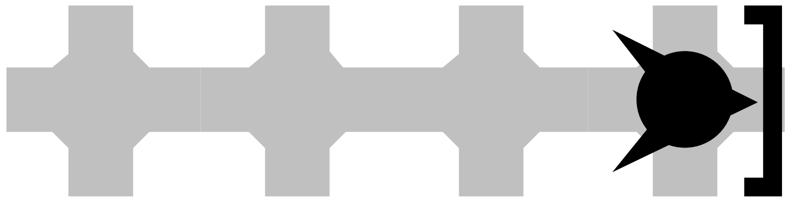


Picking up all the things between the robot's current location and the wall requires three **moves** (fence sections) but four **pickThings** (fence posts).



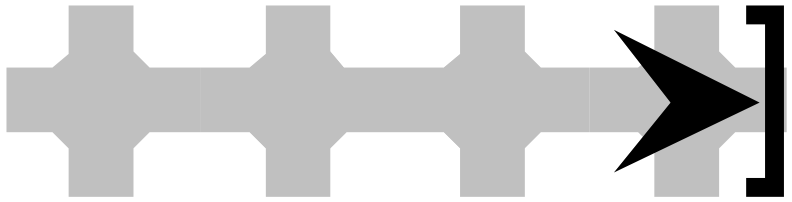
A typical "solution" has the same number of fence sections and posts:

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



We need one more post:

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



Infinite Loops

When do these loops stop?

```
while (this.isFacingNorth())
{  this.pickThing();
   this.move();
}
```

```
while (this.isBesideThing())
{  this.turnLeft();
}
```

There must be a relationship between what occurs in the body of the loop and what occurs in the test. What?

5.1.2: A Four-Step Process for while Loops

Step 1: Identify the actions that must be repeated to solve the problem.

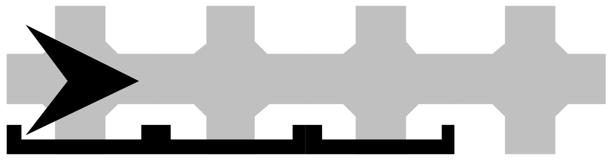
Step 2: Identify that Boolean expression that must be true when the **while** statement has completed executing. Negate it.

Step 3: Assemble the **while** loop with the actions from Step 1 as the body and the Boolean expression from Step 2 as the test.

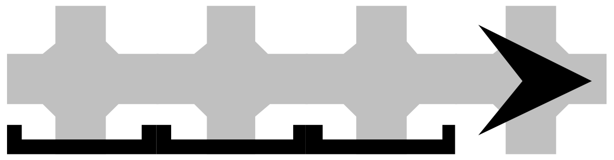
Step 4: Add additional actions before or after the loop to complete the solution.

Case Study: Move Along a Wall (Step 1)

Problem: Write a method to move a robot to the end of a wall.



Initial Situation



Final Situation

Step 1: Identify the actions that must be repeated to solve the problem.

One approach is to list the actions that must be taken for a small problem – without using a loop:

```
turn right (to check if a wall is there)
turn left
move
turn right (to check if a wall is there)
turn left
move
turn right (to check if a wall is there)
turn left
move
turn right (to check if a wall is there)
turn left
```

Case Study: Move Along a Wall (Step 2)

Step 2: Identify that Boolean expression that must be true when the **while** statement has completed executing. Negate it.

The loop should stop when the robot has moved past the end of the wall.

Negating this gives us when the loop should continue:

```
while (this robot has not moved past the end of the wall)
{ …
}
```

Refining this further,

```
while (this robot is blocked when facing left)
{ …
}
```

This implies

```
while (this.frontIsBlocked())…
```

Case Study: Move Along a Wall (Steps 3&4)

Step 3: Assemble the **while** loop with the actions from Step 1 as the body and the Boolean expression from Step 2 as the test.

Step 4: Add additional actions before or after the loop to complete the solution.

```
while (this.frontIsB…)
{  this.turnRight();
   this.turnLeft();
   this.move();
}
this.turnRight();
this.turnLeft();
```

```
turn right
turn left
move
turn right
turn left
move
turn right
turn left
move
turn right
turn left
```

```
this.turnRight();
while (this.frontIsB…)
{  this.turnLeft();
   this.move();
   this.turnRight();
}
this.turnLeft();
```

```
turn right
turn left
move
turn right
turn left
move
turn right
turn left
move
turn right
turn left
```

```
this.turnRight();
this.turnLeft();
while (this.frontIsB…)
{  this.move();
   this.turnRight();
   this.turnLeft();
}
```

```
turn right
turn left
move
turn right
turn left
move
turn right
turn left
move
turn right
turn left
```

But, which one is correct?

5.2: Tempoary (Local) Variables

Temporary variables

- are also called “local variables.”
- occur inside a method.
- store a value until the end of the method.
- have a type such as **Robot** or **int** which determines what kind of values may be stored in the variable.
- must be given an initial value.

Two examples:

$$\underbrace{\texttt{int}}_{\text{type}}\ \underbrace{\texttt{counter}}_{\text{name}} = \underbrace{\texttt{0}}_{\text{initial value}};$$

$$\underbrace{\texttt{Robot}}_{\text{type}}\ \underbrace{\texttt{karel}}_{\text{name}} = \underbrace{\texttt{new Robot(sanDiego, 1, 2, Direction.EAST)}}_{\text{initial value}};$$

5.2: Using an Integer Variable

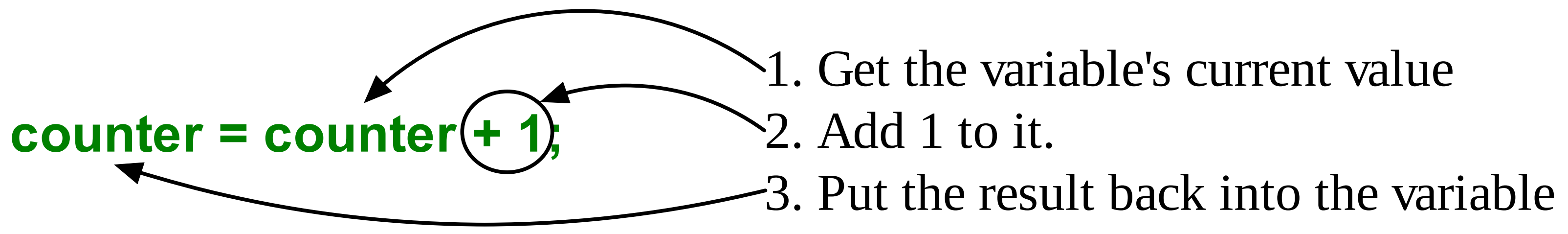
Declaring an integer variable:

```
int counter = 0;
```

Using a variable’s value:

```
if (counter == 0)      while (counter > 0)      this.myMethod(counter);
{  …                   {  …
}                      }
```

Changing a variable’s value:



```
counter = counter + 1;
```

1. Get the variable's current value
2. Add 1 to it.
3. Put the result back into the variable

5.2.1: Counting the Things on an Intersection

Wanted:

Count things on this intersection

```
if (0 things)
{  this.move();
}
if (1 thing)
{  this.turnLeft();
}
if (2 things)
{  this.turnRight();
}
```

Solution:

```
int numThingsHere = 0;

while (this.canPickThing())
{  this.pickThing();
   numThings = numThings + 1;
}

if (numThingsHere == 0)
{  this.move();
}
if (numThingsHere == 1)
{  this.turnLeft();
}
if (numThingsHere == 2)
{  this.turnRight();
}
```

5.2.2: Tracing with a Temporary Variable (1/2)

Suppose the robot is on intersection (3,5), which has 2 **Things**.

Code	test	(str,ave)	Dir	numThingsHere	# on intersection
		(3,5)	N	???	2
int numThings = 0;					
		(3,5)	N	0	2
while (this.canPickThing())					
	true	(3,5)	N	0	2
{ this.pickThing();					
		(3,5)	N	0	1
numThingsHere = numThingsHere + 1;					
		(3,5)	N	1	1
while (this.canPickThing())					
	true	(3,5)	N	1	1

5.2.2: Tracing with a Temporary Variable (2/2)

	true	(3,5)	N	1	1
{ this.pickThing();					
		(3,5)	N	1	0
numThingsHere = numThingsHere + 1;					
		(3,5)	N	2	0
while (this.canPickThing())					
	false	(3,5)	N	2	0
if (numThingsHere == 0)					
	false	(3,5)	N	2	0
if (numThingsHere == 1)					
	false	(3,5)	N	2	0
if (numThingsHere == 2)					
	true	(3,5)	N	2	0
{ this.turnRight();					
		(3,5)	E	2	0

We could do something equivalent with the number of things in the robot's backpack:

One way…

```
if (this.countThingsInBackpack() == 0)
{  this.move();
}
if (this.countThingsInBackpack() == 1)
{  this.turnLeft();
}
if (this.countThingsInBackpack() == 2)
{  this.turnRight();
}
```

Using a temporary variable…

```
int numThings = this.countThingsInBackpack();
if (numThings == 0)
{  this.move();
}
if (numThings == 1)
{  this.turnLeft();
}
if (numThings == 2)
{  this.turnRight();
}
```

5.2.4: Writing a Query

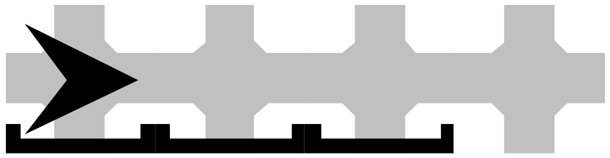
```
/** Count and return the number of things on this robot's current intersection.  Replace the
*   things after counting them.
*   @return the number of things on this robot's current intersection  */
pubic int countThingsHere()
{
   int numThingsHere = 0;

   while (this.canPickThing())
   {  this.pickThing();
      numThingsHere = numThingsHere + 1;
   }

   this.putThing(numThingsHere);

   return numThingsHere;
}
```

Using the query:

```
int numThings = this.countThingsHere();
if (numThings == 0)
{  this.move();
}
if (numThings == 1) …
```

5.2.5: Using a Boolean Temporary Variable

In the case study, it would have been useful to have a predicate, **rightIsBlocked()**, as we followed the wall.



This would have simplified the loop to

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

How?

```
turn to the right
remember if the front is blocked
turn back to the left
return the answer
```

```
public boolean rightIsBlocked()
{  this.turnRight();
   boolean blocked = this.frontIsBlocked();
   this.turnLeft();
   return blocked;
}
```

5.2.6: Scope

Scope is the region of the program in which a variable may be used.

Examples:

```
public void method()
{  int tempVar = 0;
   «statements»
}
```

```
public void method()
{  «statements»
   int tempVar = 0;
   «statements»
}
```

```
public void method()
{  if («booleanExpression»)
   {  «statements»
      int tempVar = 0;
      «statements»
   }
   «statements»
}
```

```
public void method()
{  «statements»
   int tempVar = 0;
   while («booleanExpression»)
   {  «statements»
   }
   «statements»
}
```

Rule: Use a variable from where it is declared to the end of the smallest enclosing block (set of braces).

The general forms of **while** and **if** statements are

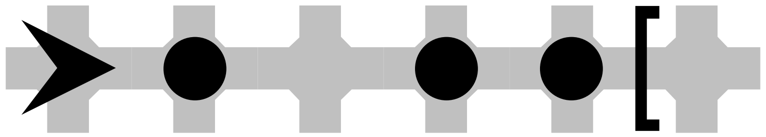
```
while («test»)
{ «statements»
}
```

```
if («test»)
{ «statements»
}
```

```
if («test»)
{ «statements1»
} else
{ «statements2»
}
```

In each of these, ***«statements»*** can include **while**, **if**, and **if-else** *statements*. Examples:

```
/** Pick up one thing (if there is a thing)
*   from each intersection between this
*   robot and the nearest wall it is facing. */
public void pickThingsToWall()
{  while (this.frontIsClear())
   {  this.move();
      if (this.canPickThing())
      {  this.pickThing();
      }
   }
}
```



```
/** Decide which way to turn based on whether
*   a Thing is present and the number of Things
*   being carried. */
public void turnRule()
{  if (this.canPickThing())
   {  // There is a Thing here, so we'll turn.
      if (this.countThingsInBackpack() > 0)
      {  this.turnRight();
      } else
      {  this.turnLeft();
      }
   }
}
```

Problem: A robot is following a trail. It should perform the *first* action in the following table that matches its current situation.

Front is blocked	Turn around
Can pick a **Thing**	Turn right
Left is blocked	Turn left
Anything else	Move

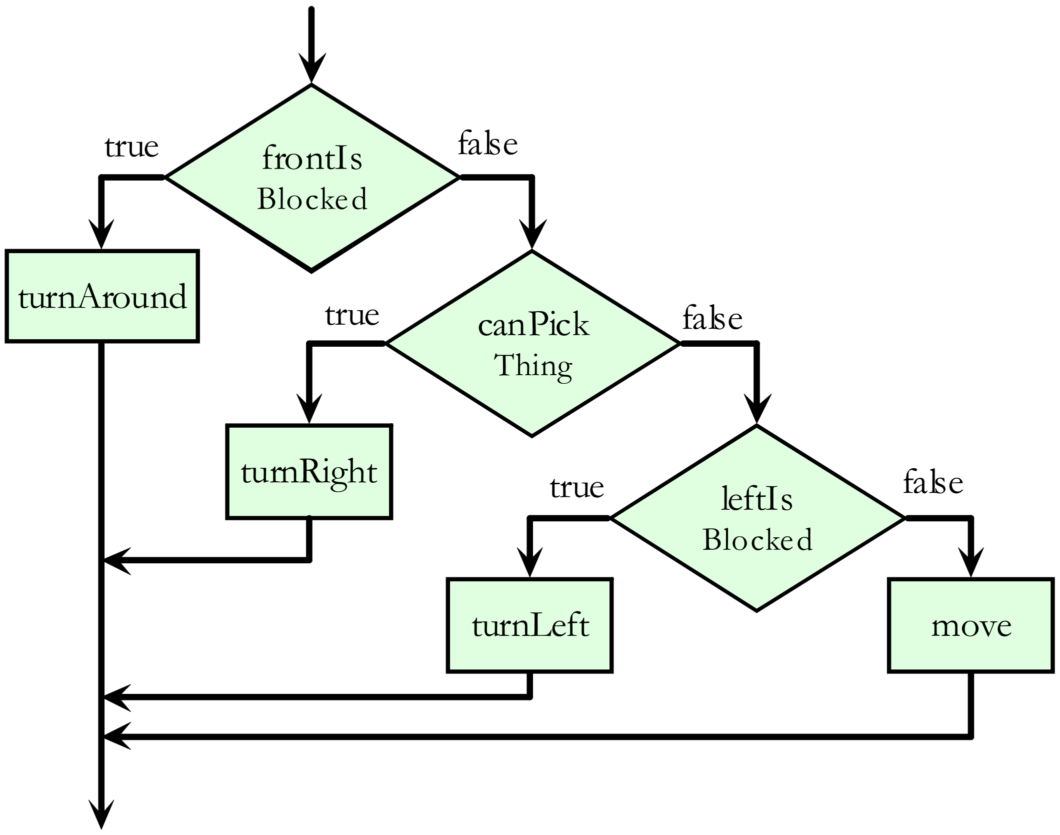
In what situations does the code fragment on the right perform the wrong action?

```
if (this.frontIsBlocked())
{  this.turnAround();
}
if (this.canPickThing())
{  this.turnRight();
}
if (this.leftIsBlocked())
{  this.turnLeft();
} else
{  this.move();
}
```

5.3.3: Cascading if Statements

The following code executes exactly one statement, the first statement encountered when the code is read top-to-bottom.

```
if (this.frontIsBlocked())
{  this.turnAround();
} else
{  if (this.canPickThing())
   {  this.turnRight();
   } else
   {  if (this.leftIsBlocked())
      {  this.turnLeft();
      } else
      {  this.move();
      }
   }
}
```



5.3.3: Cascading if Statements

Notice that each **else** clause has a single **if** statement. Java allows us to omit the braces and reformat it as follows to emphasize that only one statement is executed.

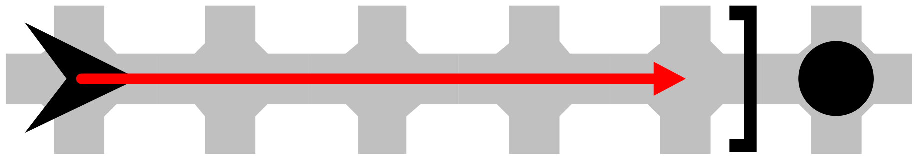
```
if (this.frontIsBlocked())
{  this.turnAround();
} else
{  if (this.canPickThing())
   {  this.turnRight();
   } else
   {  if (this.leftIsBlocked())
      {  this.turnLeft();
      } else
      {  this.move();
      }
   }
}
```

```
if (this.frontIsBlocked())
{  this.turnAround();
} else if (this.canPickThing())
{  this.turnRight();
} else if (this.leftIsBlocked())
{  this.turnLeft();
} else
{  this.move();
}
```

5.4.1: Combinging Boolean Expressions

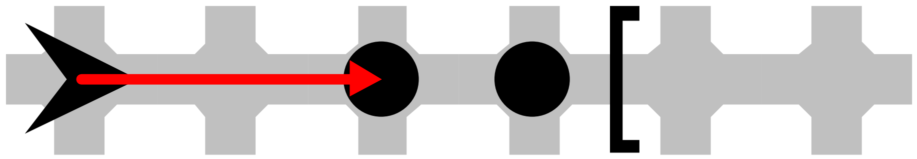
So far, we can move a robot until it is blocked by a **Wall**:

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



We can move a robot until it is on an intersection with a **Thing**:

```
while (!this.canPickThing())
{  this.move();
}
```



But what if we want to move a robot until it is blocked *or* on an intersection with a **Thing**?

```
while (this.frontIsClear() OR !this.canPickThing())
{  this.move();
}

if (this.canPickThing() AND this.frontIsBlocked())
{  this.turnLeft();
}  else
{  this.turnRight();
}
```

5.4.1: The Form of Legal Expressions

The following rules define “legal” expressions:

1. Literal values such as **true**, **false**, and **50** are legal expressions. The type of the expression is the type of the literal.
2. A variable is a legal expression. The type of the expression is the type of the variable.
3. A method call whose arguments are legal expressions with the appropriate types is a legal expression. The type of the expression is the return type of the method.
4. An operator whose operands are legal expressions with the appropriate types is a legal expression. The type of the expression is given by the return type of the operator. Operators include **&&**, **||**, **!**, the comparison operators, and the arithmetic operators.

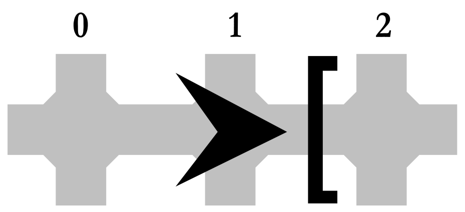
Examples:

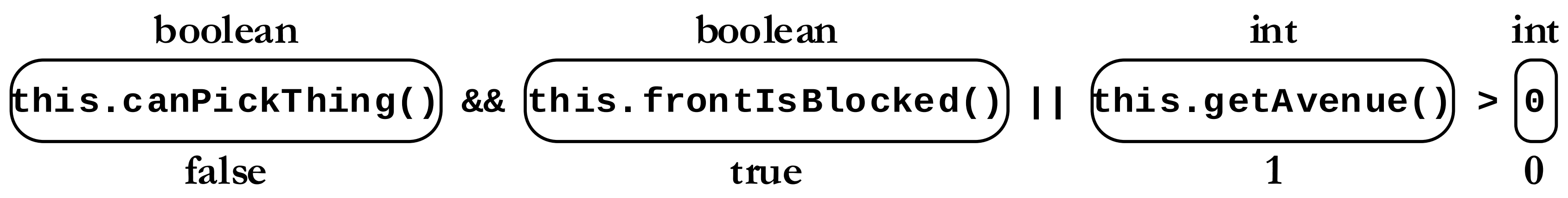
this.getAvenue() Rule 3

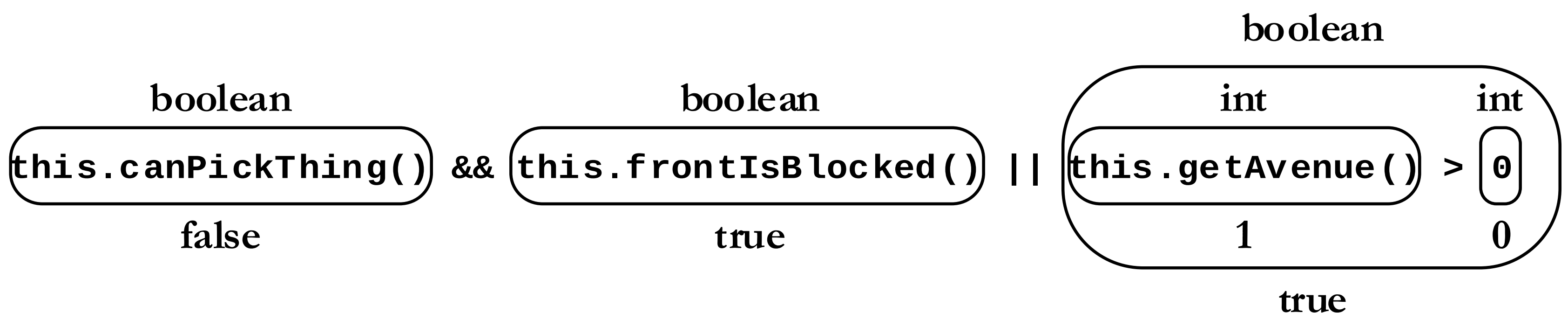
this.getAvenue() > 0 Rules, 3, 1, and 4

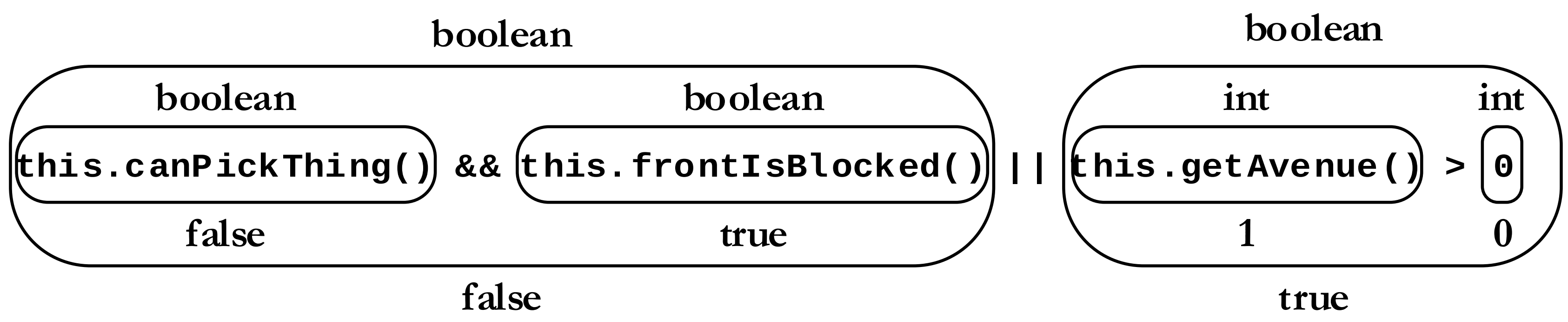
this.canPickThing() && this.frontIsBlocked() || this.getAvenue() > 0

5.4.1: Evaluating Boolean Expressions (1/2)

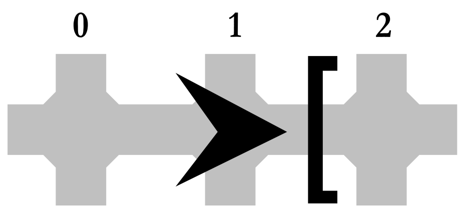


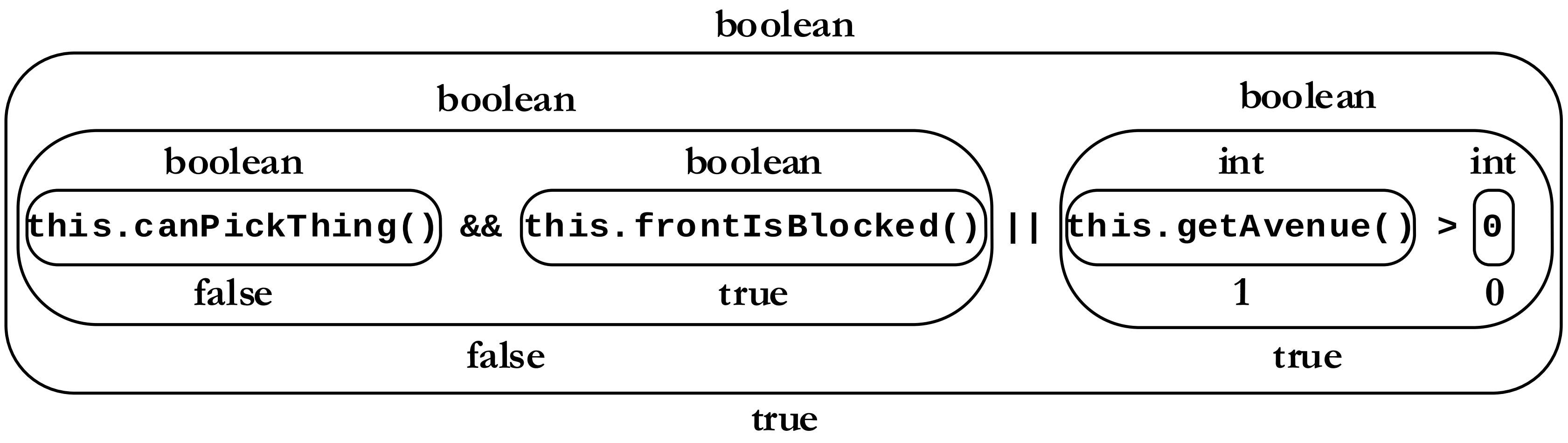




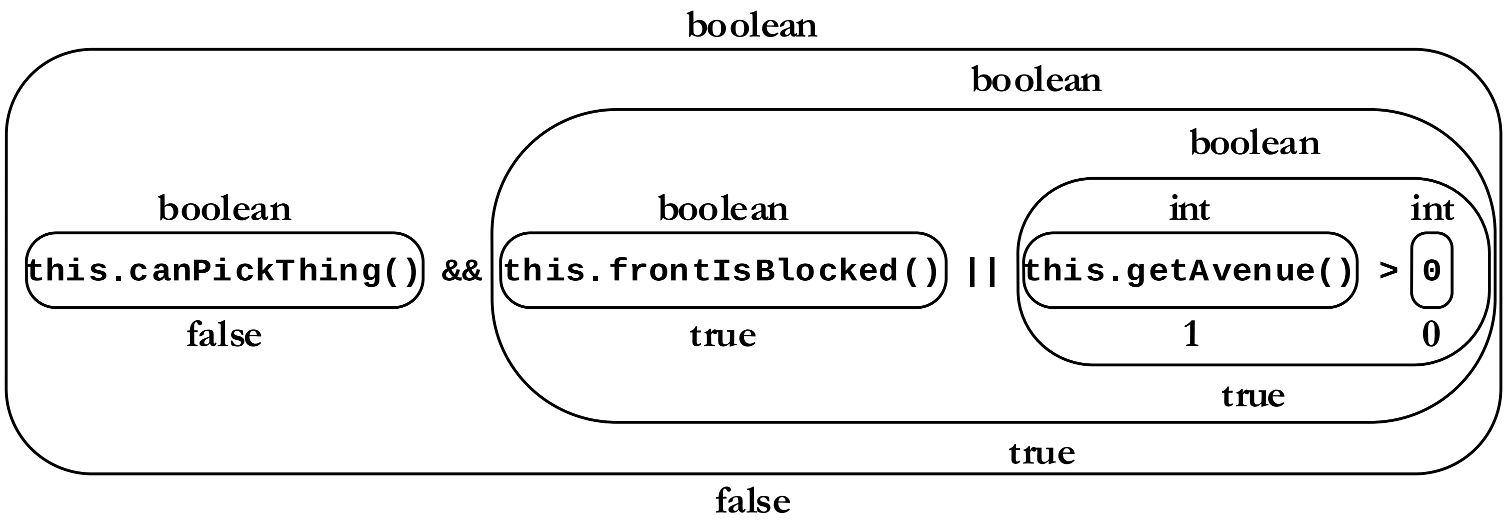


5.4.1: Evaluating Boolean Expressions (2/2)





An incorrect "solution:"



5.4.1: Operator Precedence

Operator	Precedence
method(*parameters*)	15
!	14
* / %	12
+ -	11
< > <= >=	9
== !=	8
&&	4
||	3

Simplifying Negations

Example Expression	Simplification
`!!karel.frontIsClear()`	`karel.frontIsClear()`
`!karel.frontIsBlocked()`	`karel.frontIsClear()`
`!(this.getAvenue() == 0)`	`this.getAvenue() != 0`
`!(this.getAvenue() != 0)`	`this.getAvenue() == 0`

De Morgan's Laws

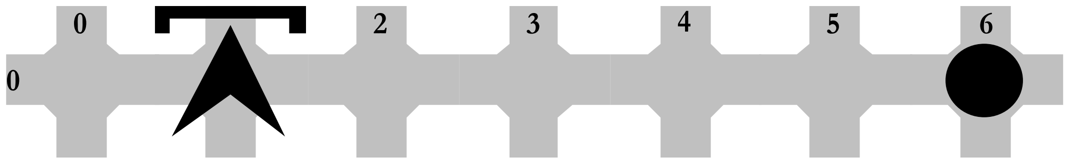
`!(b1 && b2) ≡ !b1 || !b2`	(First Law)
`!(b1 || b2) ≡ !b1 && !b2`	(Second Law)

Example:

```
!(r.canPickThing() || (r.leftIsBlocked() && r.rightIsBlocked()))
!r.canPickThing() && !(r.leftIsBlocked() && r.rightIsBlocked()
!r.canPickThing() && (!r.leftIsBlocked() || !r.rightIsBlocked()
!r.canPickThing() && (r.leftIsClear() || r.rightIsClear()
```

Consider the following code fragment and situation:

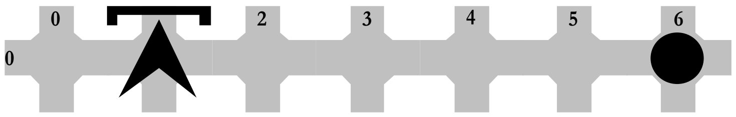
```
if (this.frontIsClear() && this.thingOnSixthAvenue())
{  this.putAllThings();
}
```

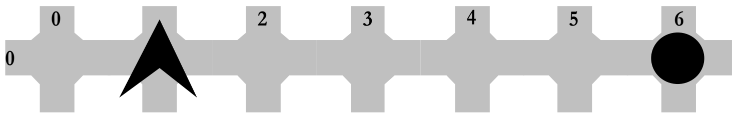


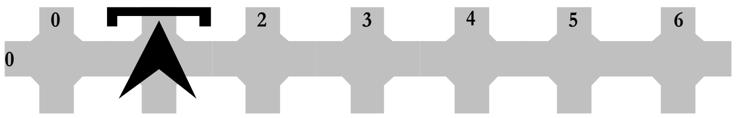
Does the robot need to check for a thing on 6th Avenue?

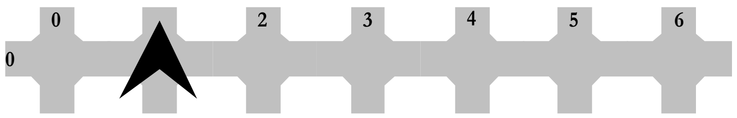
In which of the following does the robot travel to 6th Avenue?

```
if (this.frontIsClear() && this.thingsOnSixthAvenue())
{ …
}
```

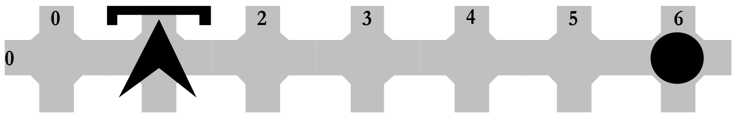
a) 

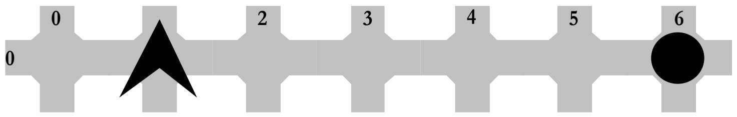
b) 

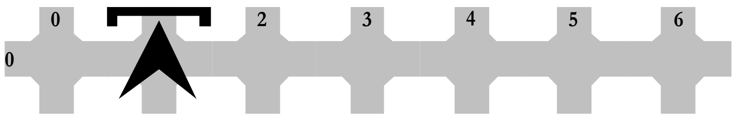
c) 

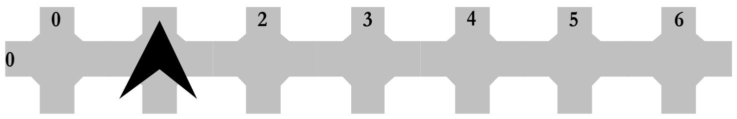
d) 

```
if (this.frontIsClear() || this.thingsOnSixthAvenue())
{ …
}
```

a) 

b) 

c) 

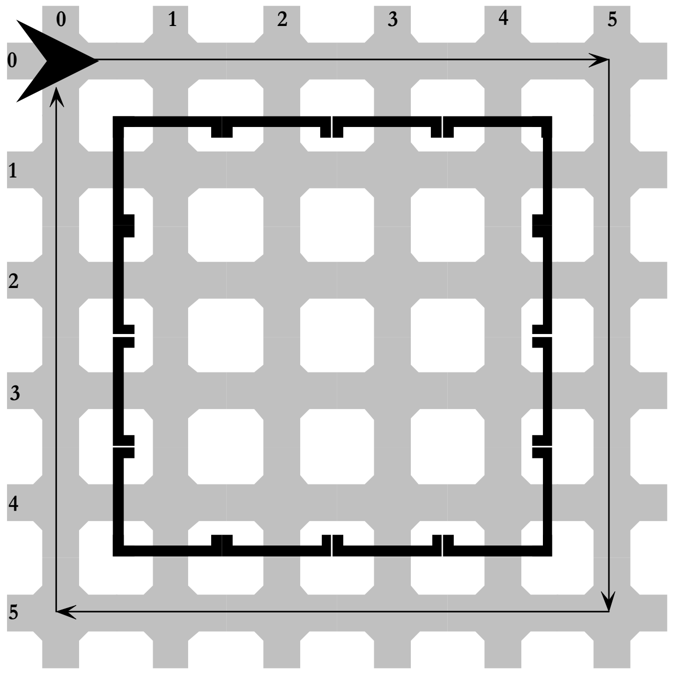
d) 

5.5.1: Using a for Statement

In the figure below, the robot needs to move along exactly four sides. On each side it needs to move exactly five times.

When the exact number of iterations is know *before* the loop begins, use a **for** loop:

```
for(int «counter»=0; «counter»<«limit»; «counter»=«counter»+1)
{ «statements to repeat»
}
```



Example:

```
public void moveAroundSquare()
{  for(int side = 0; side < 4; side = side + 1)
   {  this.moveAlongSide();
      this.turnRight();
   }
}

private void moveAlongSide()
{  for(int moves = 0; moves < 5; moves = moves + 1)
   {  this.move();
   }
}
```

5.5.2: Using a do-while Loop

A **do-while** loop performs its test at the end of the loop, meaning the body is always executed at least once.

```
do
{ «statements to repeat»
} while («test»);
```

5.5.3: Using a while-true Loop

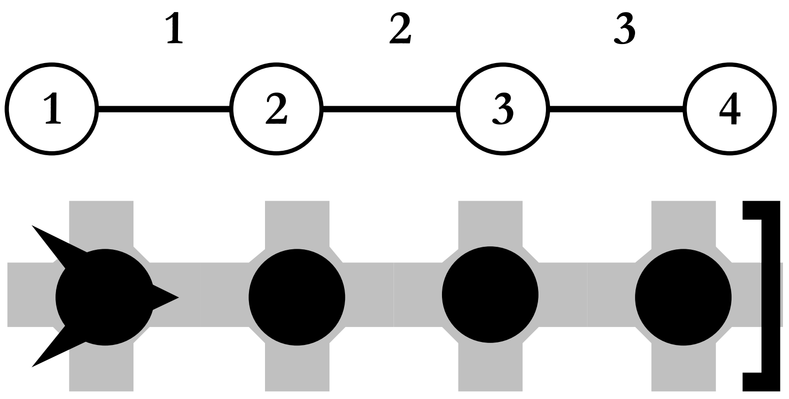
A **while-true** loop repeats until a **break** statement is executed. Execution then continues with the statement immediately following the loop. The **break** statement is always protected with an **if** statement that determines whether the loop should end.

```
while (true)
{ «optional statements1»
  if («test1»)   { break; }
  «optional statements2»
  …
  if («testN»)   { break; }
  «optional statementsN+1»
}
```

5.5.3: Using a while-true Loop

Consider the fence-post problem. We solved it earlier with the following method:

```
public void clearThingsToWall()
{  while (this.frontIsClear())
   {  this.pickThing();
      this.move();
   }
   this.pickThing();
}
```

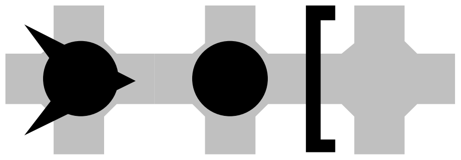


Using a **while-true** loop, this could be written as follows:

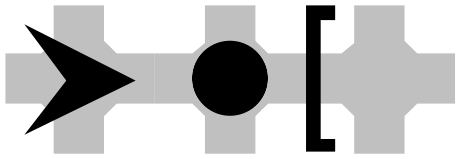
```
public void clearThingsToWall()
{  while (true)
   {  this.pickThing();
      if (this.frontIsBlocked())    {  break;  }
      this.move();
   }
}
```

5.3.3: Using a while-true Loop

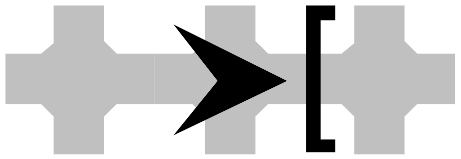
```
while (true)
{  this.pickThing();
   if (this.frontIsBlocked())  {  break;  }
   this.move();
}
```



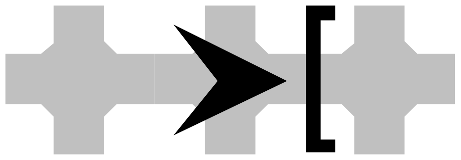
```
while (true)
{  this.pickThing();
   if (this.frontIsBlocked())  {  break;  }
   this.move();
}
```



```
while (true)
{  this.pickThing();
   if (this.frontIsBlocked())  {  break;  }
   this.move();
}
```



```
while (true)
{  this.pickThing();
   if (this.frontIsBlocked())  {  break;  }
   this.move();
}
```

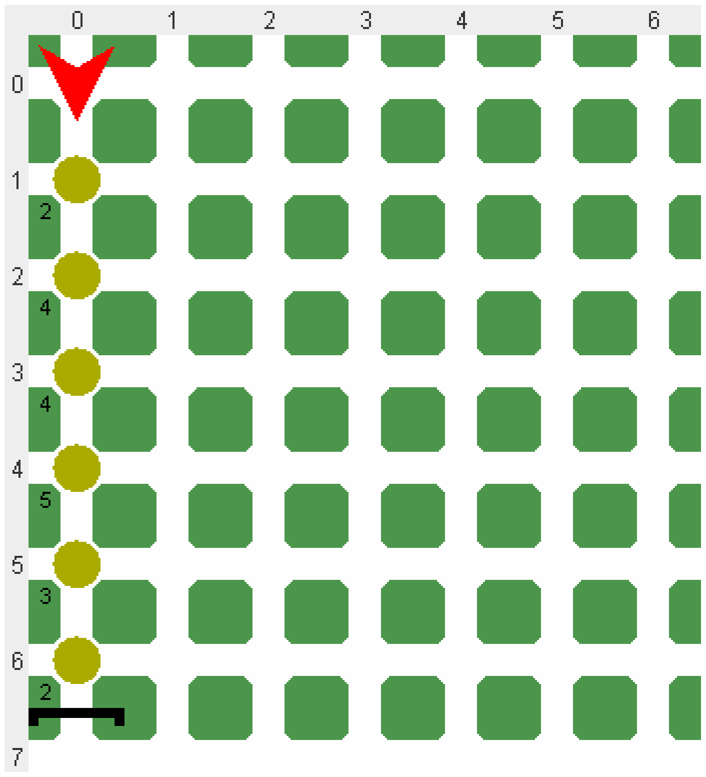


5.5.4: Choosing an Appropriate Loop

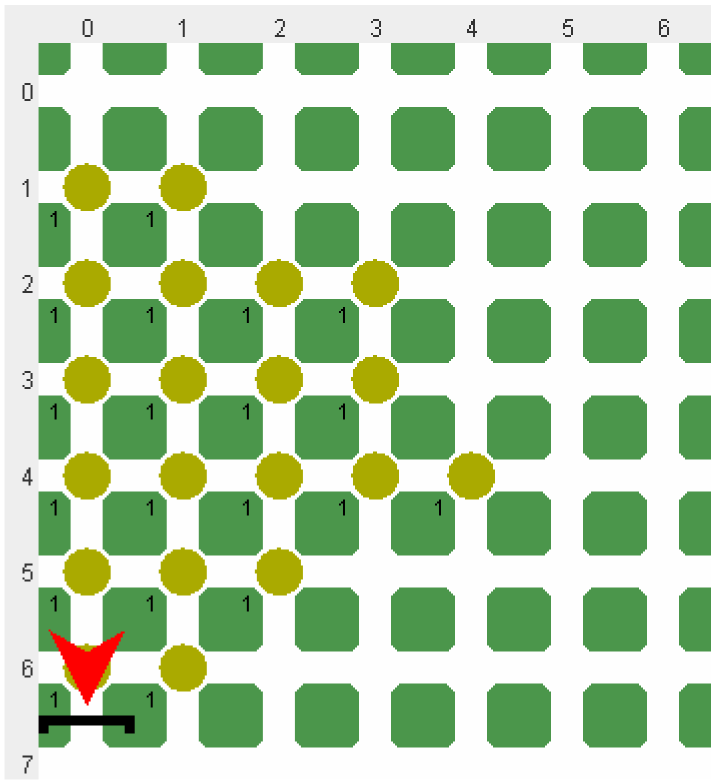
If…	**Then…**
a parameter refers to the number of times the loop will execute and the value is not needed for other purposes…	use a count-down loop or a **for** statement.
the number of times the loop will execute is known before the loop is entered...	use a **for** statement.
the loop might execute zero times…	use a **while** statement.
the loop has a single test that is relatively simple that appears at the top of the loop…	use a **while** statement.
the loop always executes at least once…	use a **do-while** statement.
the loop executes an extra “half” time for a fence-post problem…	use a **while-true** loop.
the loop has multiple exit tests or a complex test that can be more easily understood as separate tests…	use a **while-true** loop.

Case Study: Histograms

Write a **HistogramBot** which will turn piles of **Things** (the data) into a histogram (bar chart). The data is always on Avenue 0, starting with Street 1. The end of the data is signaled with a wall, as shown in the initial situation. The result of calling the robot's **makeChart** method is shown in the final situation. It is not known whether the robot already has things in its backpack.



Initial Situation



Final Situation

Case Study: main Method

```
import becker.robots.*;

/** Make a histogram (bar chart) from data (things) on Avenue 0.
*
*  @author Byron Weber Becker */
public class HistogramMain
{
   public static void main(String[ ] args)
    {
      // a file that isn't found will result in a file chooser being displayed
      City chart = new City("");
      chart.showThingCounts(true);
      HistogramBot histo = new HistogramBot(chart);

      histo.makeChart();
    }
}
```

Case Study: Beginning HistogramBot

```
import becker.robots.*;

/** A kind of robot that will make a histogram (bar chart) from data (things) on Avenue 0.
*  The end of the data is marked with a wall.
*
* @author Byron Weber Becker */
public class HistogramBot extends Robot
{
    /** Create a HistogramBot at the origin of the given city, facing south.
     * @param aCity The city containing the data for the histogram. */
    public HistogramBot(City aCity)
    {  super(aCity, 0, 0, Direction.SOUTH);
    }

    /** Make the histogram from the data supplied on Avenue 0. */
    public void makeChart()
    {
    }
}
```

Case Study: makeChart

```
import becker.robots.*;

public class HistogramBot extends Robot
{
   public HistogramBot(City aCity)…              // done

   /** Make the histogram from the data supplied on Avenue 0. */
   public void makeChart()
   {
      while (this.frontIsClear())
      {  this.move();
         this.makeBar();
      }
   }

   /** Make one bar of the histogram.  */
   protected void makeBar()
   {
   }
}
```

Case Study: makeBar

```
import becker.robots.*;

public class HistogramBot extends Robot
{  public HistogramBot(City aCity)…            // done
   public void makeChart()…                    // done

   /** Make one bar of the histogram.  */
   protected void makeBar()
   {  int numPoints = this.pickAndCountThings();
      if (numPoints > 0)
      {  this.distributePoints(numPoints);
      }
   }

   /** Pick up and count all of the things on this intersection.
    * @return The number of things picked up. */
   private int pickAndCountThings()
   {  return 0;
   }
   /** Distribute num data items (Things), one per intersection. */
   protected void distributePoints(int num)
   {
   }
}
```

Case Study: pickAndCountThings

```
import becker.robots.*;

public class HistogramBot extends Robot
{  public HistogramBot(City aCity)…          // done
   public void makeChart()…                  // done
   protected void makeBar()…                 // done

   /** Pick up and count all of the things on this intersection.
    * @return The number of things picked up. */
   private int pickAndCountThings()
   {  int numThings = 0;
      while (this.canPickThing())
      {  this.pickThing();
         numThings = numThings + 1;
      }
      return numThings;
   }

   /** Distribute num data items (Things), one per intersection. */
   protected void distributePoints(int num)
   {
   }
}
```

Case Study: distributePoints

```
import becker.robots.*;
public class HistogramBot extends Robot
{  public HistogramBot(City aCity)…                    // done
   public void makeChart()…                            // done
   protected void makeBar()…                           // done
   private int pickAndCountThings()…                   // done

   /** Distribute num data items (Things), one per intersection. */
   protected void distributePoints(int num)
   {  this.turnLeft();
      this.putDown(num);
      this.turnAround();
      this.move(num);
      this.turnLeft();
   }

   /** Put down num things, one per intersection. */
   private void putDown(int num)
   {
   }

   /** Move howFar times */
   private void move(int howFar)…
   private void turnAround()…
}
```

Case Study: putdown and move

```
import becker.robots.*;
public class HistogramBot extends Robot
{  public HistogramBot(City aCity)…                          // done
   public void makeChart()…                                  // done
   protected void makeBar()…                                 // done
   private int pickAndCountThings()…                         // done
   protected void distributePoints(int num)                  // done

   /** Put down num things, one per intersection. */
   private void putDown(int num)
   {  for(int i = 0; i < num; i = i + 1)
      {  this.putThing();
         this.move();
      }
   }

   /** Move howFar times */
   private void move(int howFar)…
   {  for(int i = 0; i < howFar; i = i + 1)
      {  this.move();
      }
   }
   private void turnAround()
   {  this.turnLeft();       this.turnLeft();
   }
}
```

5.6: Coding with Style

The style of your code has a large impact on its understandability.

Three important guidelines:

- Use stepwise refinement to avoid having deeply nested statements or long sequences of statements.
- Use positively stated, simple Boolean expressions. Transform complex Boolean expressions with:
    - Predicates to encapsulate complex expressions
    - Test reversal
    - Top factoring
    - Bottom factoring
- Indent your code so the visual structure reflects the logical structure.

5.6.2: Positively Stated Simple Expressions

Use predicates:

Instead of…	Use…

`while (!this.frontIsClear())`
`{ …`
`}`
	`while (this.frontIsBlocked())`
	`{ …`
	`}`

`if (this.getDirection() ==`
`                    Direction.SOUTH))`
`{ …`
`}`
	`if (this.isFacingSouth())`
	`{ …`
	`}`
	or
	`if (this.isFacing(Direction.SOUTH))`
	`{ …`
	`}`

`if (this.frontIsBlocked() &&`
`     this.leftIsBlocked() &&`
`     this.rightIsBlocked())`
`{ …`
`}`
	`if (this.completelyBlocked())`
	`{ …`
	`}`

5.6.2: Positively Stated Simple Expressions

Use test reversal:

Instead of…	Use…

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

`if (this.canPickThing())`
`{  // do nothing`
`} else`
`{  this.turnLeft();`
`}`
	`if (!this.canPickThing())`
	`{  this.turnLeft();`
	`} else`
	`{  // do nothing`
	`}`
	
	or, better still
	`if (!this.canPickThing())`
	`{  this.turnLeft();`
	`}`

5.6.2: Positively Stated Simple Expressions

Use bottom factoring:

Instead of…

```
if (this.canPickThing())
{  this.pickThing();
   this.turnAround();
} else
{  this.putThing();
   this.turnAround();
}
```

Use…

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

} else
{  this.putThing();

}
this.turnAround();
```

5.6.2: Positively Stated Simple Expressions

Use top factoring:

Instead of…

```
if (this.canPickThing())
{  this.turnAround();
   this.pickThing();
} else
{  this.turnAround();
   this.putThing();
}
```

Use…

```
this.turnAround();
if (this.canPickThing())
{
   this.pickThing();
} else
{
   this.putThing();
}
```

Instead of…

```
if (this.isFacingNorth())
{  this.turnAround();
   this.pickThing();
} else
{  this.turnAround();
   this.putThing();
}
```

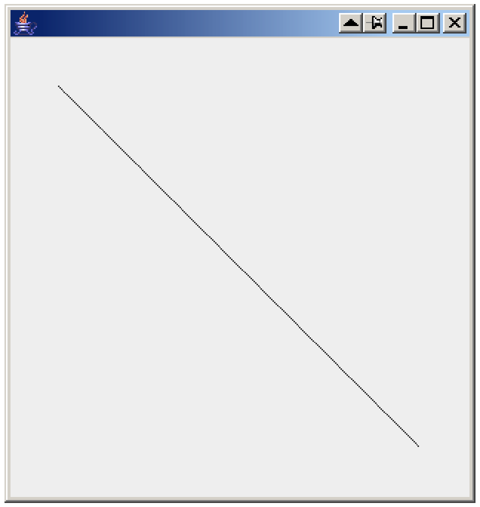
Use…

```
// WRONG!
this.turnAround();
if (this.isFacingNorth())
{
   this.pickThing();
} else
{
   this.putThing();
}
```

Application: Using Loops to Draw (1/3)

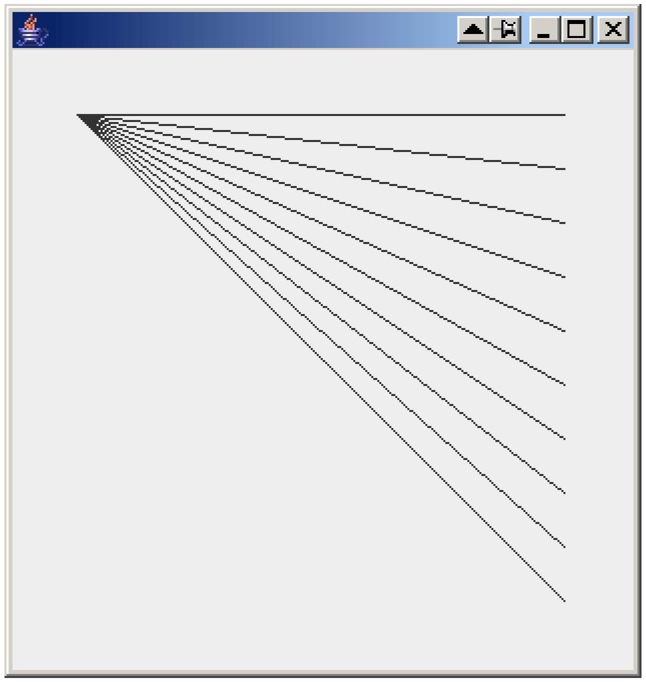
```
import javax.swing.*;
import java.awt.*;

/** Create a component that paints our "art".
* @author Byron Weber Becker */
public class ArtComponent extends JComponent
{
   public ArtComponent()
   {  super();
      this.setPreferredSize(new Dimension(300,300));
   }

   /** Paint the component with our "art". */
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);

      // Standard stuff to scale the image.
      Graphics2D g2 = (Graphics2D) g;
      g2.scale(this.getWidth()/11, this.getHeight()/11);
      g2.setStroke(new BasicStroke(1.0F/this.getWidth()));

      // draw our "art"
      g.drawLine(1, 1, 10, 10);
   }
}
```

Add this component to the content pane of a **JPanel**.



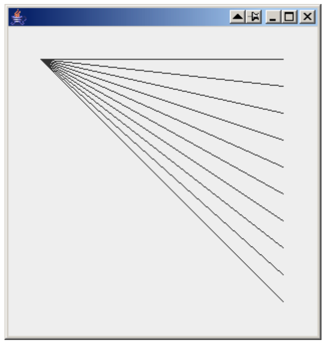
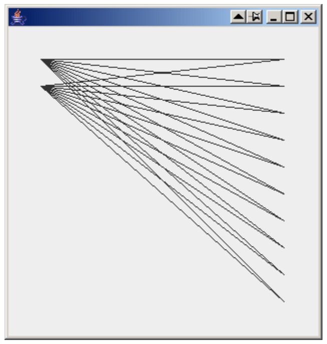
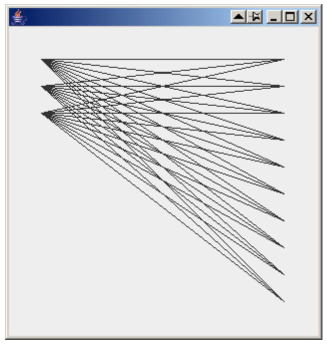
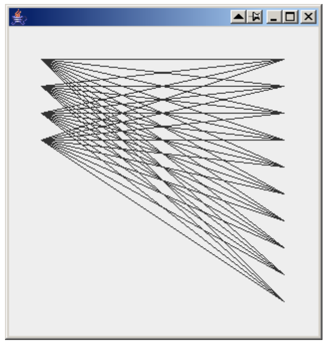
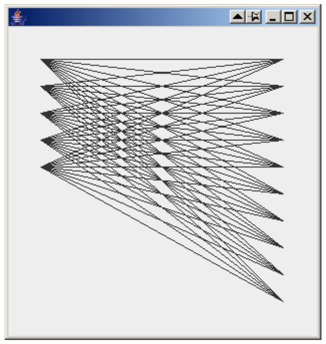
Application: Using Loops to Draw (2/3)

```
public class ArtComponent extends
JComponent
{
   public ArtComponent()…

   /** Paint the component with our "art". */
   public void paintComponent(Graphics g)
   {  super.paintComponent(g);

      // Standard stuff to scale the image.
      Graphics2D g2 = (Graphics2D) g;
      g2.scale(this.getWidth()/11, this.getHeight()/11);
      g2.setStroke(new BasicStroke(1.0F/this.getWidth()));

      // draw our "art"
      for (int line = 1; line <= 10; line = line + 1)
      {  g.drawLine(1, 1, 10, line);
      }
   }
}
```



Application: Using Loops to Draw (3/3)

```
public class ArtComponent extends JComponent
{
   public void paintComponent(Graphics g)
   { …
      // draw our "art"
      for (int left = 1; left <= 5; left = left + 1)
      {  for (int right = 1; right <= 10; right = right + 1)
         {  g.drawLine(1, left, 10, right);
         }
      }
   }
}
```

5.8.1: The Loop-and-a-Half Pattern

Name: Loop-and-a-Half

Context: A loop is used for a variation of the fence-post problem where some actions (fence posts) must be performed one more time than other actions (fence sections).

Solution: Use a **while** loop with an extra fence post action afterwards:

```
while («booleanExpression»)
{ «fencePost actions»
  «fenceSection actions»
}
«fencePost actions»
```

Alternatively, use a **while-true** loop:

```
while (true)
{ «fencePost actions»
  if (!«booleanExpression»)  { break; }
  «fenceSection actions»
}
```

Consequences: Some actions are repeated an extra time.

Related Patterns: Variation of Zero or More Times.

5.8.2: The Temporary Variable Pattern

Name: Temporary Variable

Context: A value must be stored for later use within the same method.

Solution: Use a temporary variable. Declare with

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

Using ***«name»*** causes the variable's value to be used in the expression.

An example:

```
   public int numBlockedDirections()
   {  int numWalls = 0;
      for (int turns = 0; turns < 4; turns = turns + 1)
      {  if (!this.frontIsClear())
         {  numWalls = numWalls + 1;
         }
         this.turnLeft();
      }
      return numWalls;
   }
```

Consequences: The variable will remember a value within the smallest enclosing block.

Related Patterns: Always occurs within a method pattern.

5.8.3: The Counting Pattern

Name: Counting

Context: The number of events must be counted (number of moves, times something is picked up, etc).

Solution: Increment a temporary variable each time the event happens.

```
int «counter» = 0;
while ()
{ «statements»
  «counter» = «counter» + 1;
}
```

Variations of this pattern may increment ***«counter»*** only if a certain condition is true or may use a different looping strategy.

Consequences: ***«counter»*** records the number of events since it was initialized.

Related Patterns: This pattern uses the Temporary Variable pattern plus a looping pattern such as Zero or More Times.

5.8.4: The Query Pattern

Name: Query

Context: A calculation that yields a single value is required, particularly if the calculation requires several steps, is complicated, the program's readability is improved by giving it a name, or the calculation is used more than once in the program.

Solution: Write a method with a return value:

```
«accessModifier» «returnType» «queryName»(«optParams»)
{ «optional statements»
  «returnType» answer = «expression»;
  «optional statements»
  return answer;
}
```

Consequences: Queries make the code easier to understand.

Related Patterns: This is a specialization of other method creation patterns such as Parameterless Command. The Simple Predicate and Predicate patterns are specializations of this pattern.

5.8.5: The Predicate Pattern

Name: Predicate

Context: A Boolean expression is hard to read or the result can't be calculated with a single expression.

Solution: Place the expression and any extra processing into a specialized version of the Query pattern where the return type is **boolean**. For example:

```
public boolean rightIsBlocked()
{  this.turnRight();
   boolean answer = this.frontIsBlocked();
   this.turnLeft();
   return answer;
}
```

Consequences: The processing required is encapsulated in a reusable query. Appropriately named, code becomes easier to understand.

Related Patterns: This pattern is a specialization of the Query pattern. It is often used in the test for the Once or Not At All, Zero or More Times, and Either This or That patterns. The Simple Predicate pattern studied earlier is a simplified version of this pattern.

5.8.6: The Cascading-if Pattern

Name: Cascading-if

Context: One of several groups of statements must be executed.

Solution: Order the tests from the most specific to the most general, pairing each test with the appropriate actions.

```
if («test1»)
{ «statementGroup1»
} else if («test2»)
{ «statementGroup2»
...
} else if («testN»)
{ «statementGroupN»
} else
{ «defaultStatements»
}
```

Consequences: The tests are executed in order from 1 to N. The first one that returns true will cause the associated statement group to be executed.

Related Patterns: Once or Not At All and Either This or That are simpler (and more common) versions of this pattern.

5.8.7: The Counted Loop Pattern

Name: Counted Loop

Context: A group of statements must be executed a specific number of times, a number known before execution of the loop begins.

Solution: Use a **for** statement:

```
int howFar = this.getAvenue();
for (int i = 0; i < howFar; i = i + 1)
{  this.move();
}
```

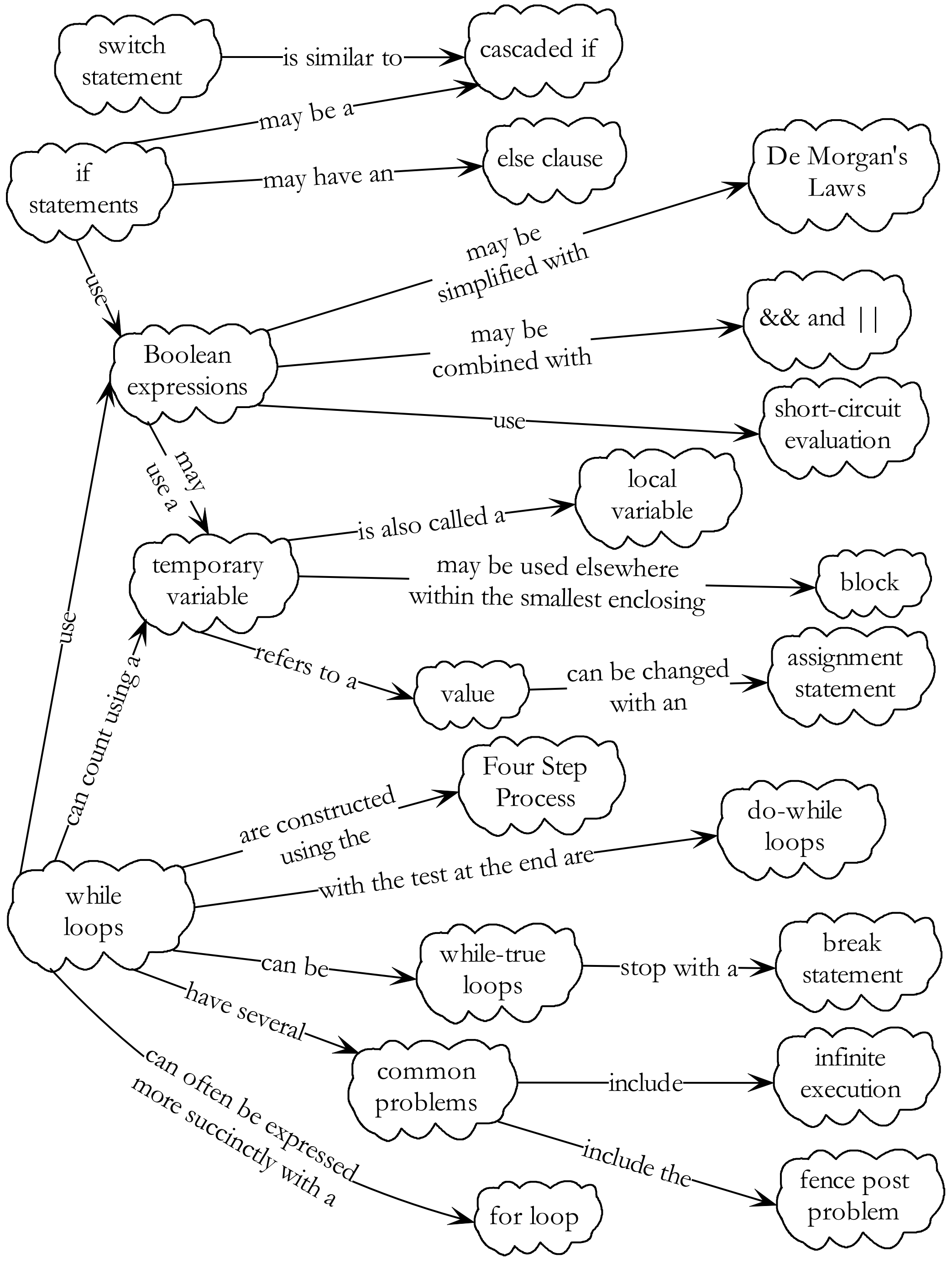
In general,

```
for (int «counter» = 0; «counter» < «limit»;
                                 «counter» = «counter» + 1)
{ «statements»
}
```

Consequences: The ***«statements»*** are executed ***«limit»*** times.

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

5.9: Concept Map



4.9: Summary

We have learned:

- how to avoid common looping errors such as the fence-post problem and how to use a four-step process to write loops.
- how to use temporary variables to remember information for later use within a method, including tasks such as counting, storing the result of a query, and writing a query.
- how statements such as **if** and **while** can be nested inside each other.
- how nesting **if-else** statements in a particular pattern lets us choose exactly one group of statements to execute.
- how to combine, evaluate, and simplify Boolean expressions.
- how to use several looping variations including the **for** statement, **do-while** statement, and **while-true** loops.
- how to use positively stated simple expressions to make our code more readable.