Java:

Learning to Program with Robots

Chapter 07: More on Variables and Methods

Chapter Objectives

After studying this chapter, you should be able to:

- Write queries to reflect the state of an object
- Use queries to write a test harness that tests your class
- Write classes that use types other than integer, including floating-point types, **booleans**, characters, and strings
- Write and use an enumerated type
- Write a class modeling a simple problem
- Describe the difference between a class variable and an instance variable
- Write classes that implement an interface and can be used with provided graphical user interfaces from the **becker** library

7.1.1: Testing a Command

Testing a class involves conducting many individual tests.

Collectively, these tests are called a *test suite*.

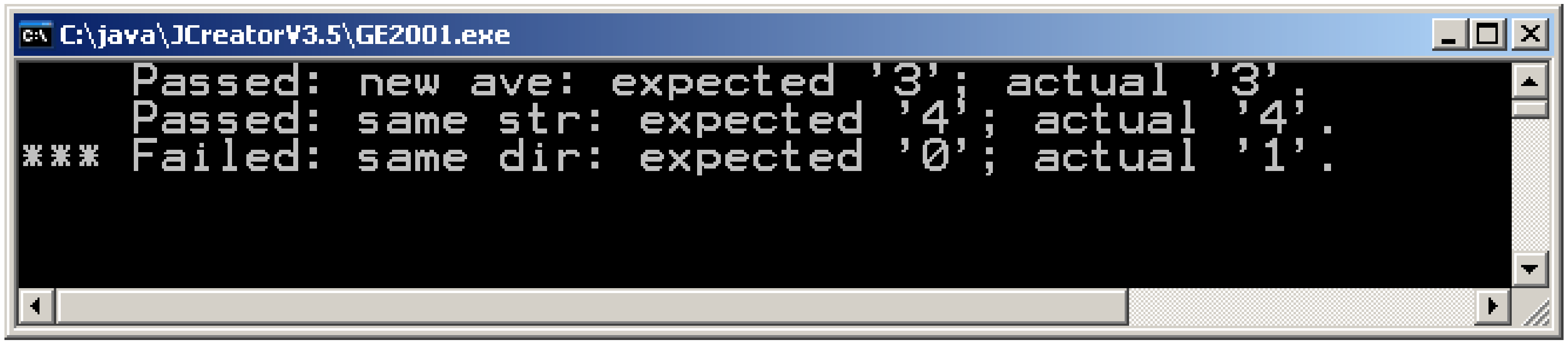
Each test involves five steps:

1. Decide which method you want to test.
2. Set up a known situation.
3. Determine the expected result of executing the method.
4. Execute the method.
5. Verify the results.

Ideally, we would like to test the class after each change. This implies automating the testing.

7.1.1: Testing a Command

```
import becker.util.Test;

public class TestHarness
{
   public static void main(String[ ] args)
   {  // Step 1:  Test move method
      // Step 2:  Put robot in an empty city on (4,2) facing East.
      SimpleCity c = new SimpleCity();
      SimpleBot karel = new SimpleBot(c, 4, 2, Constants.EAST);

      // Step 3:  The robot should end up on (4,3) facing East.
      // Step 4:  Execute the method.
      karel.move();

      // Step 5:  Verify the result.
      Test tester = new Test();
      tester.ckEquals("new ave", 3, karel.getAvenue());
      tester.ckEquals("same str", 4, karel.getStreet());
      tester.ckEquals("same dir", Constants.EAST, karel.getDirection());
   }
}
```


C:\java\JCreatorV3.5\GE2001.exe

```
    Passed: new ave: expected '3'; actual '3'.
    Passed: same str: expected '4'; actual '4'.
*** Failed: same dir: expected '0'; actual '1'.
```

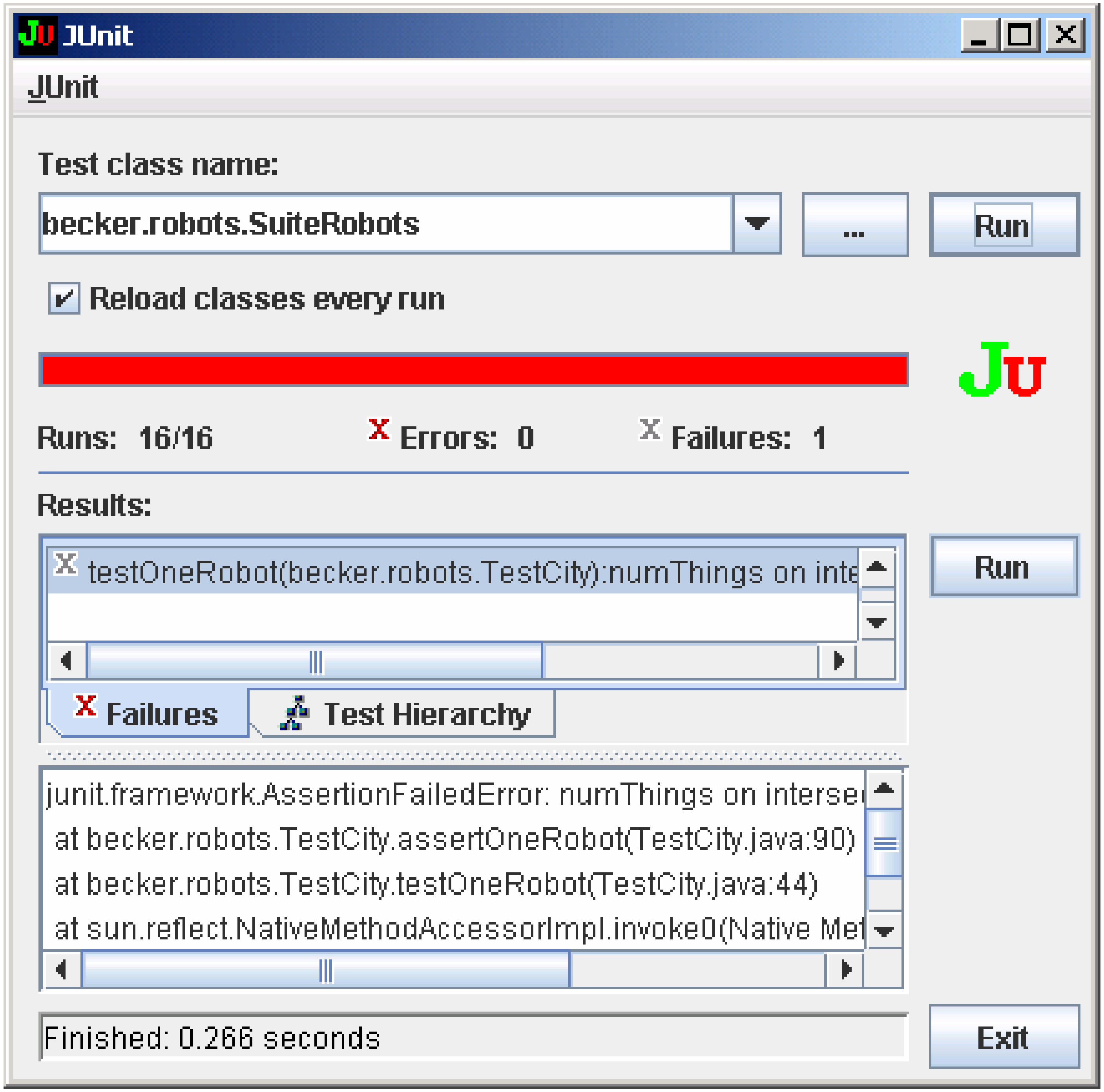
How does **ckEquals** work?

```
public class Test
{
   public void ckEquals(String msg, int expected, int actual)
   {
      if (expected == actual)
      { print passed message
      } else
      { print failed message
      }
   }
}
```

7.1.3: Using Multiple Main Methods

Each class may have its own **main** method. This allows us to put a test harness into every class! Write one more **main** method in its own class (as we have been doing) to run the program.

```
public class SimpleBot extends Paintable
{  private int street;
   private int avenue;
   …
   public SimpleBot(…)   {  …   }
   public void move()      {  …   }

   public static void main(String[ ] args)
   {  SimpleCity c = new SimpleCity();
      SimpleBot karel = new SimpleBot(c, 4, 2, Constants.EAST);

      karel.move();

      Test tester = new Test();
      tester.ckEquals("new ave", 3, karel.getAvenue());
      tester.ckEquals("same str", 4, karel.street);
      …
   }
}
```



```
becker.robots.SuiteRobots
```

```
Runs:   16/16          Errors:  0          Failures:  1
```

```
testOneRobot(becker.robots.TestCity):numThings on inte
```

```
junit.framework.AssertionFailedError: numThings on interse
 at becker.robots.TestCity.assertOneRobot(TestCity.java:90)
 at becker.robots.TestCity.testOneRobot(TestCity.java:44)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Met
```

```
Finished: 0.266 seconds
```

What is a *type*?

- It is used when declaring a {instance, temporary, parameter} variable. For example:

```
private int street;                    // instance variable
Robot karel = new Robot(…);            // temporary variable
```

- The type specifies the *values* the variable may take on.
  - **street** may be assigned integers like -10, 0, and 49 (and only integers).
  - **karel** may be assigned **Robot** objects.

- The type specifies the *operations* that may be performed.
  - **street** may be used with **+**, **-**, *****, **/**, **=**, **==**, etc.
  - **karel** may be used with the method invocation operator, **.** (dot): **karel.move()**, etc.

Java has six numeric types

Integer Types

Type	Smallest Value	Largest Value	Preci-sion
byte	-128	127	exact
short	-32,768	32,767	exact
int	-2,147,483,648	2,147,483,647	exact
long	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	exact

Floating-Point Types

Type	Smallest Magnitude	Largest Magnitude	Precision
float	$\pm 1.40239846 \times 10^{-45}$	$\pm 3.40282347 \times 10^{38}$	about 7 digits
double	$\pm 4.940656458412 \times 10^{-324}$	$\pm 1.7976931348623 \times 10^{308}$	about 16 digits

Operations available on numeric types:

`* / %`	multiplication, division, remainder
`+ -`	addition, subtraction
`< <= > >= != ==`	comparison
`=`	assignment

The type of *, /, %, +, and – is the same as the largest of the operands. For example:

```
int a = 2;
double b = 1.5;
```

The result of **a + b** is 3.5 because **b**, a **double**, stores larger numbers than **a**, an **int**.

7.2.3: Converting Between Numeric Types

int i = 159;
double d = i;
	The integer **159** is implicitly converted to **159.0** before assignment to **d**.

double d = 3.999;
int i = d;
	This results in a compile-time error. Java doesn’t know what to do with the **.999**, which can’t be stored in an **int**.

double d = 3.999;
int i = (int)d;
or
int j = (int)(d * d / 2.5);
	Java converts the double value to an integer by dropping the decimal part (not rounding). That is, **i** becomes **3**.

7.2.4: Formatting Numbers

```
double carPrice = 12225.00;
double taxRate = 0.15;

System.out.println("Car: " + carPrice);
System.out.println("Tax: " + carPrice * taxRate);
System.out.println("Total: " + carPrice * (1.0 + taxRate));
```

This code gives the following output:

```
Car: 12225.0
Tax: 1833.75
Total: 14058.749999999998
```

We would like to format it using a familiar currency symbol, such as **$**, and two decimal places.

```
import java.text.NumberFormat;
…
NumberFormat money = NumberFormat.getCurrencyInstance();
…
System.out.println(Total: " + money.format(carPrice * (1.0 + taxRate)));
```

7.2.5: Taking Advantage of Shortcuts

Rather than writing

```
public void move()
{  this.street = this.street + this.strOffset();
   this.avenue = this.avenue + this.aveOffset();
   Utilities.sleep(400);
}
```

one may write

```
public void move()
{  this.street += this.strOffset();
   this.avenue += this.aveOffset();
   Utilities.sleep(400);
}
```

In general,

```
«var» += «expression»;
```

is evaluated as

```
«var» = «var» + («expression»);
```

Similarly for **-=**, ***=**, **/=**, and **%=**.

7.3.1: The Boolean Type

A **boolean** variable stores either **true** or **false**.

```
public class SimpleBot extends Paintable
{  private int street;
   private int avenue;
   private boolean isBroken = false;
   …

   public void breakRobot()
   {  this.isBroken = false;
   }

   public void move()
   {  // Ignore a command to move if the robot is broken.
      if (!this.isBroken)
      {  this.street += this.strOffset();
         this.avenue += this.aveOffset();
         Utilities.sleep(400);
      }
   }
   …
}
```

Instance variables, temporary variables, parameter variables, and constants can all be of type **boolean**.

7.3.2: The Character Type

A single character such as **a**, **Z**, **?**, or **5** can be stored in a variable of type **char**.

Characters are the symbols you can type at the keyboard – plus many that you can't type directly.

```
public class KeyBot extends RobotSE
{ …

  /** Override a method in Robot that does nothing with the specifics of what a KeyBot
  *   should do when a specific key is typed on the keyboard. */
  protected void keyTyped(char key)
  { if (key == 'm' || key == 'M')
    {  this.move();
    } else if (key == 'r' || key == 'R')
    {  this.turnRight();
    } else if (key == 'l' || key == 'L')
    {  this.turnLeft();
    }
  }
}
```

Some special characters are written with *escape sequences*:

Sequence	**Meaning**
\'	Single quote
\"	Double quote
\	Backslash
\n	Newline – used to start a new line of text when printing at the console.
\t	Tab – inserts space so the next character is placed at the next tab stop.
\b	Backspace – moves the cursor backwards over the previously printed character.
\r	Return – moves the cursor to the beginning of the current line.
\f	Form feed – moves the cursor to the top of the next page of a printer.
\u*dddd*	A Unicode character, each *d* being a hexadecimal digit.

7.3.3: Using Strings

A *string* is a sequence of characters. They are used frequently in Java programs.

```
public class StringExample
{
public static void main(String[ ] args)
{  String msg = "Don't drink and drive.";

   System.out.println("Good advice: " + msg);
}
}
```

String is a class (like **Robot**), but has special support in Java:

- Java will automatically construct a **String** object for a sequence of characters between double quotes.
- Java will “add” two strings together with the plus operator to create a new string. This is called *concatenation*.
- Java will automatically convert primitive values and objects to strings before concatenating them with a string.

7.3.3: Special Support for Strings

```
import becker.robots.*;
public class Main
{
   public static void main(String[ ] args)
   {  String greeting = "Hello";
      String name = "karel";

      System.out.println(greeting + ", " + name + "!");

      System.out.println("Did you know that 2*PI = " + 2*Math.PI + "?");

      City c = new City();
      Robot karel = new Robot(c, 1, 2, Direction.SOUTH);
      System.out.println("c=" + c);
   }
}
```

```
Hello, karel!
Did you know that 2*PI = 6.283185307179586?
c=becker.robots.City[SimBag[robots=[becker.robots.Robot[street=1, av
enue=2, direction=SOUTH, isBroken=false,numThingsInBackpack=0]], t
hings=[ ]]]
```

A **String** object is created automatically.

Four concatenated strings.

Primitive automatically converted to string.

Object automatically converted to string.

7.3.3: Overriding toString

To convert an object to a **String**, Java calls the object's **toString** method. Classes should override **toString** to return a meaningful value.

```
public class SimpleBot extends Paintable
{  private int street;
   private int avenue;
   private int direction;
   …

   /** Represent a SimpleBot as a string. */
   public String toString()
   {  return "SimpleBot" +
         "[street=" + this.street +
         ", avenue=" + this.avenue +
         ", direction=" + this.direction +
         "]";
   }
}
```

7.3.3: Querying a String (1/2)

Method	Description
int length()	How many characters are in this string?
char charAt(int index)	Which character is at the given index (position)? Indices start at 0.
int compareTo(String aString)	Return 0 if this string is equal to **aString**; -1 if this string
boolean equals(Object anObj)	
boolean startsWith(String prefix)	

7.3.3: Querying a String (2/2)

```
int indexOf(
   char ch)
int indexOf(char ch,
   int fromIndex)
int indexOf(String subString)
int lastIndexOf(char ch)
```

7.3.3: Querying a String

```java
public class StringQueryDemo
{
   public static void main(String[ ] args)
   {  String s1 = "A string to play with.";
      String s2 = "Another string.";

      int s1Len = s1.length();
      System.out.println("'" + s1 + "' is " + s1Len + " characters long.");

      if (s1.compareTo(s2) < 0)
      {  System.out.println("'" + s1 + "' appears before '" + s2 + "' in the dictionary.");
      } else if (s1.compareTo(s2) > 0)
      {  System.out.println("'" + s1 + "' appears after '" + s2 + "' in the dictionary.");
      } else
      {  System.out.println("The two strings are equal.");
      }

      int pos = s1.indexOf("play");
      System.out.println("'play' appears at position " + pos);

      System.out.println("The character at index 3 of '" + s2 + "' is " + s2.charAt(3));
   }
}
```

7.3.3: Transforming a String

```
public class StringTransformDemo
{
   public static void main(String[ ] args)
   {
   String w = "warning: ";

   System.out.println(w.toUpperCase() + "Core breach imminent!");
   System.out.println(w.trim().toUpperCase() + "Core breach imminent!");

   System.out.println(w.substring(1, 4));
   }
}
```

Output:

```
WARNING:  Core breach imminent!
WARNING:Core breach imminent!
arn
```

Case Study 1: Palindromes

A palindrome is a phrase that is the same backwards and forwards – after all the letters have been made the same case and non-letter characters have been removed.

Here's are some samples:

civic

Madam, I'm Adam.

Was it a cat I saw?

A man, a plan, a canal. Panama!

Write a class named **Palindrome** which contains a method named **isPalindrome**. This method takes a string as a parameter and returns **true** if it is a palindrome and **false** if it is not.

Case Study 1: Palindrome Test Suite

```
import becker.util.Test;

public class Palindrome
{
   public Palindrome()        {  super();   }

   public boolean isPalindrome(String p)
   {  return true;
   }

   public static void main(String[ ] args)
   {  Palindrome pal = new Palindrome();

      Test tester = new Test();
      tester.ckEquals("a", true, pal.isPalindrome("a"));
      tester.ckEquals("aba", true, pal.isPalindrome("aba"));
      tester.ckEquals("aBbA", true, pal.isPalindrome("aBbA"));
      tester.ckEquals("Madam, I'm Adam", true,
                                    pal.isPalindrome("Madam, I'm Adam"));
      tester.ckEquals("ac", false, pal.isPalindrome("ac"));
      tester.ckEquals("ccA", false, pal.isPalindrome("ccA"));
   }
}
```

Case Study 1: Palindrome Strategy

Step 1: If characters 0 and 4 match, continue on. If they don’t, stop.

index:	*0*	*1*	*2*	*3*	*4*
letter:	c	i	v	i	c
	↑				↑

Step 2: If characters 1 and 3 match, continue on. If they don’t, stop.

index:	*0*	*1*	*2*	*3*	*4*
letter:	c	i	v	i	c
		↑		↑	

Step 3: Comparing the same character – can stop.

index:	*0*	*1*	*2*	*3*	*4*
letter:	c	i	v	i	c
			↑↑		

Case Study 1: Four Step Process

Four step process for writing a loop:

1. What must be repeated?

2. What is the test that must be true when the loop finishes?

3. Combine the results of 1 and the negation of 2 to form a loop.

4. Add statements before or after to finish the job.

Case Study 1: Develop Pseudocode

```
public boolean isPalindrome(String p)
{  left = position of first character in string
   right = position of last character in string

   while (left and right have not met/crossed && might be a palindrome)
   {  if (character at left is not the same as character at right)
      {  the string is not a palindrome
      } else
      {  advance left and right to the next positions
      }
   }

   return the answer
}
```

Case Study 1: First Refinement

```
public boolean isPalindrome(String p)
{
   int left = 0;
   int right = p.length() - 1;
   boolean mightBePal = true;

   while (left < right && mightBePal)
   {  char leftCh = p.charAt(left);
      char rightCh = p.charAt(right);

      if (rightCh != leftCh)
      {  mightBePal = false;
      } else
      {  left += 1;
         right -= 1;
      }
   }

   return mightBePal;
}
```

Case Study 1: Second Refinement

```
public boolean isPalindrome(String p)
{  p = p.toLowerCase();

   int left = 0;
   int right = p.length() - 1;
   boolean mightBePal = true;

   while (left < right && mightBePal)
   {  char leftCh = p.charAt(left);
      char rightCh = p.charAt(right);

      if (rightCh < 'a' || rightCh > 'z')          {  right -= 1;          }
      else if (leftCh < 'a' || leftCh > 'z')       {  left += 1;           }
      else if (rightCh != leftCh)                  {  mightBePal = false;  }
      else
      {  left += 1;
         right -= 1;
      }
   }

   return mightBePal;
}
```

7.3.4: Understanding Enumerations

Programmers often need to deal with sets of values:

Gender: male, female

Direction: north, south, east, west

Grade of gasoline: bronze, silver, gold

What can go wrong with code like the following?

```
public static final int EAST = 0;
public static final int SOUTH = 1;
public static final int WEST = 2;
public static final int NORTH = 3;

public void face(int directionToFace)
{ …
```

A better solution…

```
/** An enumeration of the four compass directions.
*
*   @author Byron Weber Becker */
public enum Direction
{
   EAST, SOUTH, WEST, NORTH
}

public class SimpleBot extends Paintable
{
   private int street;
   private int avenue;
   private Direction dir;
   …

   public void face(Direction directionToFace)
   {  while (this.dir != directionToFace)
      {  this.turnLeft();
      }
   }
}
```

Enumerations are like classes:

- Documented the same way.
- Go into their own file.

Values are placed in a comma-separated list.

Case Study 2: A Toll Booth

Write a class, **TollBooth**, that implements the calculations for a toll booth like is found on many highways.

When a vehicle arrives at the booth, a scale calls the **arrival** method, passing the weight of the vehicle as an argument. The toll is assessed according to the chart.

Weight	Toll
1-5000	$0.35
5001-25000	$0.50
>25000	$1.50

Each time a coin is placed in the toll booth's receptacle, the **collectCoin** method is called. The value of the coin is passed as a parameter. An associated display calls **getAmountOwed** to determine how much toll remains to be paid.

The gate mechanism calls the **okToLiftGate** query to determine if the gate should be lifted so the vehicle can pass. After the gate is lifted and the scales determine the vehicle has left, the **departure** method is called.

Calling **getTotalCollected** and **getTotalVehicles** returns the total of the tolls and the total number of vehicles that pass, respectively.

Case Study 2: Beginning the Class

```
public class TollBooth extends 
{
   public TollBooth(                    ) …

   public               arrival(                    )  …

   public               departure(                    )  …

   public               collectCoin(                    )  …

   public               getAmountOwed(                    )  …

   public               okToLiftGate(                    )  …

   public               getTotalCollected(                    )  …

   public               getTotalVehicles(                    )  …

}
```

Add return types and parameters to the method names given in the problem statement. Turn each method into a "stub" for testing by adding just enough of the body so that it will compile.

Case Study 2: Beginning the Class

```
public class TollBooth extends  Object
{
   public TollBooth( See Note )                  {  super();        }

   public  void  arrival( int weight )           {  }

   public  void  departure(  )                   {  }

   public  void  collectCoin( double value )     {  }

   public  double  getAmountOwed(  )             {  return 0.0;     }

   public  boolean  okToLiftGate( )              {  return false;  }

   public  double  getTotalCollected( )          {  return 0.0;     }

   public  int  getTotalVehicles( )              {  return 0;       }

}
```

Note: The class could be made more flexible by passing the weights and associated toll amounts when the **TollBooth** object is constructed – but for now we’ll just use hard-coded values.

Case Study 2: Start a Test Suite

```
import becker.util.Test;

public class TollBooth extends Object
{  public TollBooth()                                   {   super();          }
   public void arrival( int weight )                     {   }
   public void departure(   )                            {   }
   public void collectCoin( double value )               {   }
   public double getAmountOwed(  )                       {   return 0.0;       }
   public boolean okToLiftGate( )                        {   return false;     }
   public double getTotalCollected(  )                   {   return 0.0;       }
   public int getTotalVehicles( )                        {   return 0;         }

   public static void main(String[ ] args)
   {  Test tester = new Test();

      // Test getTotalVehicles
      TollBooth tb = new TollBooth();
      tester.ckEquals("No vehicles yet", 0, tb.getTotalVehicles());
      tb.arrival(3000);
      tb.departure();
      tester.ckEquals("Total: 1", 1, tb.getTotalVehicles());
      tb.arrival(5000);
      tb.departure();
      tester.ckEquals("Total: 2", 2, tb.getTotalVehicles());
   }
}
```

Case Study 2: Passing getTotalVehicles tests

```
public class TollBooth extends Object
{  private int tVehicles = 0;        // total vehicles
   ...
   public void departure( )
   {  this.tVehicles += 1;
   }
   ...
   public int getTotalVehicles( )
   {  return this.tVehicles;
   }

   public static void main(String[ ] args)
   {  Test tester = new Test();

      // Test getTotalVehicles
      TollBooth tb = new TollBooth();
      tester.ckEquals("No vehicles yet", 0, tb.getTotalVehicles());
      tb.arrival(3000);
      tb.departure();
      tester.ckEquals("Total: 1", 1, tb.getTotalVehicles());
      tb.arrival(5000);
      tb.departure();
      tester.ckEquals("Total: 2", 2, tb.getTotalVehicles());
   }
}
```

Case Study 2: Testing getTotalCollected

```
public class TollBooth extends Object
{  private int tVehicles = 0;
   public TollBooth()…
   public void arrival( int weight )…
   public void departure(   )…
   public void collectCoin( double value )…
   public double getAmountOwed(  )…
   public boolean okToLiftGate( )…
   public double getTotalCollected(  )…
   public int getTotalVehicles( )…

   public static void main(String[ ] args)
   {  …
      // Test getTotalCollected
      tb = new TollBooth();
      tester.ckEquals("Nothing collected yet", 0.0, tb.getTotalCollected());
      tb.arrival(3000);
      tb.collectCoin(0.25);
      tb.collectCoin(0.10);
      tb.departure();
      tester.ckEquals("Collected: 1", 0.35, tb.getTotalCollected());
      tb.arrival(5000);
      tb.collectCoin(0.25);
      tb.departure();
      tester.ckEquals("Collected: 2", 0.60, tb.getTotalCollected());
   }
```

Case Study 2: Pass getTotalCollected tests

```
public class TollBooth extends Object
{  private double tCollected;             // total collected
   ...
   public void collectCoin( double value )
   {  this.tCollected += value;
   }
   ...
   public double getTotalCollected(    )
   {  return this.tCollected;

   public static void main(String[ ] args)
   {  ...
      // Test getTotalCollected
      tb = new TollBooth();
      tester.ckEquals("Nothing collected yet", 0.0, tb.getTotalCollected());
      tb.arrival(3000);
      tb.collectCoin(0.25);
      tb.collectCoin(0.10);
      tb.departure();
      tester.ckEquals("Collected: 1", 0.35, tb.getTotalCollected());
      tb.arrival(5000);
      tb.collectCoin(0.25);
      tb.departure();
      tester.ckEquals("Collected: 2", 0.60, tb.getTotalCollected());
```

Case Study 2: Testing getAmountOwed

```
public class TollBooth extends Object
{  private int tVehicles = 0;
   private double tCollected = 0.0;
   public TollBooth()…
   public void arrival( int weight )…
   public void departure(   )…
   public void collectCoin( double value )…
   public double getAmountOwed(   )…
   public boolean okToLiftGate( )…
   public double getTotalCollected( )…
   public int getTotalVehicles( )…

   public static void main(String[ ] args)
   {  …
      // Test getAmountOwed
      tb = new TollBooth();
      tester.ckEquals("Nothing owed yet", 0.0, tb.getAmountOwed());
      tb.arrival(5000);
      tester.ckEquals("owe $0.35", 0.35, tb.getAmountOwed());
      tb.collectCoin(0.25);
      tester.ckEquals("owe $0.10", 0.10, tb.getAmountOwed());
      tb.collectCoin(0.10);
      tester.ckEquals("owe $0.00", 0.00, tb.getAmountOwed());
      tb.departure();
      …
   }
```

Case Study 2: Passing getAmountOwed

```
public class TollBooth extends Object
{  private double vToll;
    …
    public void arrival( int weight )
    {  if (weight <= 5000)
       {  this.vToll = 0.35;
       } else if (weight <= 25000)
       {  this.vToll = 0.50;
       } else
       {  this.vToll = 1.50;
       }
    }

    public void departure( )
    {  this.tVehicles += 1;
       this.vToll = 0.0;
    }

    public double getAmountOwed( )
    {  return this.vToll;
    }

   public void collectCoin( double value )
   {  this.tCollected += value;
      this.vToll -= value;
   }

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

Case Study 2: Testing okToLiftGate

```
public class TollBooth extends Object
{  private int tVehicles = 0;
   private double tCollected = 0.0;
   private double vToll;

    public TollBooth()…                              public double getAmountOwed(  )…
    public void arrival( int weight )…               public boolean okToLiftGate( )…
    public void departure(  )…                       public double getTotalCollected( )…
    public void collectCoin(double value )…          public int getTotalVehicles( )…

   public static void main(String[ ] args)
   {  …
      // Test okToLiftGate
      tb = new TollBooth();
      tester.ckEquals("gate down", false, tb.okToLiftGate());
      tb.arrival(5000);
      tester.ckEquals("gate down", false, tb.okToLiftGate());
      tb.collectCoin(0.25);
      tester.ckEquals("gate down", false, tb.okToLiftGate());
      tb.collectCoin(0.10);
      tester.ckEquals("gate down", true, tb.okToLiftGate());
      tb.departure();
      tester.ckEquals("gate down", false, tb.okToLiftGate());
      …
   }
```

Case Study 2: Passing okToLiftGate tests

```
public class TollBooth extends Object
{  private boolean gateUp = false;        // should the gate be lifted up?
   …
   public void departure( )
   {  this.tVehicles += 1;
      this.vToll = 0.0;
      this.gateUp = false;
   }

   public boolean okToLiftGate()
   {  return this.gateUp;
   }

   public void collectCoin( double value )
   {  this.tCollected += value;
      this.vToll -= value;
      this.gateUp = this.vToll <= 0.0;
   }

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

Case Study2: Completed Class (1/3)

```
import becker.util.Test;

/** A TollBooth collects money from passing vehicles according to a set fee schedule and
* determines when it's ok to lift the gate to allow the vehicles to pass.
*
* @author Byron Weber Becker */
public class TollBooth extends Object
{
   private int tVehicles = 0;              // total number of vehicles
   private double tCollected = 0.0;        // total amount collected
   private double vToll;                   // toll still owing for the current vehicle
   private boolean gateUp = false;         // should the gate be up?

   public TollBooth()
   {  super();
   }

   /** Get the total amount collected in tolls. */
   public double  getTotalCollected()
   {  return this.tCollected;
   }
```

Case Study 2: Completed Class (2/3)

```
/** Get the total number of vehicles that have passed. */
public int  getTotalVehicles()
{  return this.tVehicles;
}

/** A vehicle with the given weight has arrived at the toll booth. */
public  void  arrival( int weight )
{  if (weight <= 5000)
   {  this.vToll = 0.35;
   } else if (weight <= 25000)
   {  this.vToll = 0.50;
   } else
   {  this.vToll = 1.50;
   }
}

/** A vehicle has departed from the toll booth. */
public void  departure()
{  this.tVehicles += 1;
this.vToll = 0.0;
this.gateUp = false;
}
```

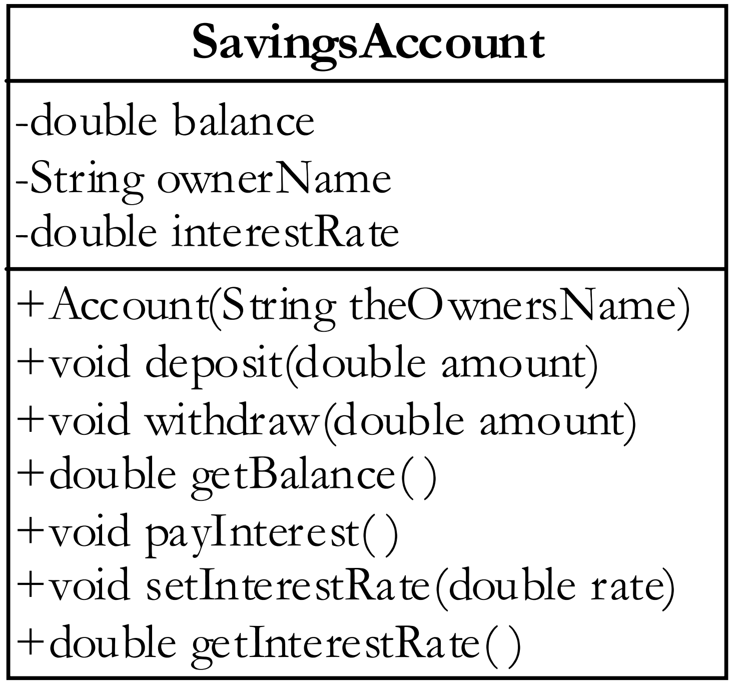
Case Study 2: Completed Class (3/3)

```
    /** Collect a coin in payment for the toll. */
    public void  collectCoin(double value)
    {  this.tCollected += value;
       this.vToll -= value;
       this.gateUp = this.vToll <= 0.0;
    }

    /** Get the amount still owed for the current vehicle's toll. */
    public double  getAmountOwed()
    {  return this.vToll;
    }

    /** Determine whether enough has been paid to lift the gate. */
    public boolean  okToLiftGate()
    {  return this.gateUp;
    }

    /** Test the class. */
    public static void main(String[ ] args)
    {  Test tester = new Test();
       // Test getTotalVehicles
       TollBooth tb = new TollBooth();
      tester.ckEquals("No vehicles yet", 0, tb.getTotalVehicles());
    }
}
```

Many other tests have been omitted here.

7.5.1: Using Class Variables

Instance variables store a value on a per-object basis. Every object has its own copy of the variable that may be different from the value stored by other objects.


SavingsAccount
-double balance
-String ownerName
-double interestRate
+Account(String theOwnersName)
+void deposit(double amount)
+void withdraw(double amount)
+double getBalance( )
+void payInterest( )
+void setInterestRate(double rate)
+double getInterestRate( )

```
public class SavingsAccount extends Object
{  private double balance = 0.0;
   private String ownerName;
   private double interestRate = 0.025;

   public void setInterestRate(double rate)
   {  this.interestRate = rate;
   }
}
```

Every savings account has its own balance and owner.

All savings accounts *should* have the same interest rate – but an instance variables allows them to be different.

Sets the interest rate for only this account.

7.5.1: Using Class Variables

```
public class SavingsAccount extends Object
{  private double balance = 0.0;
   private String ownerName;
   private static double interestRate = 0.025;

   public double getInterestRate()
   {  // The preferred way to reference a class variable.
      return SavingsAccount.interestRate;
   }
   public double getInterestRate()
   {  // Another way to reference a class variable.
      return this.interestRate;
   }
   public double getInterestRate()
   {  // Still another way to reference a class variable.
      return interestRate;
   }

   public void setInterestRate(double rate)
   {  SavingsAccount.interestRate = rate;
   }
}
```

Class variables (also known as *static variables*) are shared by all of the instances of a class.

SavingsAccount
-double balance -String ownerName -static double interestRate
+Account(String theOwnersName) +void deposit(double amount) +void withdraw(double amount) +double getBalance() +void payInterest() +void setInterestRate(double rate) +double getInterestRate()

Sets the interest rate for all accounts.

7.5.1: Assigning Unique ID Numbers

```
public class SavingsAccount extends Object
{  private double balance = 0.0;
   private String ownerName;
   private static double interestRate = 0.025;
   private final int accountNum;
   private static int nextAccountNum = 0;

   public SavingsAccount(String owner)
   {  super();
      this.ownerName = owner;
      this.accountNum = nextAccountNum;
      SavingsAccount.nextAccountNum += 1;
   }

   public double getInterestRate()
   {  // The preferred way to reference a class variable.
      return SavingsAccount.interestRate;
   }

   public void setInterestRate(double rate)
   {  SavingsAccount.interestRate = rate;
   }
}
```

```
public class SavingsAccount extends Object
{  …
   private static double interestRate = 0.025;

   public static double getInterestRate()
   {  return SavingsAccount.interestRate;
   }

   public static void setInterestRate(double rate)
   {  SavingsAccount.interestRate = rate;
   }

   public static void main(String[ ] args)
   {  Test.ckEquals("initial rate", 0.025, SavingsAccount.getInterestRate());
      SavingsAccount bill = new SavingsAccount("Bill Gates");
      SavingsAccount melinda = new SavingsAccount("Melinda Gates");
      Test.ckEquals("initial rate", 0.025, bill.getInterestRate());

      SavingsAccount.setInterestRate(0.030);
      Test.ckEquals("new rate", 0.030, SavingsAccount.getInterestRate());
      Test.ckEquals("new rate", 0.030, bill.getInterestRate());
      Test.ckEquals("new rate", 0.030, melinda.getInterestRate());
   }
}
```

7.5.2: Using Class Methods

Helpful class methods in the Java library include:

In the **Math** class:

```
int abs(int x)
double abs(double x)
double cos(double x)
double log(double x)
int max(int x, int y)
double max(double x, double y)
int min(int x, int y)
double min(double x, double y)
double pow(double x, double y)
double random()
long round(double x)
double sin(double x)
double sqrt(double x)
double tan(double x)
double toDegrees(double x)
double toRadians(double x)
```

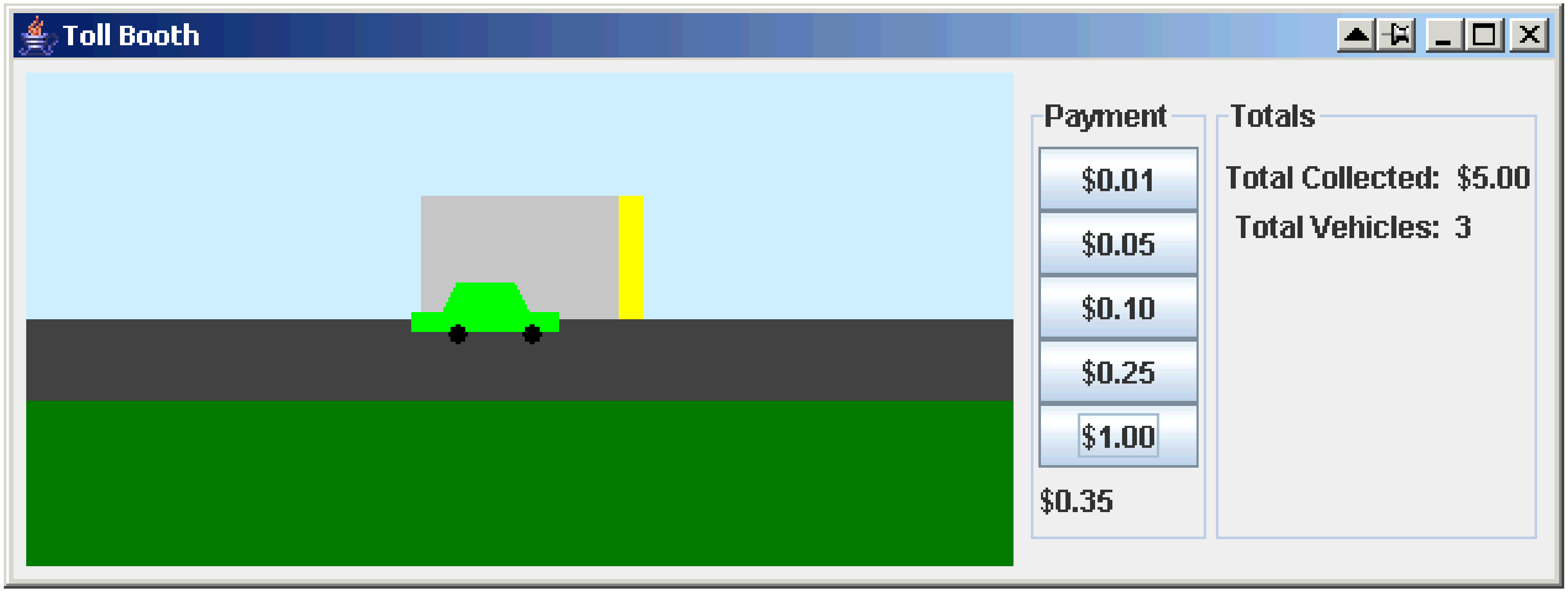
In the **Character** class:

```
boolean isDigit(char ch)
boolean isLetter(char ch)
boolean isLowerCase(char ch)
boolean isUpperCase(char ch)
boolean isWhitespace(char ch)
```

Usage:

```
if (Math.random() < 0.50)
{  // do something about ½ of the time
} else
{  // do something else about ½ of the time
}
```

The toll booth class can be used with a graphical user interface in the **becker** library.



$0.35

Total Collected: $5.00

Total Vehicles: 3

```
public class Main
{  public static void main(String[ ] args)
   {  TollBooth booth = new TollBooth();              // our code
      TollBoothGUI gui = new TollBoothGUI(booth);     // someone else's code
   }
}
```

7.6.1: Specifying Methods with Interfaces

Problem: The graphical user interface (**TollBoothGUI**) has already been written. It needs to call methods in **TollBooth**. How can it be assured that our implementation of **TollBooth** has the required methods with the required names, parameters, and return types?

Solution: The author of the graphical user interface provides a file listing the methods **TollBoothGUI** expects to find in **TollBooth**.

```
public interface ITollBooth
{  public void arrival(int weight);
   public void departure();
   public double getAmountOwed();
   public double getTotalCollected();
   public int getTotalVehicles();
   public void collectCoin(double value);
   public boolean okToLiftGate();
}
```

This file is called an *interface*, which is completely different from a *graphical user interface*. This interface is used to guarantee the presence of the specified methods in a class that implements the interface.

How is the **ITollBooth** interface used?

```
public class TollBoothGUI extends …
{
   private ITollBooth model;
   …
   public TollBoothGUI(ITollBooth model)
   {  super();
      this.model = model;
      …
   }
   …
}

public class TollBooth extends Object implements ITollBooth
{  private int tVehicles = 0;             // total number of vehicles
   private double tCollected = 0.0;       // total amount collected
   …
}
```

7.6.4: Informing User Interface of Changes

```
import becker.util.ViewList;
import becker.util.ViewList;

public class TollBooth extends Object implements ITollBooth
{  …
   private ViewList views = new ViewList();
   …
   public void addView(IView aView)
   {  this.views.add(aView);
   }

    public void arrival(int weight)
    {  …
         this.vToll = 0.35;
       …
       this.views.updateAllViews();
    }

    public void departure()
    {  this.tVehicles += 1;
       …
       this.views.updateAllViews();
    }
}
```

```
public double getTotalCollected()
{  return this.tCollected;
}

public int  getTotalVehicles()
{  return this.tVehicles;
}
```

7.7.1: The Test Harness Pattern

Name: Test Harness

Context: You want to increase the reliability of your code and make the development process easier.

Solution: Write a **main** method in each class used for testing.

```
import becker.util.Test;
public class «className» …
{  // instance variables and methods
   …
   public static void main(String[ ] args)
   {  // Create a known situation
      «className» «instance» = new «className»(…);
      // Execute the code being tested
      «instance».«methodToTest»(…);
      // Verify the results
      Test.ckEquals(«idString», «expectedValue», «actualValue»);
      …
```

Consequences: Writing tests before coding helps focus development and ensure quality.

Related Patterns: This pattern is a specialization of Java Program.

7.7.2: The toString Pattern

Name: toString

Context: You want to easily print information about an object, often for debugging.

Solution: Override the **toString** method in each class to print out relevant information.

```
public String toString()
{  return  "«className»[" +
           "«instanceVarName1»=" + this.«instanceVarName1» +
           ", «instanceVarName2»=" + this.«instanceVarName2» +
           ...
           ", «instanceVarNameN»=" + this.«instanceVarNameN» +
           "]";
}
```

Consequences: This method is called automatically by **print** and **println**, making it easy to print relevant information.

Related Patterns: This is a specialization of the Query pattern.

7.7.3: The Enumeration Pattern

Name: Enumeration

Context: You would like variables to hold a value from a specific set of values such as **MALE** or **FEMALE** or one of the four directions.

Solution: Define an enumeration type listing the desired values.

```
public enum «typeName»
{ «valueName1», «valueName2», «valueName3», …, «valueNameN»
}
```

For example:

```
public enum JeanStyle
{ CLASSIC, RELAXED, BOOT_CUT, LOW_RISE, STRAIGHT
}

pbulic class DenimJeans
{ private JeanStyle style;
  public DenimJeans(JeanStyle aStyle)…
```

Consequences: Enumeration variables may have only values defined in the enumeration, helping to avoid programming errors.

Related Patterns: Named Constant

7.7.4: The Assign a Unique ID Pattern

Name: Assign a Unique ID

Context: Each instance of a class requires a unique identifier.

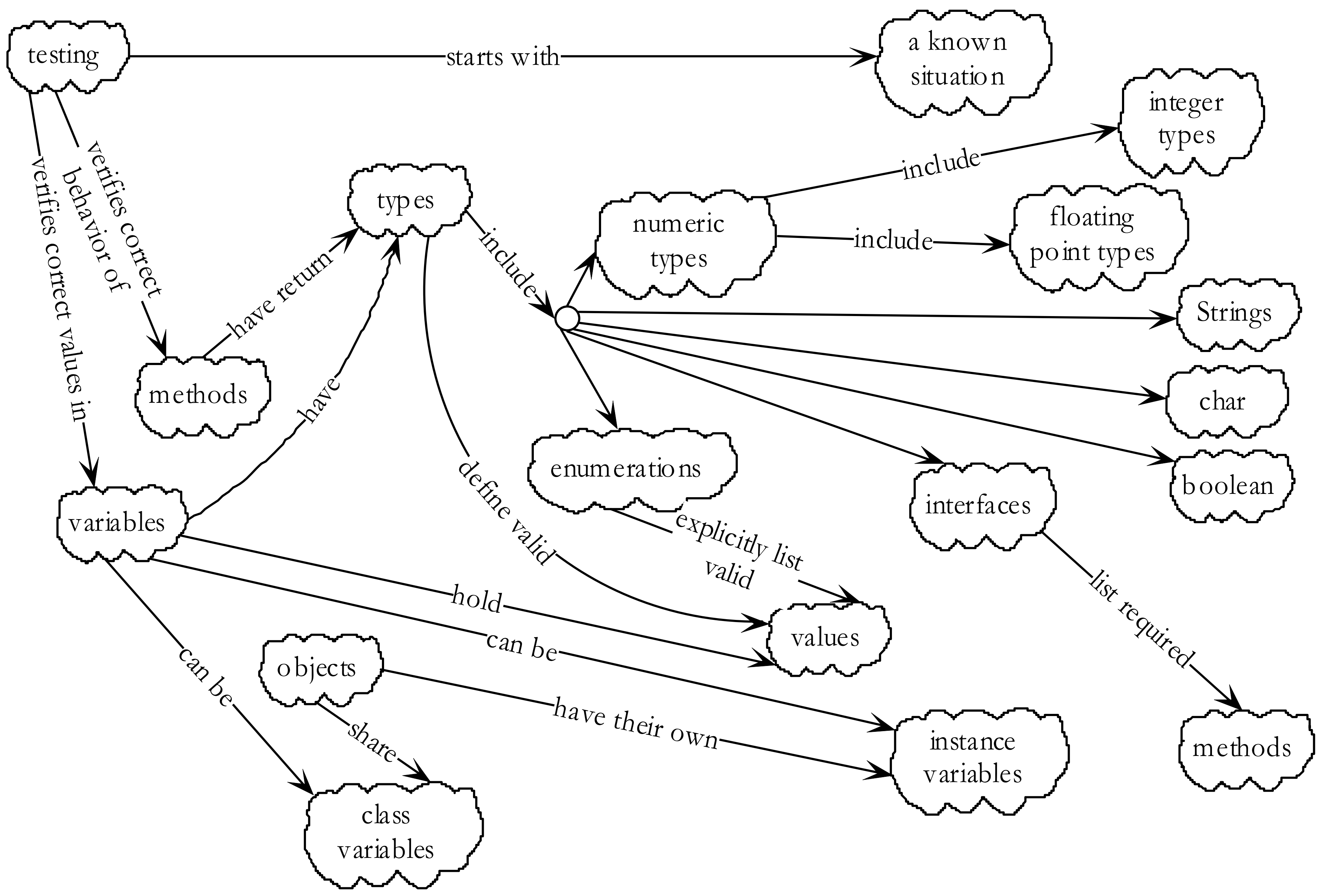
Solution: Store the unique ID in an instance variable. Use a class variable to maintain the next ID to assign.

```
public class «className»…
{  private int «uniqueID»;
   private static int «nextUniqueID» = «firstID»;

   public «className»(…)
   {  …
      this.«uniqueID» = «className».«nextUniqueID»;
      «className».«nextUniqueID» += 1;
   }
}
```

Consequences: Unique identifiers are assigned to each instance of the class for each execution of the program.

Related Patterns: This pattern makes use of the Instance Variable pattern.

7.8: Concept Map



Summary

We have learned:

- how to test a class with its own **main** method.
- about numeric types such as **int** and **double**, including their differing ranges and precision, converting between types, formatting, and shortcuts such as **+=**.
- about non-numeric types, including **boolean**, **char**, **String**, and enumerated types.
- how to use these types in a class that had nothing to do with robots.
- about class variables and methods.
- how to use a Java interface to make a class we write work with a class written by someone else, such as a graphic user interface.