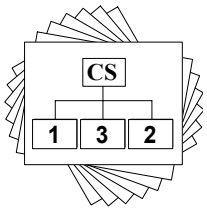
SIGCSE 2003
Workshop #18

Objects for Learning to Program with Java

Byron Weber Becker
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

bwbecker@uwaterloo.ca
http://www.cs.uwaterloo.ca/~bwbecker/robots/



Outline

1. Introductions *(20 min)*
2. Pedagogical Assumptions *(30 min)*
3. Introduce Karel *(45 min)*
4. Inheritance, Algorithms and Stepwise Refinement *(*
5. Break *(10 min)*
6. Introducing Variables *(20 min)*
7. Introducing Polymorphism *(20 min)*
8. Closing Discussion *(10 min)*

Who are we?

Name and Institution:	Byron Weber Becker University of Waterloo, Waterloo, Ontario
Experience teaching CS1 and OOP:	I've been teaching CS1 since 1991. After a very brief fling with Turing, taught for many years in Pascal. I was responsible for our conversion from Pascal to Java in Fall, 1998. After a false start, we began using Karel the Robot to teach OOP. We've been very excited by the results, and I'm looking forward to sharing it with you. I've been working on a textbook using this approach.
Class Sizes:	Usually 90 – 120 per class; about 1,000 per year.
Majors:	About 600 CS majors; 400 for math-related majors. We've recently begun using the same approach with non-majors.

Good Early O-O Example Programs

What makes a good early example in an object-oriented programming course?

- Should it be as simple as possible, like **HelloWorld**?
- Should it use objects, like **String** or **System.out**?
- Should it use other objects, like **Rectangle** or **Turtle**?
- Is it OK to use a provided library?
- What other characteristics are important?

In your teams, examine some of the following 10 examples (taken from real Java textbooks plus one CACM article purporting to improve on **HelloWorld**). List the characteristics, both positive and negative, that seem important to you in early example programs.

Good Early O-O Examples?

Example 1:

```
public class HelloWorld
{  public static void main(String[ ] args)
   {  System.out.println("Hello, world!");
   }
}
```

Example 2:

```
class Nothing
{  public static void main(String[ ] args)  {    }
}
```

Example 3:

```
import java.awt.Rectangle;
public class MoveRectangle
{  public static void main(String[ ] args)
   {  Rectangle cerealBox = new Rectangle(5, 10, 20, 30);
      cerealBox.translate(15, 25);
      System.out.println(cerealBox);
   }
}
```

Example 4:

```
public class PieceOfFabric extends SimpleGUI
{  private double sqMeters;
   public double toSqYards()
   {  double conversionFactor = 1.196;
      return conversionFactor * sqMeters;
   }
   public void readSqMeters()
   {  sqMeters = getDouble("Enter the fabric area in sq. meters:");
   }
   public void displayFabric()
   {  displayResult("The fabric size is " + sqMeters + " square meters or " +
            toSqYards() + "square yards.");
   }
}

public class ConvertFabric
{  public static void main(String[ ] args)
   {  PieceOfFabric aPiece = new PieceOfFabric();
      aPiece.readSqMeters();
      aPiece.displayFabric();
}
```

Example 5:

```
public class FirstProgram
{  public static void main(String[ ] args)
   {  System.out.println("Hello out there.");
      System.out.println("Want to talk some more?");
      System.out.println("Answer y for yes or n for no.");

      char answerLetter;
      answerLetter = MyLib.readLineNonwhiteChar();
      if (answerLetter = 'y')
         System.out.println("Nice weather we are having.");

      System.out.println("Good-bye.");

      System.out.println("Press enter key to end program.");
      String junk;
      junk = MyLib.readLine();
   }
}
```

Example 6:

```
public class Add16And23
{  public static void main(String[ ] args)
   {  int sum;
      sum = 16 + 23;
      System.out.println("The sum of 16 and 23 is " + sum);
   }
}
```

Example 7:

```
import turtleGraphics.*;
public class DrawSquare
{  public static void main(String[ ] args) throws TurtleException
   {  Turtle myTurtle = new Turtle();
      myTurtle.move(500);
      myTurtle.turnRight(90);
      myTurtle.move(500);
      myTurtle.turnRight(90);
      …
   }
}
```

Example 8:

```
import java.awt.*;
class Rings extends Frame
{  public static void main(String[ ] args)
   {  Frame f = new Rings();
      f.resize(300, 200);
      f.show();
   }
   public void paint(Graphics g)
   {  g.setColor(Color.red);
      g.drawOval(10, 30, 30, 30);
      …
   }
   public Rings()
   {  setTitle("Rings");
   }
   public boolean handleEvent(Event e)
   {  if (e.id == Event.WINDOW_DESTROY)
         System.exit(0);
      return super.handleEvent(e);
   }
```

Example 9:

```
import javabook.*;
class FunTime
{  public static void main(String[ ] args)
   {  SketchPad doodleBoard;
      doodleBoard = new SketchPad();
      doodleBoard.setVisible(true);
   }
}
```

Example 10:

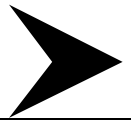
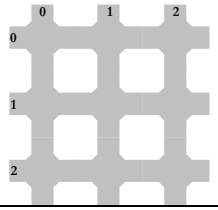
```
class HelloWorld
{  public static void printHello()
   {  System.out.println("Hello, World");
   }
}
class UseHello
{  public static void main(String[ ] args)
   {  HelloWorld myHello = new HelloWorld();
      myHello.printHello();
   }
}
```

Byron's Assumptions

Assumptions Regarding "Good" O-O Examples

1. Early examples ought to use objects, the central feature of the paradigm.
2. Objects should be explicitly instantiated (unlike **String**s and **System.out**). Understanding how objects are created is a vital part of learning to think with objects.
3. Objects should have their methods invoked, otherwise students view them simply as data containers or abstract pieces of syntax.
4. Each object should have easily discernable state and behavior. If not, the two core aspects of an object will remain a mystery to students.
5. Examples should contain two or more objects from the same class to drive home that each object has its own state but shares behavior with other members of its class.
6. Static methods, other than "main", should be avoided because they don't affect the state or behavior of individual objects, thus clouding two core concepts.
7. Students should use interesting objects before being asked to write their own classes.

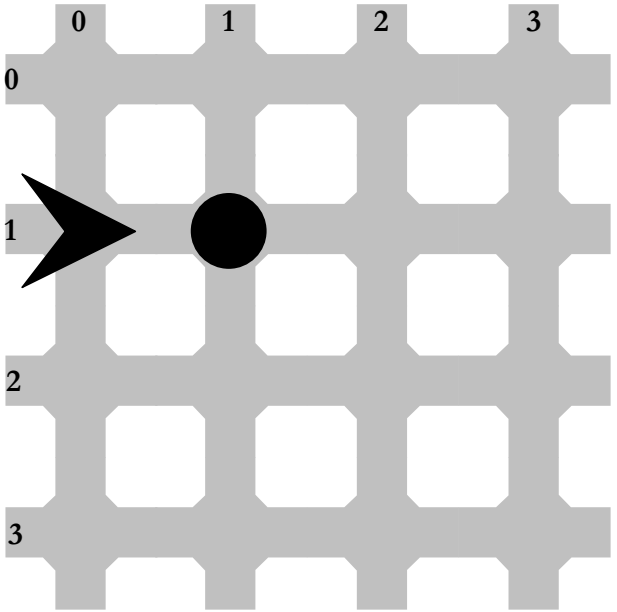
Key Elements in Karel the Robot

Robot		Can move, turn, pick things up and put things down.
City		Contains the streets and avenues where robots move. The intersections can hold other kinds of things.
Thing		A non-descript item in the city that can be picked up and moved by robots.
Flasher		A special kind of **Thing** that flashes, like a warning light used by maintenance workers. Can be turned on and off.
Streetlight		A special kind of **Thing** that illuminates an intersection. Streetlights can't be moved by robots. Can be turned on and off.
Wall		A special kind of **Thing** that blocks entry to and exit from an intersection.
CityFrame		A window in which these elements are displayed.

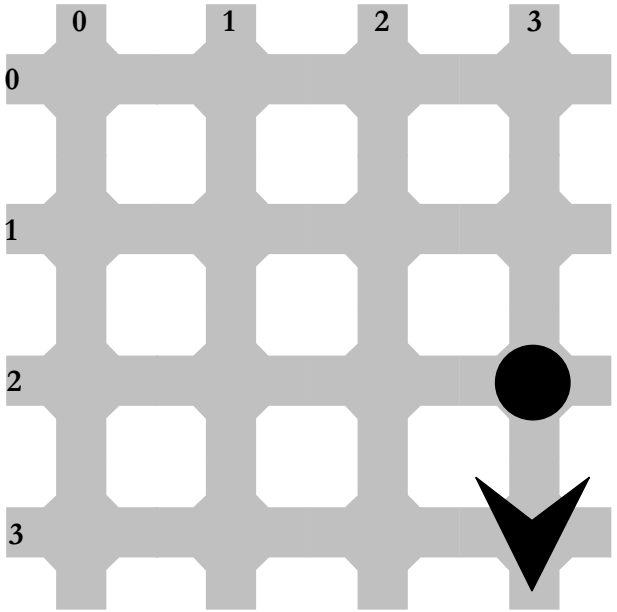
Problems can often be specified with the help of a diagram or two and just a few words of text. For example:

Move the **Thing** at (1,1) to (3,2).

Initial Situation

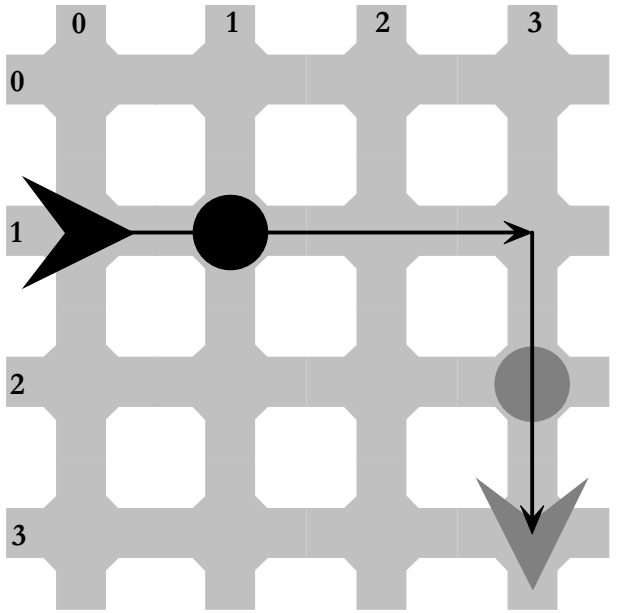


Final Situation

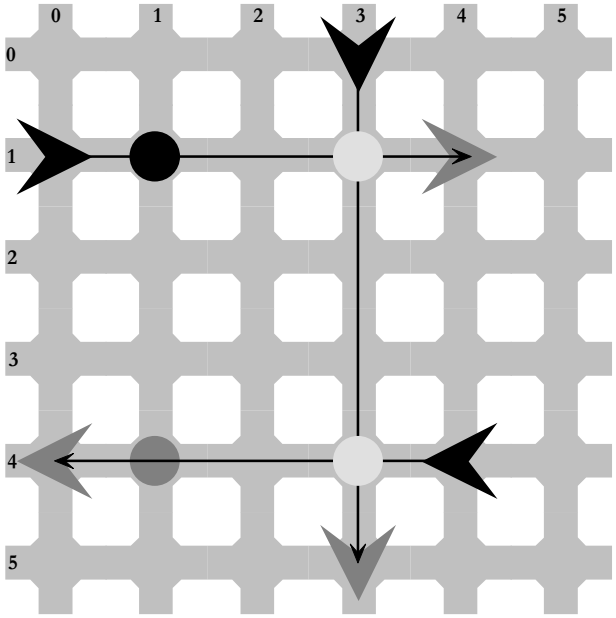


Ex01: Using one robot

```
import becker.robots.*;

/** Pick up and carry a Thing to another place. */
public class UseObj1 extends Object
{  public static void main(String[] args)
   {  City reno = new City();
      Robot karel = new Robot(reno, 0, 1, Directions.EAST, 0);
      Thing theThing = new Thing(reno, 2, 1);
      CityFrame frame = new CityFrame(reno, 4, 4);

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



```
import becker.robots.*;

public class UseObj2 extends Object
{  public static void main(String[] args)
   {  // set up the initial situation
      City reno = new City();
      Robot karel = new Robot(reno, 0, 1, Directions.EAST, 0);
      Robot sue = new Robot(reno, 3, 0, Directions.SOUTH, 0);
      Robot jim = new Robot(reno, 4, 4, Directions.WEST, 0);
      Thing theThing = new Thing(reno, 1, 1);
      CityFrame frame = new CityFrame(reno, 6, 6);

      // karel moves the Thing to sue
      karel.move();
      karel.pickThing();
      karel.move();
      karel.move();
      karel.putThing();
      karel.move();

      // sue moves the Thing to jim
      sue.move();
      sue.pickThing();
      sue.move();
```

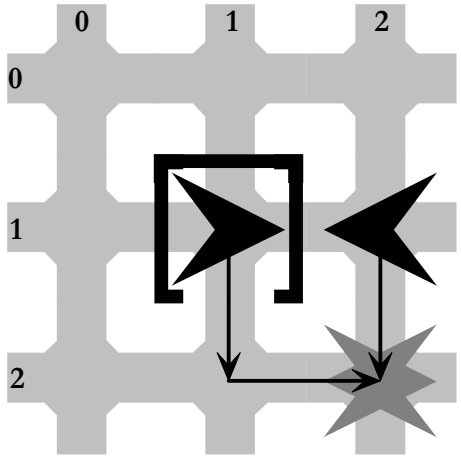


Ex03: Using two robots; three walls

```
import becker.robots.*;
public class UseObj3 extends Object
{  public static void main(String[] args)
   {  // Set up the initial situation
      City alcatraz = new City("..\\alcatraz.txt");
      Robot mark = new Robot(alcatraz, 1, 1, Directions.EAST, 0);
      Robot ann = new Robot(alcatraz, 2, 1, Directions.WEST, 0);
      CityFrame frame = new CityFrame(alcatraz, 6, 6);

      // mark escapes alcatraz
      mark.turnLeft();
      mark.turnLeft();
      mark.turnLeft();
      mark.move();
      mark.turnLeft();
      mark.move();

      // ann goes to meet mark
      ann.turnLeft();
      ann.move();
      ann.turnLeft();
      ann.turnLeft();
      ann.turnLeft();
   }
```

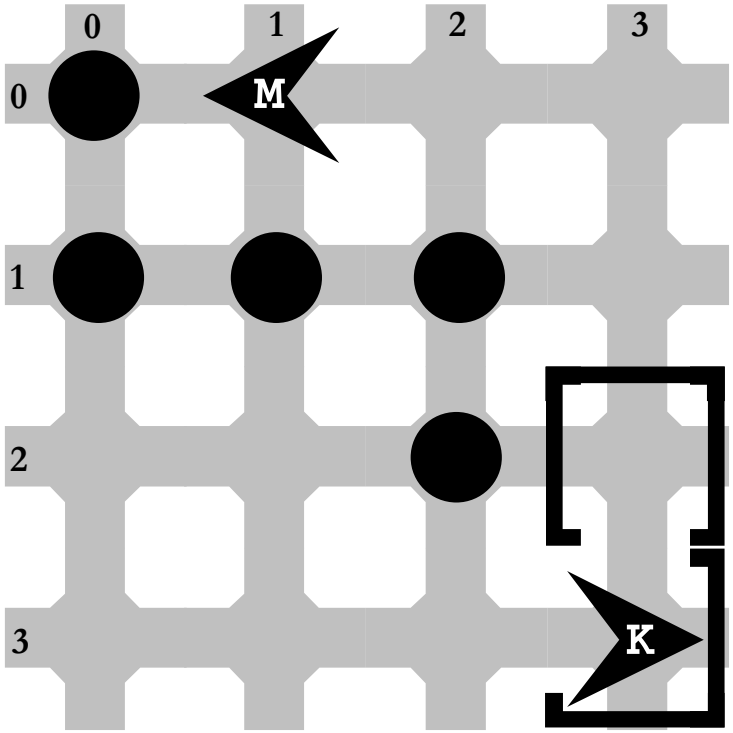
alcatraz.txt:

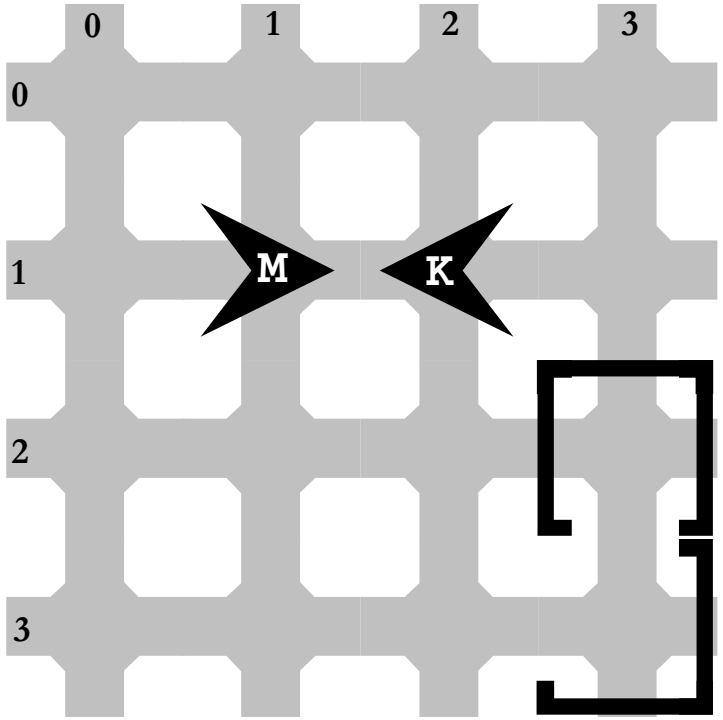
```
becker.robots.Wall 1 1 East
becker.robots.Wall 1 1 West
becker.robots.Wall 1 1 North
```



Programming Exercise

On the way home from the supermarket karel's bag ripped slightly at the bottom, spilling a few expensive items (things) on the ground. Fortunately, karel's neighbor maria noticed it and called just as karel got home. This initial situation is shown on the left. Write a program in which karel and maria both begin picking up the items, meeting as shown in the final situation on the right. The "M" and "K" are for illustrative purposes only; your program will not include them.





What Have We Learned?

- How to instantiate objects.
- Objects have state
  - Sometimes the state must be initialized with parameters.
  - State changes over time.
- Objects have services that can be invoked.
- Programs can use several objects that are the same "kind"; such objects are independent of each other.
- A typical program consists of quite a few different objects, all working together.
- Documentation is a useful resource.
- It is normal to write only a small portion of a program, relying on packages written by others for the majority of the program.

```
import javax.swing.*;                  // use JFrame, JPanel, JButton, JTextArea
import java.awt.*;                     // use Color

public class UseObj5 extends Object
{  public static void main(String[] args)
   {  JFrame frame = new JFrame();                  // declare the objects
      JPanel contents = new JPanel();
      JButton saveButton = new JButton("Save");
      JTextArea textDisplay = new JTextArea(5, 10);
      Color backgroundColor = new Color(255, 0, 0);

      frame.setBounds(300, 300, 150, 200);         // set up the frame
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      // set up the contents
      contents.setBackground(backgroundColor);
      contents.add(saveButton);
      contents.add(textDisplay);

      frame.setContentPane(contents);      // show the contents
      frame.setVisible(true);
   }
}
```

Advantages of using predefined, interesting classes

-
-
-
-
-

Disadvantages

-
-
-
-

Students often ask, “Can a robot turn right?”

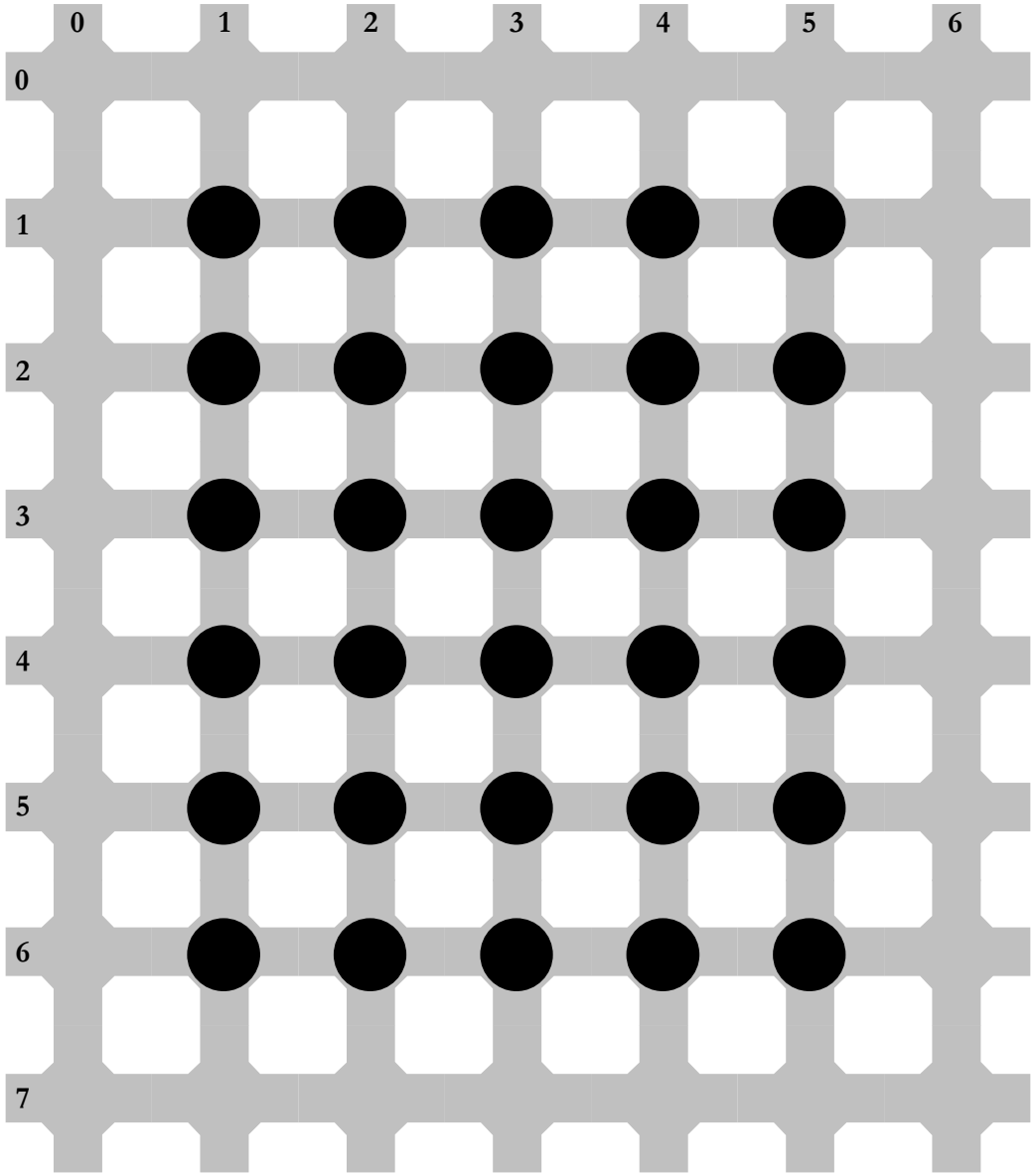
We reply, “No, but let me teach you how you can make a new kind of robot that turns right.”

```
import becker.robots.*;

public class NewKindOfRobot extends Robot
{
   public NewKindOfRobot(City c, int ave, int str, int dir, int nThings)
   {  super(c, ave, str, dir, nThings);
   }

   public void turnRight()
   {  this.turnLeft();
      this.turnLeft();
      this.turnLeft();
   }
}
```

```
public static void main(String[ ] args)
{   City reno = new City();
    NewKindOfRobot karel =
          new NewKindOfRobot(reno, 1, 1,
                             Directions.EAST, 0);
    CityFrame f = new CityFrame(reno, 5, 5);

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

A total of 30 flashers, arranged in 6 rows of 5, need to be picked up.

How could this task be accomplished? Brainstorm as many approaches as you can.



Harvester Stepwise Refinement (1/3)

```
/** A class of robot which can harvest a field of things.  The
    field must be 5 things wide and 6 rows high.
    @author Byron Weber Becker */
class Harvester extends RobotSE
{
    /** Construct a new Harvester robot.
        @param theCity the city where the robot will be created.
        @param ave robot's initial avenue
        @param str robot's initial street
        @param dir robot's initial direction, one of {Directions.NORTH, SOUTH, EAST, WEST}.
        @param numThings the number of things to place in the robot's backpack. */
    public Harvester(City theCity, int ave, int str, int dir, int numThings)
    {  super(theCity, ave, str, dir, numThings);
    }

    /** Harvest a field of things.  The robot is assumed to be on the
        north-west corner of the field. */
    public void harvestField()
    {  this.harvestTwoRows();
       this.positionForNextHarvest();
       this.harvestTwoRows();
       this.positionForNextHarvest();
       this.harvestTwoRows();
    }
```

Harvester Stepwise Refinement (2/3)

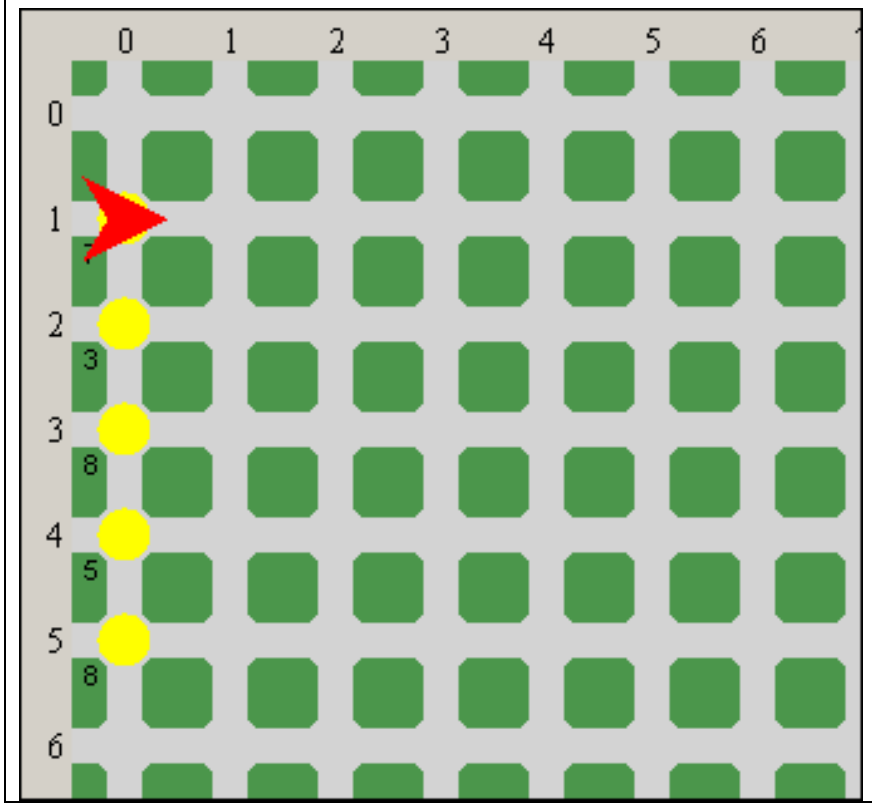
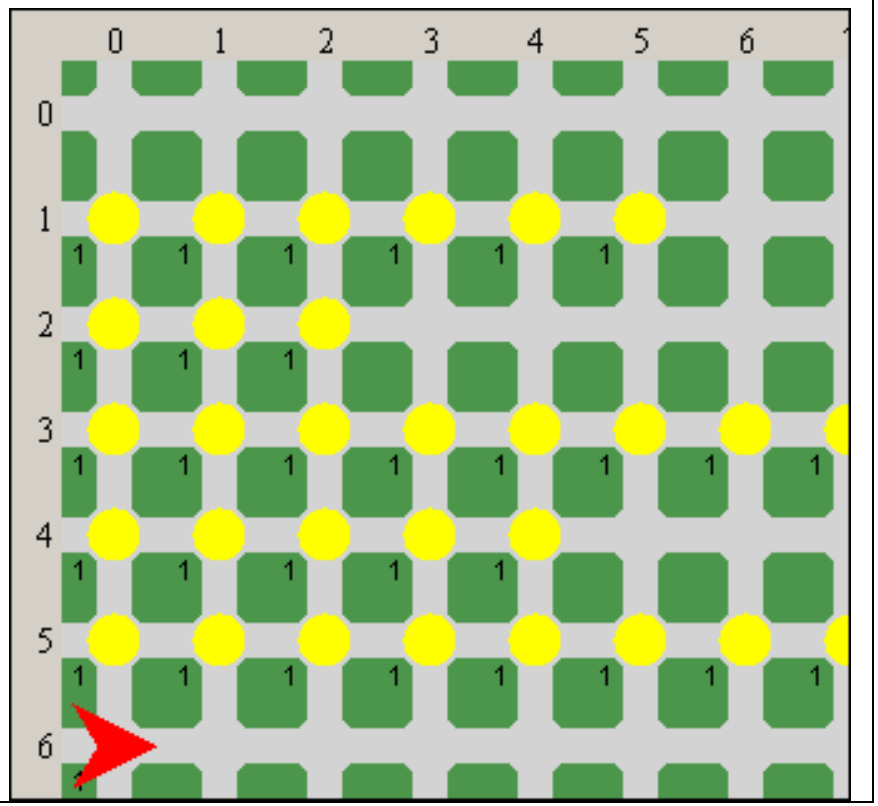
```
/** Harvest two rows of the field, returning to the same avenue
    but one street farther south. */
private void harvestTwoRows()
{  this.harvestOneRow();
   this.goToNextRow();
   this.harvestOneRow();
}

/** Harvest one row of five things. */
private void harvestOneRow()
{  this.harvestIntersection();
   this.move();
   this.harvestIntersection();
   this.move();
   this.harvestIntersection();
   this.move();
   this.harvestIntersection();
   this.move();
   this.harvestIntersection();
}
```

Harvester Stepwise Refinement (3/3)

```
   /** Go one row south and face west.  The robot must be facing east.  */
   private void goToNextRow()
   {  this.turnRight();
      this.move();
      this.turnRight();
   }

   /** Position the robot for the next harvest by moving one street
      south and facing west. */
   private void positionForNextHarvest()
   {  this.turnLeft();
      this.move();
      this.turnLeft();
   }

   /** Harvest the things on one intersection. */
   private void harvestIntersection()
   {  this.pickThing();
   }
}
```

Programming Exercise

Create a new kind of robot that will create a bar graph from data represented by piles of things on consecutive intersections on the same avenue as the robot.

Initial Situation	Final Situation
	

A file, **starter/BarGraph.java** will get you started.

The next slide summarizes the Robot queries available (useful for looping).

Available Queries

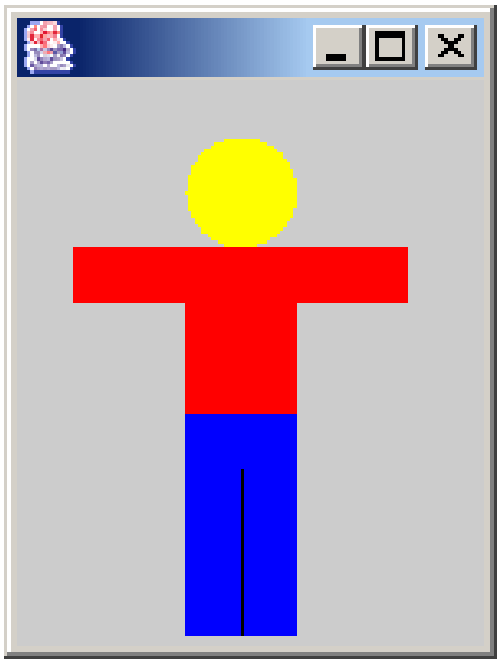
The following queries are available in the **Robot** class:

int getAvenue()	Which avenue is this robot on?
int getStreet()	Which street is this robot on?
int getDirection()	Which direction is this robot facing? One of **Directions.{NORTH, SOUTH, EAST, WEST}**.
int getSpeed()	How many milliseconds will this robot need for the next **move** or **turnLeft** instruction?
boolean frontIsClear()	Is it safe for the robot to move forward to the next intersection?
boolean isBesideThing()	Is this robot beside something which it can pick up?
boolean isBesideThing(Predicate kindOfThing)	Is this robot beside some specified kind of thing?
int countThingsInBackpack()	How many things are in this robot's back-pack?

What Have We Learned?

- New kinds of objects can be derived by extending an existing class.
  - Define the new in terms of the old (e.g.: **turnRight** is defined in terms of **turnLeft**)
  - New kinds of objects retain the capabilities of the old.
  - New kinds of objects can override selected capabilities of the old.
- Flow of control transfers to a method when it is invoked.
- There are often many different ways to solve a problem. Think carefully about the pros and cons of each one.
- A method may call another method defined within the same class.
  - Useful for decomposing a problem into more manageable pieces.
  - Can be declared **private** to avoid cluttering the public interface.
  - Can be declared **protected** to permit subclasses to override.
- A computer can appear to do several things at once.

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

class StickFigurePanel extends JPanel
{  public StickFigurePanel()    {  super();  }

   public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      Graphics2D g2 = this.setupGraphics(g);
      this.paintHead(g2);
      this.paintTorso(g2);
      this.paintLegs(g2);
   }

   private void paintHead(Graphics2D g2)
   {  g2.setColor(Color.yellow);
      g2.fillOval(3, 1, 2, 2);
   }

   private void paintTorso(Graphics2D g2)
   {  g2.setColor(Color.red);
      g2.fillRect(1, 3, 6, 1);
      g2.fillRect(3, 3, 2, 3);
   }
```



Objects encapsulate services (methods) and attributes. Services are shared among all objects belonging to a particular class. Each object has its own private attributes.

- Robot class:
  - All robots can move (a shared service).
  - Each robot has its particular location and direction (private attributes).
- Bank Account class:
  - All bank accounts have deposit, withdraw, and transfer methods (shared services).
  - Each bank account has its own owner and balance (private attributes).

Attributes are implemented with instance variables. Instance variables are simply variables that are global to the class. They can be used within any method.

When an object is constructed, a new copy of the instance variables are created specifically for that object.

A Simple Version of Robot (1/2)

```
import java.awt.*;
import java.awt.geom.*;
import becker.util.Utilities;

public class Robot extends Object implements Displayable
{  private int avenue;                  //robot's avenue
   private int street;                  //robot's street
   private double direction = 0.0; //robot's direction

   /** Create a new robot. */
   public Robot(City c, int anAvenue, int aStreet)
   {  super();
      this.avenue = anAvenue;
      this.street = aStreet;
      c.add(this, 2);
   }

   /** Move forward one intersection. */
   public void move()
   {  this.avenue = this.avenue + (int)(Math.cos(this.direction));
      this.street = this.street + (int)(Math.sin(this.direction));
      Utilities.sleep(400);   // sleep a little bit so the user can see that we moved
   }
```

A Simple Version of Robot (2/2)

```
    /** Turn left 90 degrees. */
    public void turnLeft()
    {  this.direction = this.direction - Math.PI/2.0;
       Utilities.sleep(400);
    }

    /** Paint this robot on the screen. */
    public void paint(Graphics2D g)
    {  g.setColor(Color.red);
       int centerX = this.avenue * Intersection.SIZE + 25;   //local variables
       int centerY = this.street * Intersection.SIZE + 25;

       //a robot shape, centred on (centerX,centerY) and facing the given direction
       RobotIcon icon =
            new RobotIcon(centerX, centerY, this.direction);
       g.fill(icon);
    }
}
```

Instance Variables vs. Temporary (local) Variables:

Issue	Instance	Temporary
Where?	Inside class, outside of methods.	Inside a method.
Scope?	Entire class	Point of declaration to the end of the smallest enclosing block.
Lifetime?	Lifetime of the object.	Lifetime of the method.
Declaration?	**private int street;** **private double dir = 0.0;**	**int currentStr;** **int interest = this.balance * .1;**
Visibility?	Prefer **private**; **public**, **protected** and package are also possible.	Does not apply.
Types?	integer types: **byte**, **short**, **int**, **long** floating point: **float**, **double** other: **char**, **boolean** reference: **String**, **Robot**, **City**, **Account**, …	

Exercises

- Add a **teleport** method to the **Robot** class. Test it with the following code:

```
City c = new City();
Robot r = new Robot(c, 1, 1);
CityFrame f = new CityFrame(c);   // robot should appear at (1, 1) facing East
r.teleport(4, 7);                 // robot should jump to (4, 7) facing East
```

- Add the ability to change the robot's colour by adding a **setColor** method. Test with

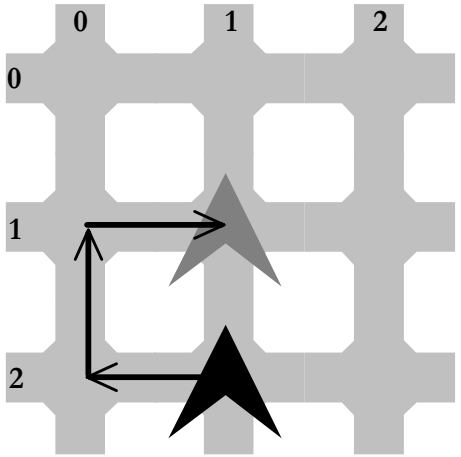
```
r.setColor(Color.green);
```

  - You will need to add a method with an argument of type **Color**.
  - You will need to add an instance variable.
  - You will need to use the instance variable in one of the existing methods.

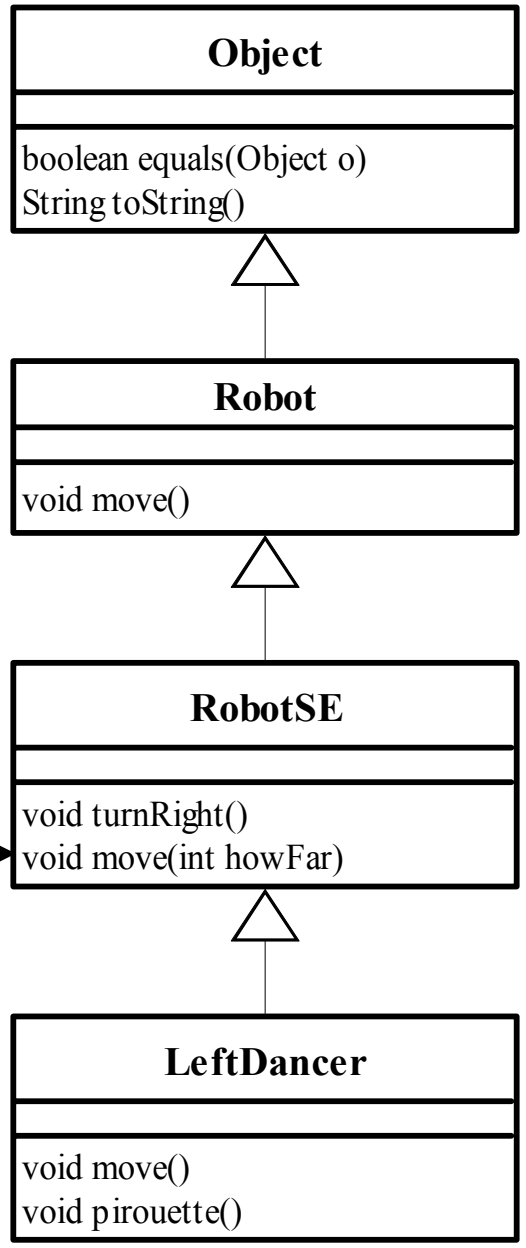
- With the current implementation, the robot moves from one intersection to another in a single jump. Animate the robot so that it takes more, but smaller steps, making its movement appears smoother. Hint: make the street and avenue of type **double**; make changes in the **move** and **paint** methods.
- Add two new methods, **goFaster()** and **goSlower()** which adjust the speed with which the robot travels.

```
import becker.util.Test;
public class Account extends Object
{  private String owner;

   public Account(String theOwner)
   {  super();
      this.owner = theOwner;
   }

   public static void main(String[ ] args)
   {  System.out.println("Testing Account class.");
      Account a = new Account("BW Becker");
      Test.ckEquals("owner set", "BW Becker", a.owner);
      /*
      Test.ckEquals("initial balance", 0.0, a.balance);
      a.deposit(500.00);
      Test.ckEquals("deposit 500", 500.00, a.balance);
      a.withdraw(250.00);
      Test.ckEquals("withdraw 250", 250.00, a.balance);
      Test.ckEquals("test getBalance()", 250.00, a.getBalance());
      Account b = new Account("AW Becker");
      a.transfer(b, 100.00);
      Test.ckEquals("after transfer out a", 150.00, a.getBalance());
      Test.ckEquals("after transfer out b", 100.00, b.getBalance()); */
   }
```

```
public class LeftDancer extends RobotSE
{
   public LeftDancer(…)
   {…}

   public void move()
   {  this.turnLeft();
      super.move();
      this.turnRight();
      super.move();
      this.turnRight();
      super.move();
      this.turnLeft();
   }

   public void pirouette()
   {  for(int i=0; i<4; i++)
      {  this.turnLeft();
      }
   }
}
```



What happens if **move(int howFar)** calls **this.move()**? What happens if it calls **super.move()**?



Example 1: No Polymorphism

```
public static void main(String[ ] args)
{  City danceFloor = new City();

   LeftDancer[ ] chorusLine = new LeftDancer[4];
   for(int i=0; i<chorusLine.length; i++)
   {  chorusLine[i] = new LeftDancer(danceFloor, 1+i, 4, Directions.NORTH, 0);
   }

   CityFrame f = new CityFrame(danceFloor, 7, 5);

   for(int i=0; i<3; i++)
   {  for(int j=0; j<chorusLine.length; j++)
         chorusLine[j].move();
   }
   for(int i=0; i<chorusLine.length; i++)
   {  chorusLine[i].pirouette();
   }
}
```

What happens if…?

```
public static void main(String[ ] args)
{  City danceFloor = new City();

   Robot[ ] chorusLine = new Robot[4];
   for(int i=0; i<chorusLine.length; i++)
   {  chorusLine[i] = new LeftDancer(danceFloor, 1+i, 4, Directions.NORTH, 0);
   }
   CityFrame f = new CityFrame(danceFloor, 7, 5);

   for(int i=0; i<3; i++)
   {  for(int j=0; j<chorusLine.length; j++)
         chorusLine[j].move();
   }

   for(int i=0; i<chorusLine.length; i++)
   {  chorusLine[i].pirouette();
   }
}
```

Example 3: A Collection of Dancers

```
public static void main(String[ ] args)
{  City danceFloor = new City();

   Robot[ ] chorusLine = new Robot[4];
   for(int i=0; i<chorusLine.length; i++)
   {  if (i % 3 == 0)
      {  chorusLine[i] = new LeftDancer(danceFloor, 1+i, 4, Directions.NORTH, 0);
      } else if (i % 3 == 1)
      {  chorusLine[i] = new RightDancer(danceFloor, 1+i, 4, Directions.NORTH, 0);
      } else
      {  chorusLine[i] = new Robot(danceFloor, 1+i, 4, Directions.NORTH, 0);
      }
   }
   CityFrame f = new CityFrame(danceFloor, 7, 5);

   for(int i=0; i<4; i++)
   {  for(int j=0; j<chorusLine.length; j++)
         chorusLine[j].move();
   }
}
```

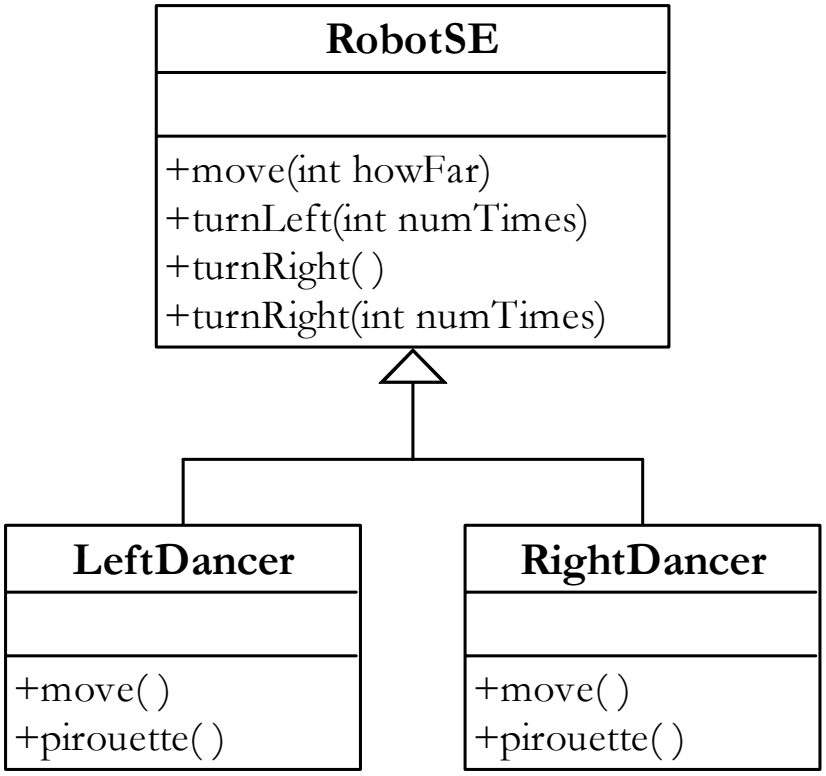
Example 4: InstanceOf (1/2)

```
import becker.robots.*;
import becker.io.*;

public class Example4 extends Object
{
   public static void main(String args[ ])
   {  City danceFloor = new City();
      TextInput in = new TextInput();

      Robot[ ] chorusLine = new Robot[4];
      for(int i=0; i<chorusLine.length; i++)
      {  System.out.print("Enter 'L' for left, 'R' for right, anything for Robot: ");
         char kind = in.readLine().charAt(0);
         if (kind == 'L')
            chorusLine[i] = new LeftDancer(danceFloor, 1+i, 4, Directions.NORTH, 0);
         else if (kind == 'R')
            chorusLine[i] = new RightDancer(danceFloor, 1+i, 4, Directions.NORTH, 0);
         else
            chorusLine[i] = new Robot(danceFloor, 1+i, 4, Directions.NORTH, 0);
      }
      in.close();
      CityFrame f = new CityFrame(danceFloor, 7, 5);
```

Example 4: InstanceOf (2/2)

```
    for(int i=0; i<3; i++)
    {  for(int j=0; j<chorusLine.length; j++)
          chorusLine[j].move();
    }

    // Pirouette, if it can.
    for(int i=0; i<chorusLine.length; i++)
    {  if (chorusLine[i] instanceof LeftDancer)
       {  LeftDancer ld = (LeftDancer)chorusLine[i];
          ld.pirouette();
       } else if (chorusLine[i] instanceof RightDancer)
       {  RightDancer rd = (RightDancer)chorusLine[i];
          rd.pirouette();
       }
    }
}
```

Use polymorphism when you have two or more different kinds of a thing and each kind has a different way to respond to the same message.



Applications

- A company may have different kinds of employees, each with a different method of calculating pay.
- A game such as Othello may have different strategies for choosing a move.
- A drawing program has different classes for each shape it can draw (circles, rectangles, lines), each with a draw method to display that shape.
- An operating system may connect to many different kinds of printers, each with its own way of implementing the "print" command.
- A web browser must be able to display images of several different types (JPEG, GIF, etc).

To show polymorphism, we need some plausible excuse to do something like:

```
Robot karel = new LeftDancer(…);
```

Options:

- Use a collection of different kinds of objects (see above).
- Ask the user at run-time:

```
char choice = in.getChar("Right or Left Dancer? [enter r or l]");
Robot aDancer = null;
if (choice == 'r')          {  aDancer = new RightDancer(…);}
else if (choice == 'l')     {  aDancer = new LeftDancer(…);   }
else                        {  /* handle error */             }
aDancer.move();
```

- Receive an unknown kind of robot as a parameter:

```
chorusLine.addDancer(new LeftDancer());
…
public void addDancer(Robot aDancer)
{  aDancer.move(); }
```

- Receive an unknown kind of robot as a return value:

```
Robot aDancer = chorusLine.getDancer(i);
aDancer.move();
```

Robots

- **examineThing()** – returns a reference to a **Thing** on the current intersection. Could be a **Thing**, **Light**, **Streetlight**, **Flasher**, **Wall**, etc.
- {**pick**, **put**, **examine**, **beside**}**Thing(Predicate pred)** take subclasses of **Predicate** to determine which subclass of **Thing** should be manipulated.
- **intersection()** returns a reference to the currently occupied intersection. Could be a subclass such as **gasStationIntersection** or **wormHoleIntersection**.

Intersections

- **addThing(Thing t)** – called when something is added to an intersection. Could be a **Thing**, **Flasher**, **Wall**, **Robot**, etc. **removeThing(…)** is similar.

Sims (superclass of Robot, Thing, Intersection)

- **setIcon(Icon i)** – pass it subclasses of **Icon** to affect how the **Sim** appears.

City

- **makeIntersection(int ave, int str)** is used to make (and return) custom intersections.
- **customizeIntersection(Intersection i)** could receive any kind of intersection.

More stuff that we could talk about…

- Revisit any of the preceding topics for clarification, new ideas, suggestions, …
- Overall impressions of the approach
- The textbook I'm writing to support the approach
- How we deliver CS1 to 1,000 students per year
  - Support staff
  - Using undergraduate tutors
  - Class – Practicum – Lab – Assignment format
  -
-