BEIT Conference Slides:
Java Instance Variables & Polymorphism

Preconditions

- Students are familiar with classes and objects via the robot world.
- Students can extend and existing class.

Postconditions

- Students will understand how to use instance variables in a class.
- Students will understand the basic principles of polymorphism.

Outline

- Brief Review
- Instance Variables
  - Examples
  - Compared to Local Variables
  - Exercises
  - Testing; Non-Robot Class
- Arrays
- Polymorphism

Resources

www.cs.uwaterloo.ca/~bwbecker/papers/BEIT2003

www.cs.uwaterloo.ca/~bwbecker/robots

Robots: Learning to Program with Robots

Review

```
import becker.robots.*;
public class Main
{  public static void main(String[ ] args)
   {  City c = new City();
      MyBot mb = new MyBot(c, 1, 1, Directions.EAST, 0);
      CityFrame f = new CityFrame(c);

      mb.move(5);
   }
}
```

```
import becker.robots.*;
public class MyBot extends Robot
{  public MyBot(City c, int ave, int str, int dir, int numThings)
   {  super(c, ave, str, dir, numThings);
   }

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

Instance Variables

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 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 is created specifically for that object. Changing the values for one object (i.e. Robot) does *not* affect any other object.

A Simplified Robot Class (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; //direction – measured in radians; facing EAST.

   /** Create a new robot. */
   public Robot(City c, int anAvenue, int aStreet)
   {  super();                          // must be first statement in the constructor
      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 Simplified Robot Class (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);
}
}
```

Variables

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 block.
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, Directions.EAST, 0);
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** method.
- Add two new methods, **goFaster()** and **goSlower()** which adjust the speed with which the robot travels.

Testing and Non-Robot Classes

```
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()); */
   }
```

Three Steps for Initializing Arrays:

1. Declare the name	**Robot[ ] chorusLine;** **int[ ] daysInMonth;**	ChorusLine:
2. Allocate the space	**chorusLine = new Robot[5];** **daysInMonth = new int[12];**	ChorusLine: 0 1 2 3 4
3. Initialize values	**chorusLine[0] = new Robot (c, 1, 2, Directions.EAST,0);** **daysInMonth[0] = 31;**	ChorusLine: 0 1 2 3 4 Robot currrentAvenue: 1 currentStreet: 2 direction: EAST backpack:

Examples of Accessing Arrays:

```
// chorusLine.length gives number of elements
for (int i=0; i<chorusLine.length; i++)
{  Robot r = chorusLine[i];
   r.move();
}
```

```
double avg = 0.0;
for(int i=0; i<days.length; i++)
   avg = avg + days[i];
System.out.println(
        avg/days.length);
```

Dancers: Example 0

```
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 Robot(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();
   }
}
```

LeftDancer and RightDancer

```java
public class LeftDancer extends RobotSE
{
  public LeftDancer(
     City c, int a, int s, int d, int n)
  {  super(c, a, s, d, n);
  }

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

  public void pirouette()
  {  this.turnLeft(4);
  }
}
```

```java
public class RightDancer extends RobotSE
{
  public RightDancer(
     City c, int a, int s, int d, int n)
  {  super(c, a, s, d, n);
  }

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

  public void 
  {  this.turnRi
  }
}
```

RobotSE:

- move(int howFar)
- turnRight really turns right.
- includes turnLeft(int n) and turnRight(int n)

Dancers: Example 1

```
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();
   }
}
```

Dancers: Example 2

```
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 RightDancer(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();
   }
}
```

Dancers: Example 3

```
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 if (i % 3 == 2)
      {  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();
   }
}
```

Dancers: Example 4 (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);
```

Dancers: Example4 (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();
       }
    }
}
```

Polymorphism:

- Different kinds of objects that have the same basic capability, but implemented slightly differently (i.e.: a plain move vs. a dancing move)
- The same method name is used.
- The client (code using the object) doesn't really care how it does something (a plain move vs. a dancing move).
- Core requirement: all the different varieties extend the same superclass, overriding one or more methods.
- Places where polymorphism makes sense:
  - 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, each implementing **chooseNextMove** differently.
  - A drawing program has different classes for each shape (lines, rectangles, circles, …), each with a **draw** method to display that shape.
  - A web browser can display different kinds of images (jpeg, gif, etc). Each kind implements **drawImage** differently.

Summary

Understanding Object-Oriented Java involves:

- knowing how to use objects instantiated from existing classes.
- knowing how to create a customized class by extending an existing class.
- knowing how to implement instance variables.
- understanding method resolution when methods are overridden and overloaded.
- knowing how to use polymorphism.

In addition, the "standard stuff" is required:

- selection and repetition; Boolean expressions
- parameter passing
- arrays
- I/O

Teachers

- must appreciate the difference between objects and references to objects.