Java:

Learning to Program with Robots

Chapter 01: Programming with Objects

Chapter Objectives

After studying this chapter, you should be able to:

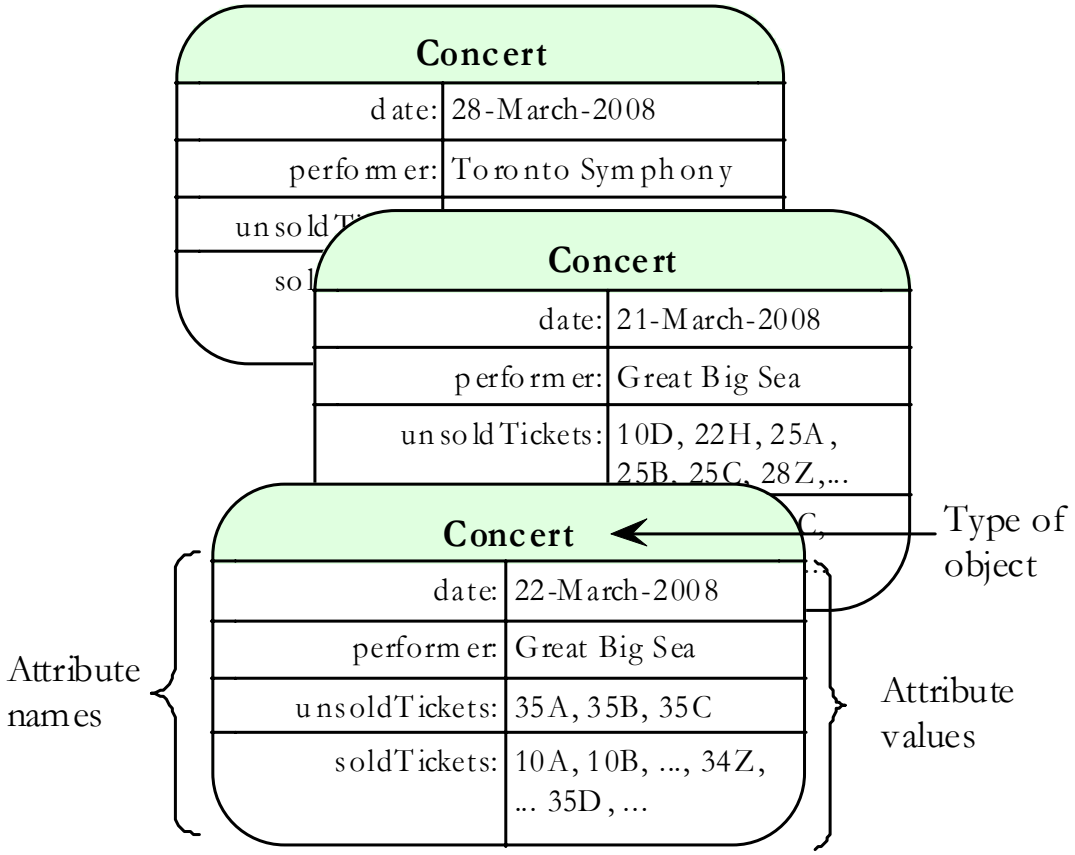
- Describe models
- Describe the relationship between objects and classes
- Understand the syntax and semantics of a simple Java program
- Write object-oriented programs that simulate robots
- Understand and fix errors that can occur when constructing a program
- Read documentation for classes
- Apply the concepts learned with robots to display a window as used in a graphical user interface

- Models are simplified descriptions containing information and operations used to solve problems

Model	**Information**	**Operations**
Concert	Who's performing Performance Date Which seats are sold	Sell a ticket Count tickets sold
Schedule	List of tasks to perform, each with estimated time	Insert or delete a task Calc estimated finish time
Restaurant Seating	Occupied tables Unoccupied tables # of seats at each table	Mark a table occupied Mark a table unoccupied

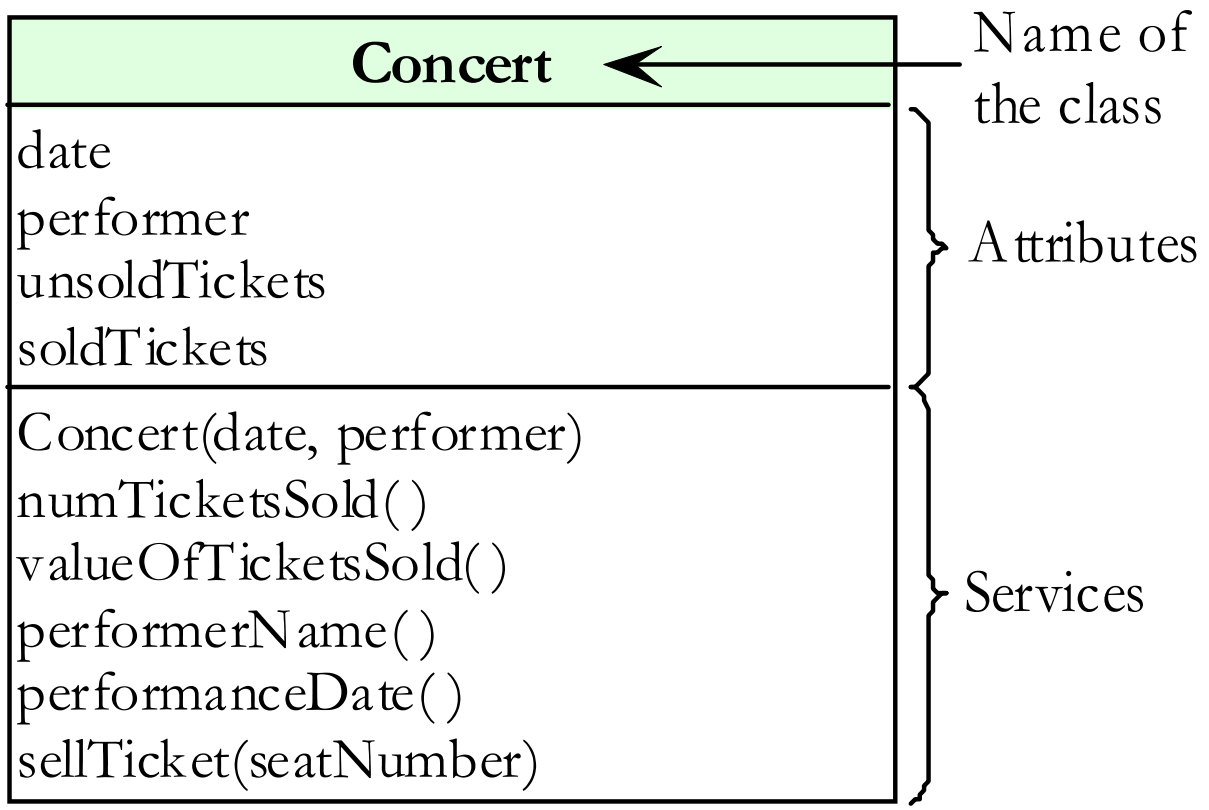
- Models can be maintained:
  - in our heads
  - with paper and pencil
  - with software

- Java programs are composed of software objects
  - Software objects have:
    - Information, called *attributes*
    - *Services* that either change the attributes (a *command*) or answer a question about attributes (a *query*)
  - A program may have many similar objects
  - Objects can be visualized with an *object diagram*
    - shows attribute names and values

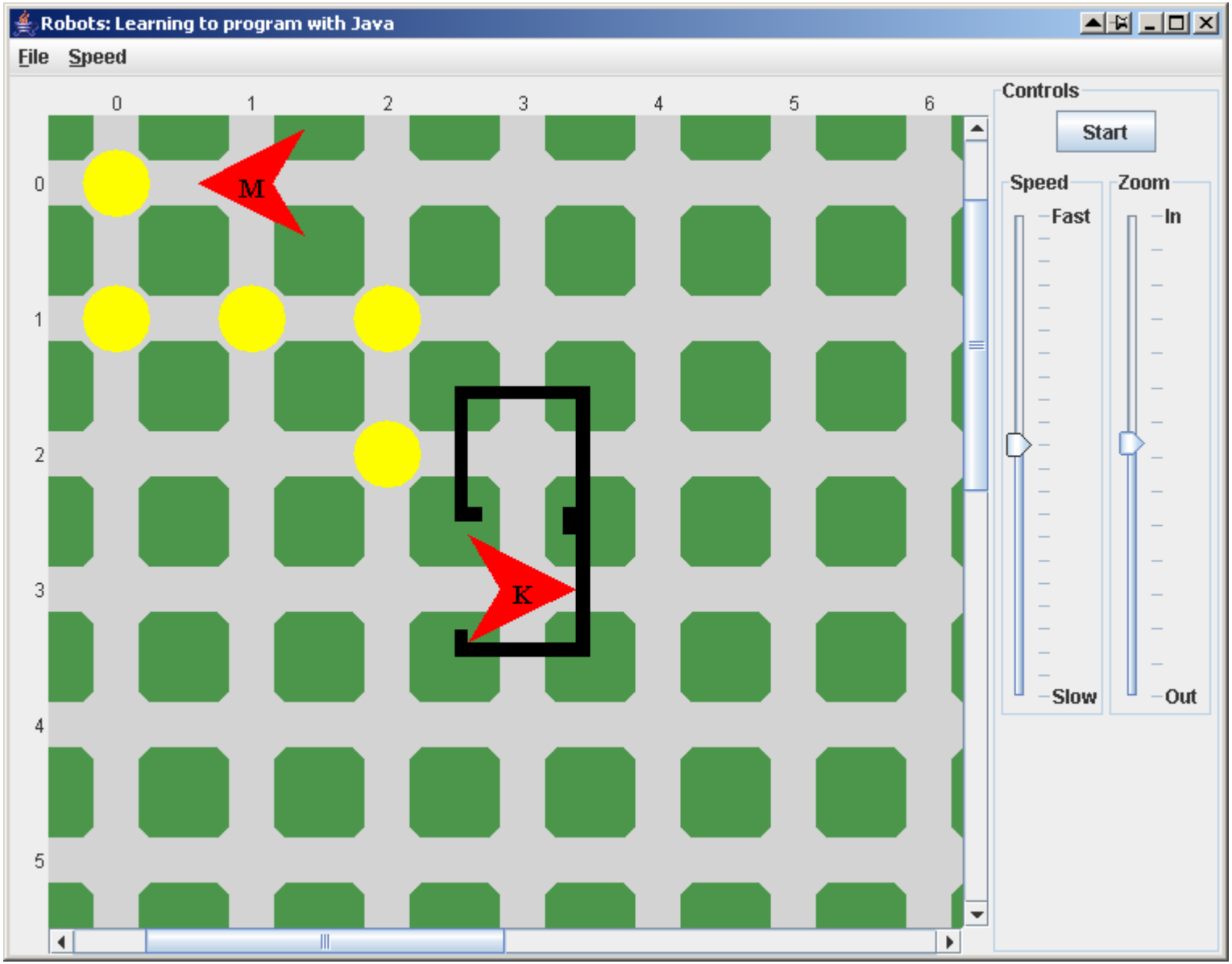


- A group of objects that
  - have the same kind of information
  - offer the same services

  are called a *class*
- Classes are represented with a *class diagram*



1.2: Understanding Karel's World

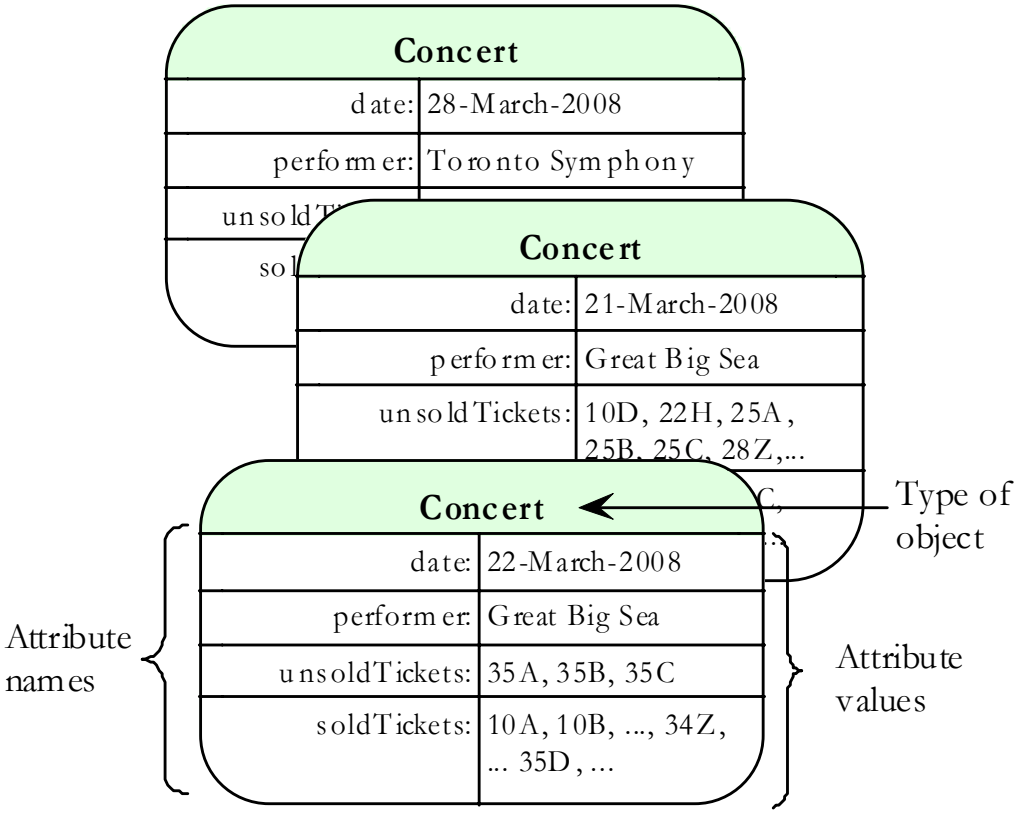


Quick Quiz

1. Draw an object diagram for the robot labelled “M” on the previous slide.

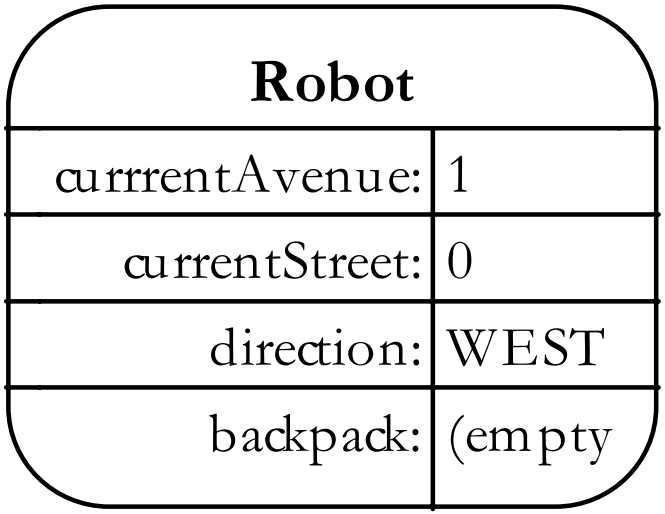
   Hint: Three **Concert** object diagrams are shown to the right.

2. Draw your object diagram again after the robot has executed the following commands:
   move()
   pickThing()

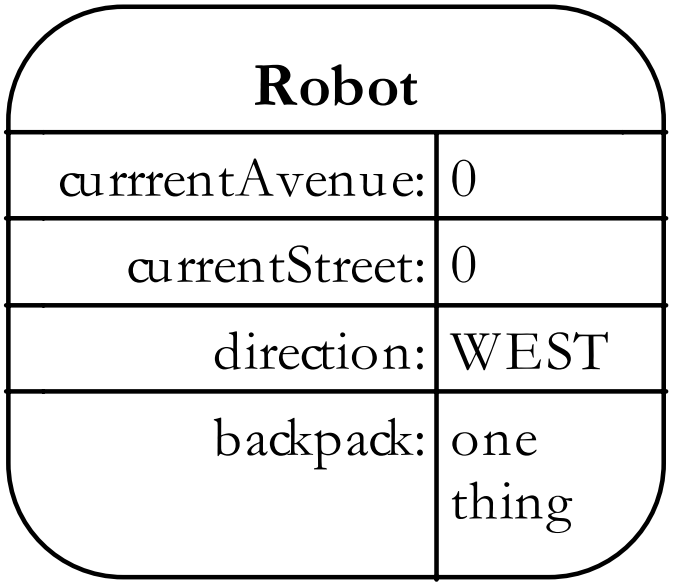


Quick Quiz Solutions

1.

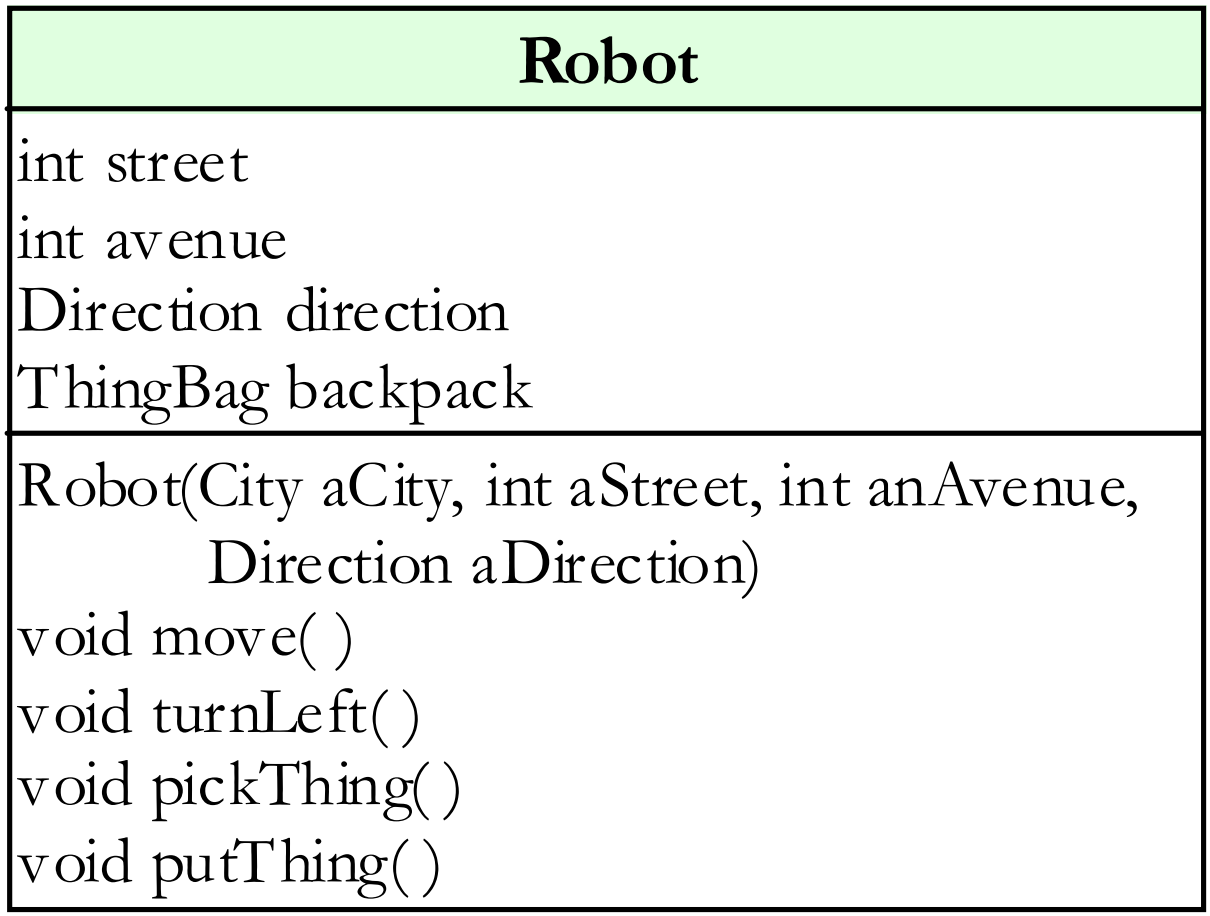


2.

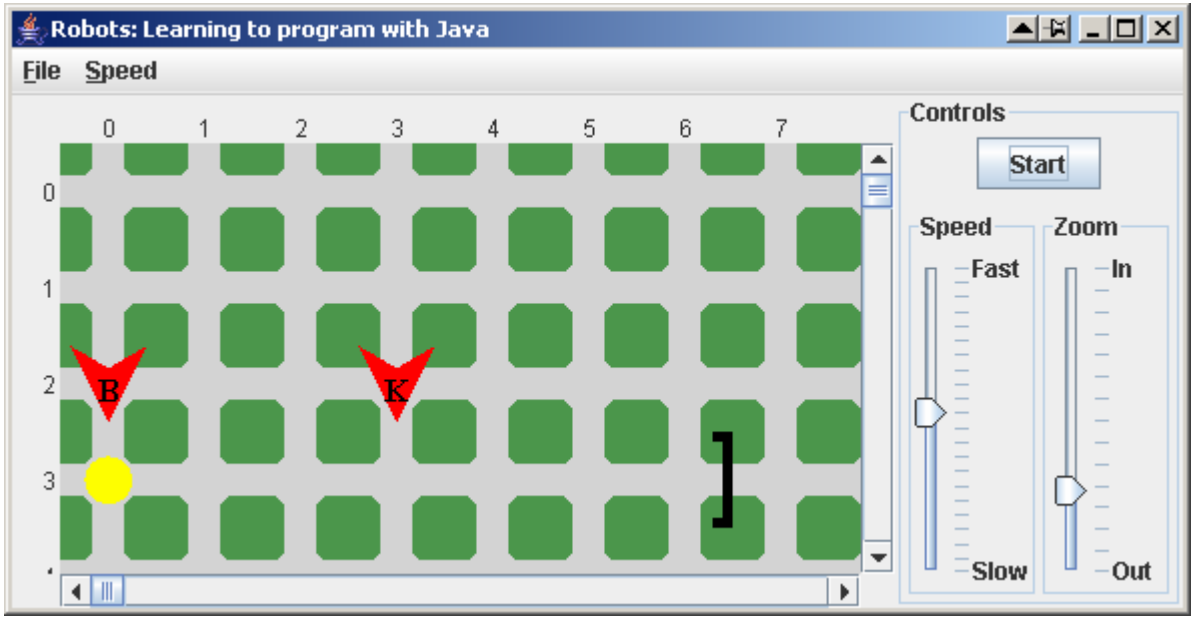


Solutions may also contain attributes for the label and color.

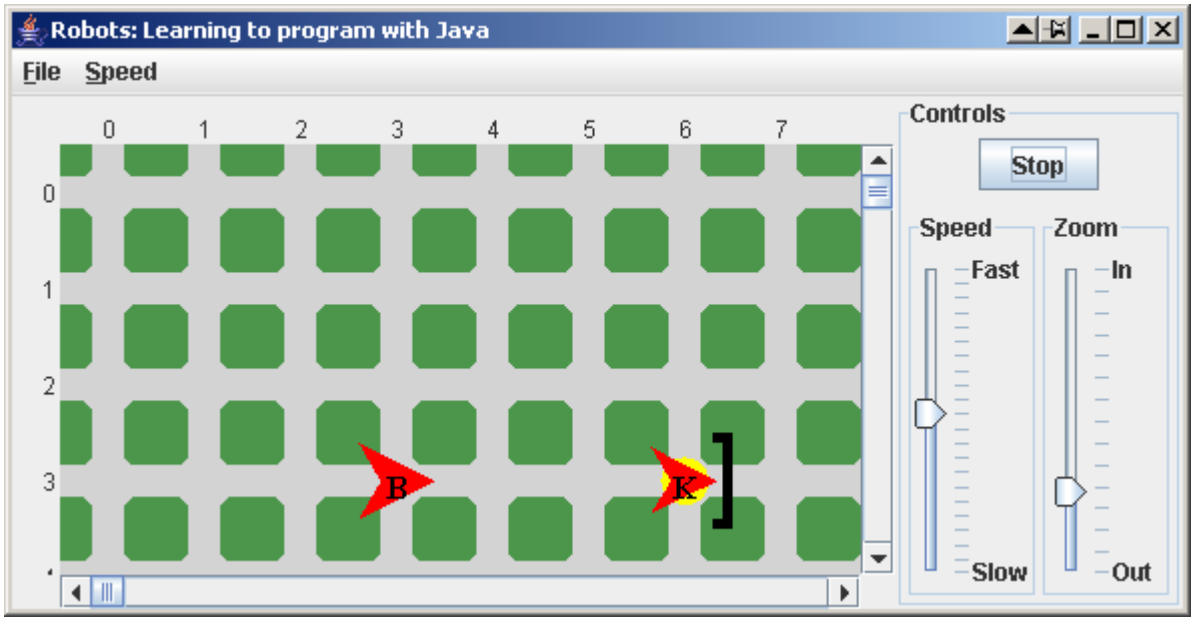
A class diagram for the robot class:



Two robots running a “relay.”



Initial Situation



Final Situation

“B” picks up the baton and takes it to “K”, who finishes the race.

1.4: An Example Program (2/3)

```
// Set up the initial situation
City beijing = new City();
Robot ben = new Robot(beijing, 2, 0, Direction.SOUTH);
Robot karel = new Robot(beijing, 2, 3, Direction.SOUTH);
Thing baton = new Thing(beijing, 3, 0);
Wall finishLine = new Wall(beijing, 3, 6, Direction.EAST);
karel.setLabel("K");
ben.setLabel("B");

// Run the relay
ben.move();  // bwb
ben.turnLeft();
ben.pickThing();
ben.move();
ben.move();
ben.move();
ben.putThing();

karel.move();
karel.turnLeft();
karel.pickThing();
karel.move();
karel.move();
karel.move();
karel.putThing();
```

1.4: An Example Program (3/3)

```
import becker.robots.*;

public class RobotRelay
{

   public static void main(String[ ] args)
   {
      Code on the previous slide goes here.

      All of the code goes into a computer file named
      RobotRelay.java

   }
}
```

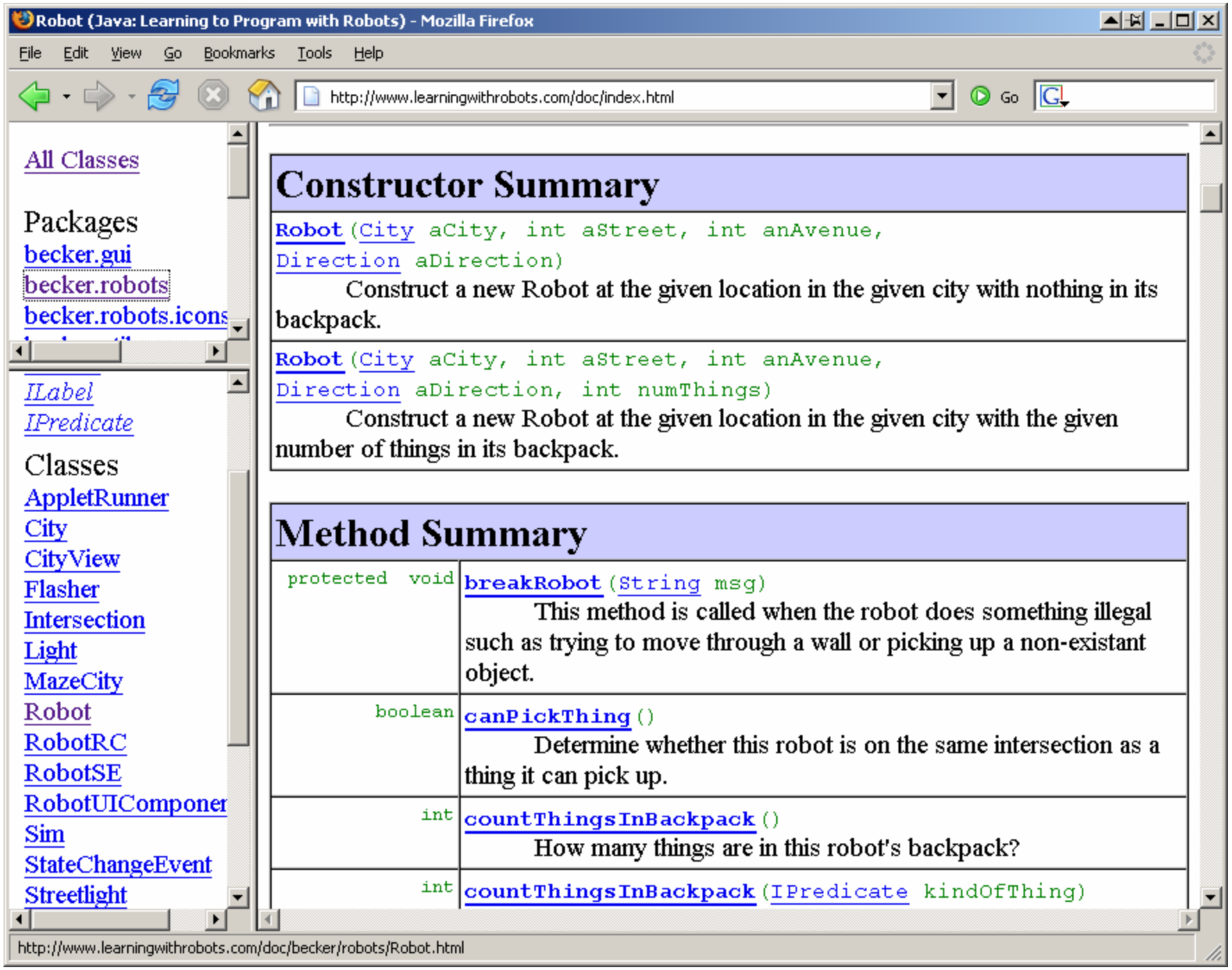
1.4.5: Tracing a Program (1/2)

	ben				karel				baton	
Program Stmt	str	ave	dir	bp	str	ave	dir	bp	str	ave
	2	0	S	--	2	3	S	--	3	0
ben.move();										
	3	0	S	--	2	3	S	--	3	0
ben.turnLeft();										
	3	0	E	--	2	3	S	--	3	0
ben.pickThing();										
	3	0	E	ba	2	3	S	--	3	0
ben.move();										
	3	1	E	ba	2	3	S	--	3	1
ben.move();										
	3	2	E	ba	2	3	S	--	3	2
ben.move();										
	3	3	E	ba	2	3	S	--	3	3

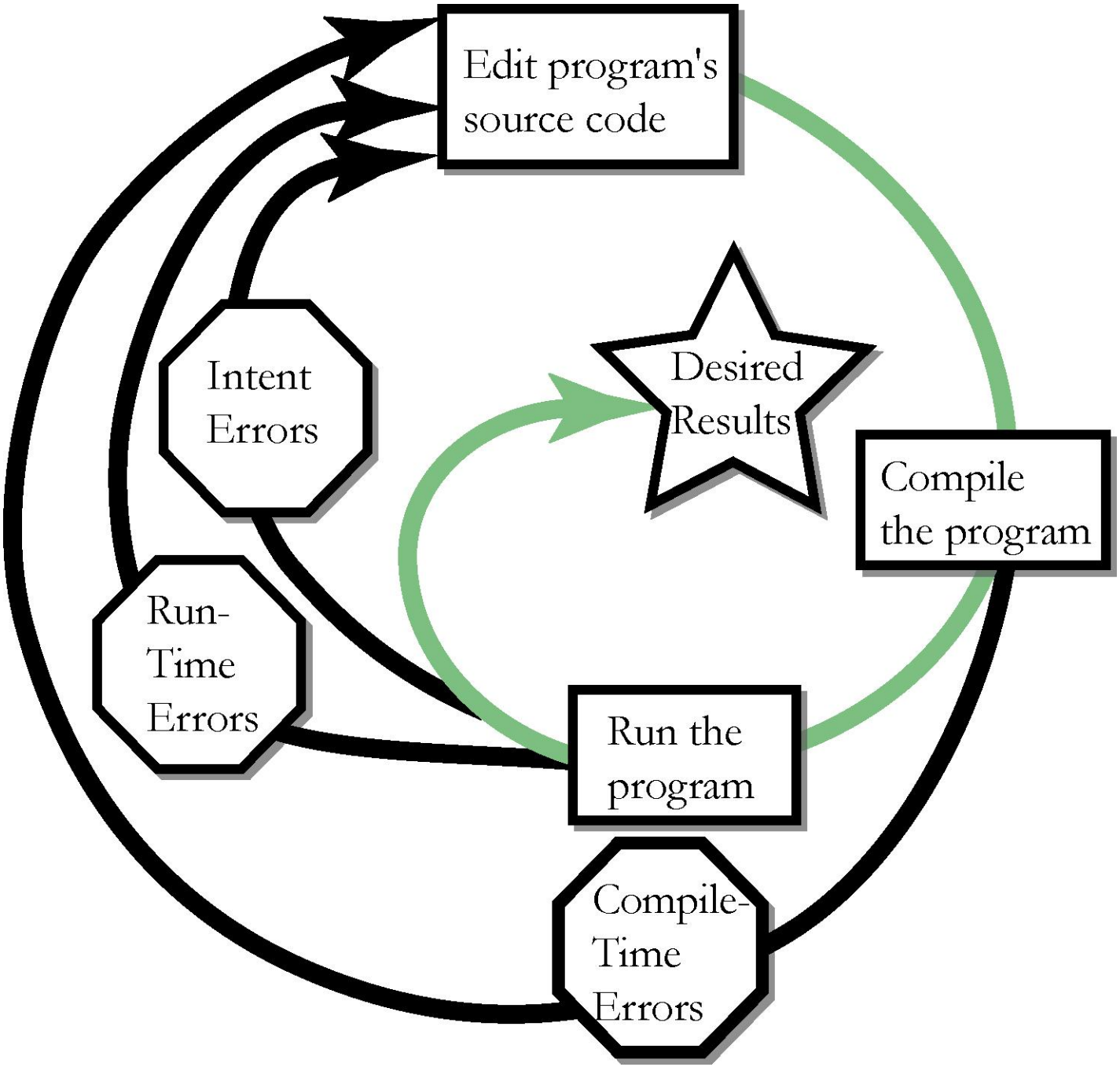
1.4.5: Tracing a Program (2/2)

	ben				karel				baton	
Program Stmt	str	ave	dir	bp	str	ave	dir	bp	str	ave
	3	3	E	ba	2	3	S	--	3	3
ben.putThing();										
	3	3	E	--	2	3	S	--	3	3
karel.move();										
	3	3	E	--	3	3	S	--	3	3
karel.turnLeft();										
	3	3	E	--	3	3	E	--	3	3
karel.pickThing();										
	3	3	E	--	3	3	E	ba	3	3
karel.move();										
	3	3	E	--	3	4	E	ba	3	4
karel.move();										
etc.	3	3	E	--	3	5	E	ba	3	5

1.4.8: Reading Documentation to Learn More



1.5: Compiling and Executing Programs



1.5.1: Compile-Time Errors

Three kinds of errors:

- Compile-Time Errors
  - The compiler can't translate your program into an executable form because your program doesn't follow the language's rules.
  - Examples:
    - **karel.move;** instead of
      karel.move();
    - **Public class RobotRelay** instead of
      public class RobotRelay
    - Unmatched braces; a **{** without a corresponding **}**
- Run-Time Errors
- Intent (Logic) Errors

Three kinds of errors:

- Compile-Time Errors
- Run-Time Errors
  - The compiler can translate your program and it begins to run, but then an error occurs.
  - Example:
    - Code positions the robot in front of a wall

```
karel.move();
```

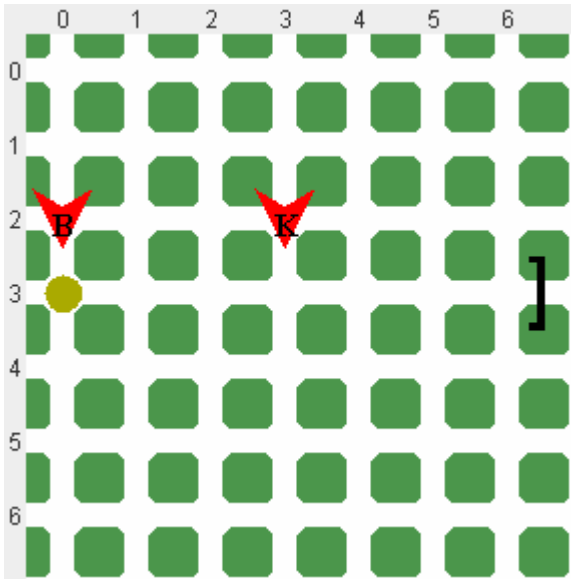
    - The robot is told to move
    - Running into the wall causes the robot to break (a run-time error)

- Intent (Logic) Errors

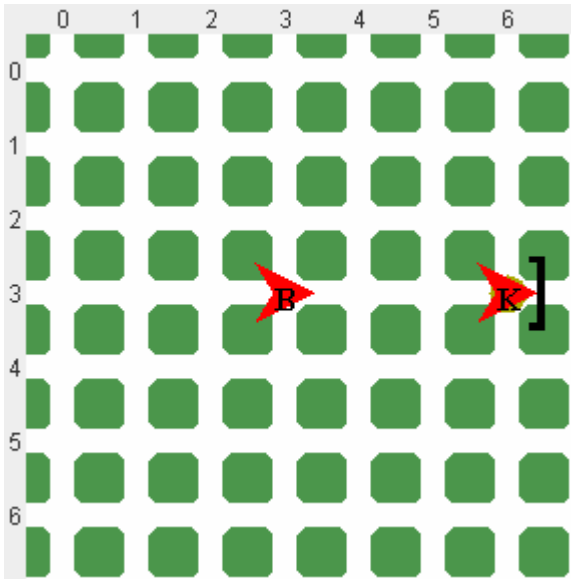
1.5.3: Intent (Logic) Errors

Three kinds of errors:

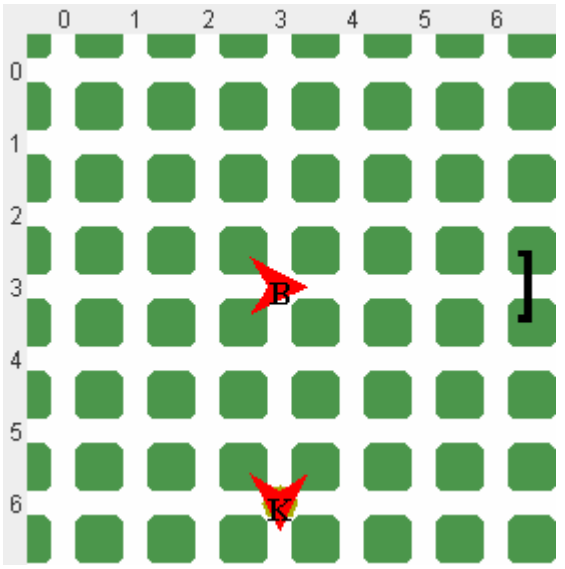
- Compile-Time Errors
- Run-Time Errors
- Intent (Logic) Errors
  - The compiler can translate your program and it runs to completion, but it doesn't do what you want it to.
  - Example: In the relay race, the programmer forgets to instruct karel to turn left after picking up the baton.



Initial Situation



Correct Final



Incorrect Final

Patterns are fragments of code that appear repeatedly.

We give them names and learn them so that:

- we can recognize when they are being used
- we can discuss them easily with others
- we can apply them in new situations

When patterns are used in the text, an icon and the pattern name appears in the margin. Discussed in detail later in the chapter.

```
 7       // Set up the initial situation
 8       City  ny         = new City();
 9       Wall  blockAve0  = new Wall(ny, 0, 2, Direction.WEST);
10       Wall  blockAve1  = new Wall(ny, 1, 2, Direction.WEST);
11       Robot mark       = new Robot(ny, 0, 2, Direction.WEST);
12       Robot ann        = new Robot(ny, 0, 1, Direction.EAST);
13
14       // mark goes around the roadblock
15       mark.turnLeft();
16       mark.move();
17       mark.move();
18       mark.turnLeft();        // start turning right as three turns left
19       mark.turnLeft();
20       mark.turnLeft();        // finished turning right
21       mark.move();
```

1.7.1: The Java Program Pattern

Name: Java Program

Context: Writing a Java program

Solution:

```
import «importedPackage»;     // may have 0 or more import statements

public class «className»
{
   public static void main(String[ ] args)
   { «list of statements to be executed»
   }
}
```

Consequences: A class is defined that can begin the execution of a program.

Related Patterns:

- All the other patterns in Chapter 1 occur within the context of the Java Program pattern.
- All Java programs use this pattern at least once.

1.7.2: The Object Instantiation Pattern

Name: Object Instantiation

Context: An object is needed to carry out various services.

Solution:

Examples:

```
City manila = new City();
Robot karel = new Robot(manila, 5, 3, Direction.EAST);
```

Pattern:

```
«variableType» «variableName» =
                        new «className»(«argumentList»);
```

For now, *«variableType»* and *«className»* will be the same. The *«argumentList»* is optional.

Consequences: A new object is constructed and assigned to the given variable.

Related Patterns: The Command Invocation pattern requires this pattern to construct the object it uses.

1.7.3: The Command Invocation Pattern

Name: Command Invocation

Context: You want an object to perform one of its services.

Solution:

Examples:

```
karel.move();
collectorRobot.pickThing();
```

Pattern:

```
«objectReference».«commandName»(«argumentList»);
```

The *«argumentList»* is optional.

Consequences: The command is performed by the object.

Related Patterns: The Object Instantiation pattern must be preceded by this pattern. The Sequential Execution pattern uses this pattern two or more times.

1.7.4: The Sequential Execution Pattern

Name: Sequential Execution

Context: Your problem can be solved with a sequence of steps where the order of the steps matters.

Solution: List the steps to be executed in order so that each statement appears after all the statements upon which it depends.

For example, the following two program fragments are the same except for their order. They do different things; only one of which is correct in a given context.

```
karel.move();
karel.turnLeft();
```

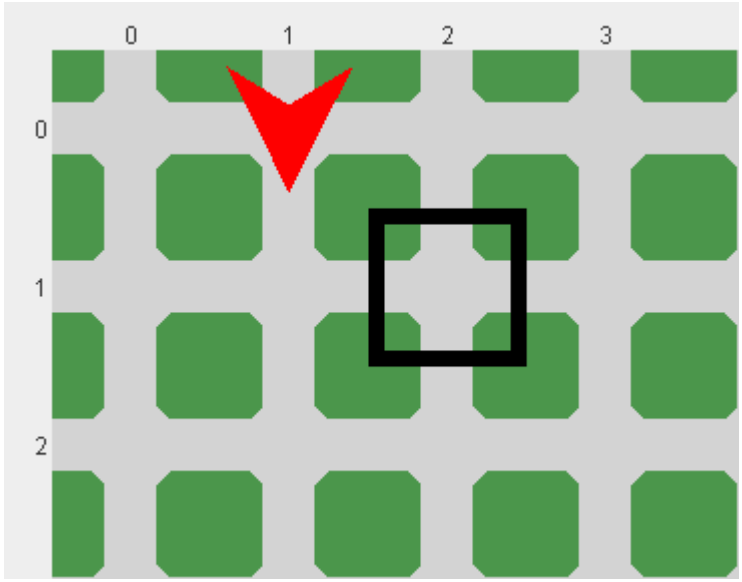
```
karel.turnLeft();
karel.move();
```

Consequences: Each statement is executed in turn. The result usually depends on the statements that have been previously executed.

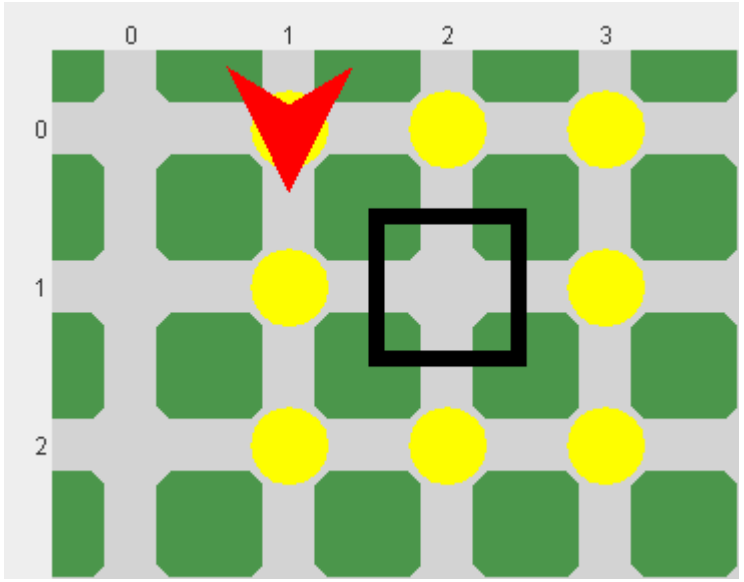
Related Patterns: This pattern uses the Command Invocation pattern two or more times.

Case Study 1: Plant Flowers

You have a garden enclosed with four walls, as shown in the initial situation. You want to plant flowers around it, as shown in the final situation. Program a robot, **karel**, to do this for you.



Initial Situation



Final Situation

Questions:

- Where do the “flowers” (**Thing** objects) come from?
- How many walls are there? How are they positioned?

Case Study 1: Plant Flowers

```
import becker.robots.*;

// Plant flowers around a square garden wall.
public class PlantFlowers
{
   public static void main(String[ ] args)
   {
      // Code to create the initial situation goes here.









      // Code to plant the flowers goes here.









   }
}
```

Case Study 1: Plant Flowers

```
import becker.robots.*;

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

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

      // Code to plant the flowers goes here.

   }
}
```

Quick Quiz

1.Name all the patterns used in this case study.

2.Which patterns are not used?

Quick Quiz Solutions

1.Patterns that are used:

- Java Program
- Object Instantiation
- Sequential Execution

2.Patterns that are not used:

- Command Invocation

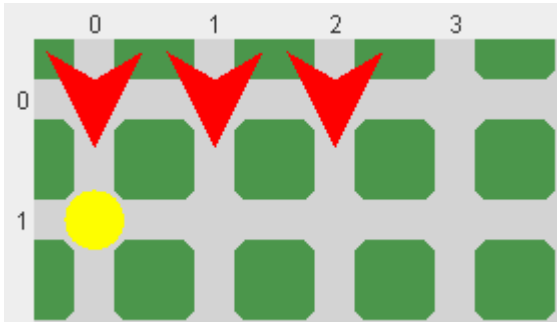
Case Study 1: Plant Flowers

```
...
    Robot karel = new Robot(berlin, 0, 1, Direction.SOUTH, 8);

    // Code to plant the flowers goes here.
    karel.move();
    karel.putThing();
    karel.move();
    karel.putThing();
    karel.turnLeft();

    karel.move();
    karel.putThing();
    karel.move();
    karel.putThing();
    karel.turnLeft();

    karel.move();
    karel.putThing();
    karel.move();
    karel.putThing();
    karel.turnLeft();

    karel.move();
    karel.putThing();
    karel.move();
    karel.putThing();
    karel.turnLeft();
  }
}
```

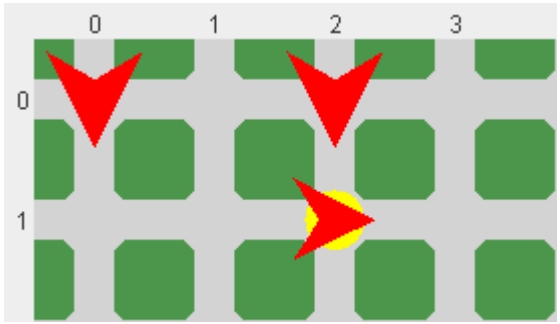
Note: The robot does the same steps four times, once for each side of the square. In the next lesson we'll learn how to exploit that fact.

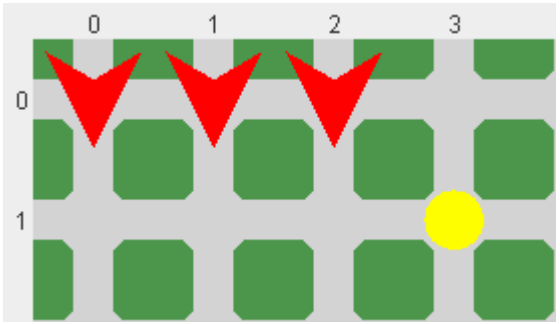
Case Study 2: An Assembly Line

Write a program in which three robots on an "assembly line" are positioned along street 0 at avenues 0, 1, and 2. A "part" (**Thing**) is positioned at (1, 0) on a "conveyor belt" along street 1. Starting with the westernmost robot, each robot processes the part in some way and then moves it into position for the next robot on the assembly line before returning to its own starting position.



Initial Situation





Final Situation

Questions:

- What path must each robot take to do its task?
- Does it matter which robot goes first?
- How can a robot turn around? Turn right?

```
import becker.robots.*;

// Simulate an assembly line with three robots and one part.
public class AssemblyLine
{

   public static void main(String[ ] args)
   {
```

Case Study 2: An Assembly Line

```
import becker.robots.*;

// Simulate an assembly line with three robots and one part.
public class AssemblyLine
{

   public static void main(String[ ] args)
   {  // Set up the initial situation
      City guelph = new City();
      Robot rayna = new Robot(guelph, 0, 0, Direction.SOUTH);
      Robot roopa = new Robot(guelph, 0, 1, Direction.SOUTH);
      Robot ruth = new Robot(guelph, 0, 2, Direction.SOUTH);

      Thing part = new Thing(guelph, 1, 0);
```

Case Study 2: An Assembly Line

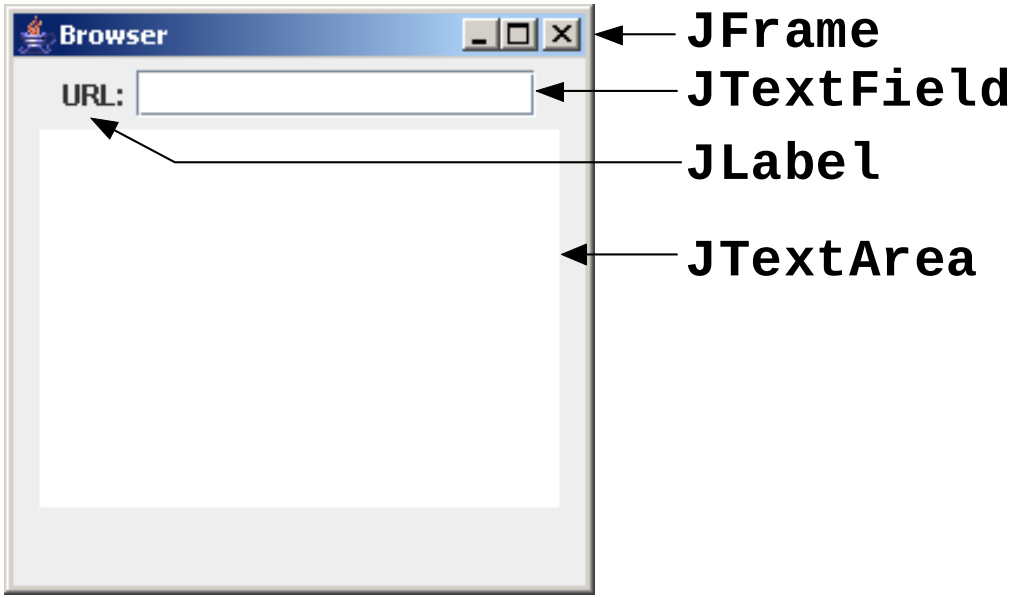
```
public static void main(String[ ] args)
{   // Set up the initial situation
    City guelph = new City();
    Robot rayna = new Robot(guelph, 0, 0, Direction.SOUTH);
    Robot roopa = new Robot(guelph, 0, 1, Direction.SOUTH);
    Robot ruth = new Robot(guelph, 0, 2, Direction.SOUTH);

    Thing part = new Thing(guelph, 1, 0);

    // The first robot moves the thing to the next stage.
    rayna.move();
    rayna.pickThing();
    rayna.turnLeft();
    rayna.move();
    rayna.putThing();
    rayna.turnLeft();
    rayna.turnLeft();
    rayna.move();
    rayna.turnLeft();
    rayna.turnLeft();
    rayna.turnLeft();
    rayna.move();
    rayna.turnLeft();
    rayna.turnLeft();
    // Repeat the above steps for each of the other robots.
```

Application: Using what we've learned

Apply the patterns learned with Robots to other situations

e.g.: To create the beginnings of a graphical user interface.

Use the **JFrame**, **JLabel**, **JTextField**, and **JTextArea** classes to write a program that looks (sort of) like a Web browser:



Use the following patterns:

- Java Program
- Object Instantiation
- Command Invocation

Application: Getting Ready to Program

Application: The Java Program Pattern

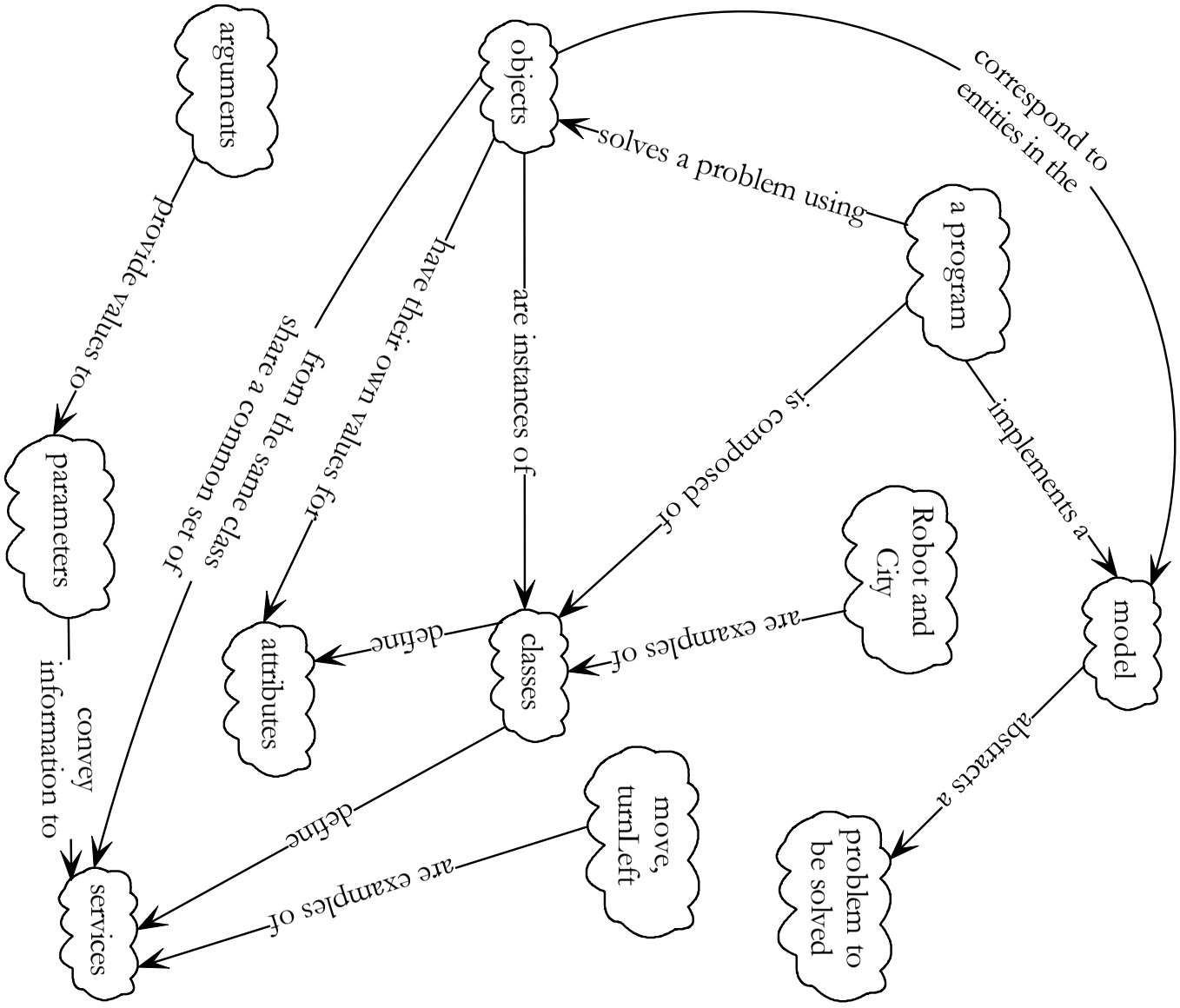
```
import javax.swing.*;

// Write a program that display a window which looks sort of like a Web browser.
public class Browser
{
   public static void main(String[ ] args)
   {
      // Construct appropriate objects

      // Use their services
   }
}
```

Application: The Object Instantiation Pattern

```
import javax.swing.*;

// Write a program that display a window which looks sort of like a Web browser.
public class Browser
{
   public static void main(String[ ] args)
   {
      // Construct appropriate objects
      JFrame frame = new JFrame();
      JPanel contents = new JPanel();
      JLabel label = new JLabel("URL:");
      JTextField url = new JTextField(15);
      JTextArea html = new JTextArea(10, 20);

      // Use their services
   }
}
```

```
...
  public static void main(String[ ] args)
  {
    // Construct appropriate objects
    JFrame frame = new JFrame();
    JPanel contents = new JPanel();
    JLabel label = new JLabel("URL:");
    JTextField url = new JTextField(15);
    JTextArea html = new JTextArea(10, 20);

    // Use their services
    contents.add(label);
    contents.add(url);
    contents.add(html);

    frame.setContentPane(contents);

    frame.setTitle("Browser");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLocation(250, 100);
    frame.setSize(250, 250);
    frame.setVisible(true);
  }
}
```

1.8: Concept Map



Summary

We have learned:

- how to create objects using an existing class
  (e.g.: **Robot karel = new Robot(myCity, 1, 2, Direction.EAST);**)
- how to use an object's services
  (e.g.: **karel.move();**)
- that these program statements must be contained within the Java Program pattern.
- that objects have attributes to store information.
- that objects are defined by a class.
- how to use documentation to find out more about a class.
- that several kinds of errors can affect a program.
- that many code patterns occur repeatedly in programs.