<< Lecture 7 >>

Announcements

- **Updates to the syllabus**
  - Decided that you can miss 2 quizzes with no consequences
  - Midterm 1: Friday, October 9
  - Midterm 2: Wednesday, November 18
  - Final exam: to be announced by the school

Little review questions

1. What will be printed in the *standard output*?

```
String a, b;

a = "hello worldo!";
b = a;
a = null;

System.out.println( a );
System.out.println( b );
```

Ans.

```
null
hello worldo!
```

2. Correct the following code to prevent it from crashing; consider general code errors and special input that can make it fail

```
/* This method sums the sizes of the arrays supplied through
   the sarray parameter */

public int sumLength( String[] sarray ) {
   int c = 0;

   for( int i = 0 ; i <= sarray.length ; i = i + 1 )
      c = c + sarray[i];

   return c;
}
```

Ans.

```
public int sumLength( String[] sarray ) {
   if ( sarray == null ) return 0;
   int c = 0;
   for( int i = 0 ; i < sarray.length ; i = i + 1 )
      if ( sarray[i] != null ) c = c + sarray[i].length;
   return c;
}
```

Iteration 2 ???

- We will begin iteration 2 by reviewing what we have seen already, but adding slightly more detail to everything
  - Most details are necessary to know, but not really as practical (you can easily work around them)
- Then we will cover the object oriented paradigm—that's the point of iteration 2
- But let us start reviewing some basic classes

The Math class

- The **Math** class contains a collection of mathematical functions
- **Everything in this class is static**—the perfect example of a class composed solely of handy functions
- Example methods:

```
// Absolute value
Math.abs( -5 );       // 5

// Power (exponentiation)
Math.pow( 3, 2 );     // 9.0 ; this is 3 squared
Math.pow( 9, 0.5 );   // 3.0 ; this is the square root of 9
Math.pow( 10, -1 );   // 0.1 ; negative exponentiation
Math.pow( 1, 5 );     // 1.0 is neutral under multiplication

// the exponent function
Math.exp( 3 );        // 20.0855...

// the natural logarithm function
Math.log( 20.086 );   // 3.000...
Math.log( 1 );        // 0.0

// the logarithm, base 10
Math.log10( 100 );    // 2.0
Math.log10( 0.1 );    // -1.0

// the pi and e constants (static final)
Math.PI;              // 3.149265...
Math.E;               // 2.718281...

// the max and min functions (pseudo-useful)
Math.max( -0.5, 20 );    // 20.0
Math.min( -0.5, 20 );    // -0.5

// the trigonometric functions
Math.cos( Math.PI/3 );   // 0.5000...
Math.sin( Math.PI/6 );   // 0.4999...
Math.acos( 0.5 );        // 1.047... (Math.PI/3)

// and a long et cetera . . .
```

- If we try to instantiate **Math**, we get an error! The constructor is private (yes, we can make our constructors private), which means that it cannot be used outside of **Math**! (Tricky stuff)

String basics

- There is a **String** class, so variables of type **String** contain objects!
- The Java compiler allows you to create new instances of Strings easily, just for convenience

```
String x = null;
x = "this is a String";
```

- Arrays can also be initialized conveniently, remember?

```
int[] z = null;
z = {0, 1, 2, 3, 4, 5};
```

- Again, all for the sake of convenience
- Each time we create a String, we have a new object (duh!)

```
System.out.println( "test" == "test" );
System.out.println( "test".equals("test") );
```

- Fortunately, there is the **String.equals** method
- String concatenation is done using its own method

```
System.out.println( "this is " + "a String" );
System.out.println( "this is ".concat( "a String" ) );
```

- The fact that we can do concatenation using the + symbol is just a very nice convenience—we can't overload this operator :(
  - Note that this is not exactly **String.concat**; the left side of + can be integer, etc., and it allows **null** values
- Strings have a nice relation with arrays of characters

```
char[] A = { 'H','e','l','l','o' };
String b = new String(A);
System.out.println( b );

char[] C = "World".toCharArray();
for(char d: C) System.out.print(d);
```

- Other basic String methods include **charAt** and **length**

```
String a = "aeiou";

System.out.println( a.length() );    // this is 5
System.out.println( a.charAt(4) );   //indices in 0..4
```

- Finding a character or a substring is done through **indexOf**

```
String a = "serious gaming research";

System.out.println( a.indexOf('g') );          //  8
System.out.println( a.indexOf('g',10) );       // 13
System.out.println( a.indexOf("gam") );        //  8
System.out.println( a.indexOf("gam",10) );     // -1
```

Classes for basic types

- There are help classes for basic (primitive) types as well
- They are hybrid! They have static and non static members

```
Integer x = new Integer( 200.0 );   // error
Integer x = new Integer( 200 );     // good!
x.intValue();      // 200
x.doubleValue();   // 200.0
x.shortValue();    // 200
x.byteValue();     // -56  … byte has 127 as maximum
x.toString();      // "200"
x.equals( new Integer(200) );   //true

Integer.parseInt("200");   // 200
```

- I hope you paid attention to the last line—the Integer class has a static method for translating Strings into ints!
- The other number types have similar behavior

```
Double x = new Double( 20 );     // good!
Double x = new Double( 20.5 );   // good!
x.intValue();      // 20
x.doubleValue();   // 20.5
x.shortValue();    // 20
x.byteValue();     // 20

x.toString();      // "20.5"
x.equals( new Float( 20.5 ) );  // false—demands same type

Double.parseDouble("20.5");     // 20.5
```

- Booleans also get their special convenient class

```
Boolean x = new Boolean( 25 );      // BAD
Boolean x = new Boolean( true );    // good!
x.booleanValue()   // true

x.toString();      // "true"

Boolean.parseBoolean("TrUe");   // true!!!
```

- Method **Boolean.parseBoolean(String)** is not sensitive to case!
- **Character** has also some pretty useful static members

```
Character.isAlphabetic( 'c' );      // true
Character.isDigit( 'c' );           // false
Character.isLetterOrDigit( '9' );   // true
Character.isIdeographic( '喳' );     // true    (forbidden in exams)
Character.isUppercase( 'x' );       // false
Character.isWhitespace( '\t' );     // true
Character.toUpperCase( 'a' );       // 'A'
```

Variable scope

- Variables in Java exist within their block and its nested blocks, but Java is object oriented...
- Variables can be defined in two levels:
  - at the class level—i.e., as fields
  - at the method level
    - function parameters (or arguments)
    - in the blocks within the method
    - can't reuse a name

```
class ScopeDemo {

  int x = 5, y = 6;

  public void someMethod( int x ) {
    int x = 4;   // WRONG
    for(int y = 0; y < 10; y = y + 1 ) {
      this.x = y;
      System.out.println("> "+this.x+","+x+";  "+this.y+","+y);
    }
  }
}

public class Demo{
  public static void main(String[] x) {
    ScopeDemo sd = new ScopeDemo();
    sd.someMethod( -100 );
  }
}
```

Summarized operations

- **!=** : distinct—evaluates to true when the operands are different
  - for a,b: **a != b** is the same as **!(a==b)**
  - a,b still must be of the same kind/type!

```
System.out.println( 5 != 3 );    // true
```

- +=, -=, *=, /= : updating assignments
  - **a** = **a** + **c** can be summarized in **a** += **c**
  - other operations follow by analogy

```
int a = 15;
a /= 3;
System.out.println( a );    // 5
```

- ++, --: increment, decrement by one
    - ++**x** : increment x first, then return its value
    - **x**++ : return x's value, then increment it
    - -- follows by analogy

```
int x = 0;
System.out.println( x++ );  // 0
System.out.println( x++ );  // 1
System.out.println( ++x );  // 3
System.out.println( x++ );  // 3
System.out.println(x);      // 4
```

Type conversion

- **This will be very important later on!**
- In Java, it is possible to convert a value from one type to another; this is called **casting**
- Below is a simple example with numbers:

```
double x = 27.5;
System.out.println( x );  // 27.5
int y = (int) x ;
System.out.println( y );  // 27
```

- If we antecede a value by **(type)**, we are telling the compiler to convert the type of the value to this new type

Method overloading

- It is possible to define several methods with the same name within the same class—*how come!?*
- If the methods differ in the number of parameters or the types of the parameters, then there is no ambiguity!
- Java automatically picks the method that *best matches* the caller's supplied parameters
- If the type of an parameter does not match any of the supplied types, Java will try casting; the method that matches with the *closest type* will be used

```
/*
 * Example of a class with overloaded methods
 * Zup has one public, overloaded method: typ
 * typ might accept double or String as parameter */

class Zup {
  private void print(String x) { System.out.println(x); }

  public void typ(double x) { print("double"); }

  public void typ(String x) { print("String"); }
}
```

```
// Demo.main instantiates Zup and makes use of its overloaded methods

public class Demo{
  public static void main(String[] args) {
    Zup z = new Zup();
    z.typ( 5 );
    z.typ( "hello worldo" );
    z.typ( null );
  }
}
```

- Java is smart enough: **5** goes as **double**, not **int**, and **null** goes as **String**

```
// Now, we have defined Zup(String) and Zup(int[])

class Zup {
  private void print(String x) { System.out.println(x); }

  public void typ(String x) { print("String"); }

  public void typ(int[] x)  { print("Integer Array"); }
}

public class Demo{
  public static void main(String[] args) {
    Zup z = new Zup();
    z.typ( "hello worldo" );
    z.typ( null );             // WRONG!
    z.typ( (int[]) null );     // Correct
  }
}
```

- Now, the **null** is ambiguous—we can resolve this with **casting**

```
// Now, we have defined Zup(int) and Zup(double)

class Zup {
  private void print(String x) { System.out.println(x); }

  public void typ(int x)     { print("Integer"); }

  public void typ(String x) { print("String"); }
}

public class Demo{
  public static void main(String[] args) {
    Zup z = new Zup();
    z.typ( 5.0 );       // WRONG
    z.typ( 5 );         // Correct
    z.typ( "hello worldo" );
  }
}
```

- Java does not like destroying precision all that much—double won't be treated as int, but the converse is acceptable

<< Lecture 8 will resume from here >>