COMP 250

Lecture 6

Objects & Classes 1:

- String, *wrapper classes*, Math, defining our own classes, constructors

this

Jan. 19, 2022
Java comes with many built-in classes:

- **String**
- **Wrapper classes** such as:
  - **Boolean**
  - **Byte**
  - **Character**
  - **Integer**
  - ....

We can also define our own classes.

Classes are “reference types” rather than “primitive types”.

(Other reference types include arrays – see last lecture.)
String Examples

We can make a string object in different ways, e.g.:

```java
String s = "Hello";
String s1 = new String("Hello "); // allowed but unnecessary
```

This is similar to how there are different ways to make arrays.
String Examples

There are several methods associated with strings. We call the methods using the dot notation as follows:

```java
String s = "Hello";

int m = s.length(); // Then m would have the value 5.

char c = s.charAt(1); // Then c would have the value 'e';
```
String s = "Hello";

int m;

char c;

m = s.indexOf('o'); m is 4.

m = s.indexOf('p'); m is -1 (indicating ‘not found’)

c = s.charAt(8); Produces a runtime error

StringIndexOutOfBoundsException
String concatenation

String s0 = "Hello" ;
String s1 = " there" ;

The following expressions (and more) each produce the string "Hello there".

"Hello" + " there"

s0 + s1

s0.concat(s1)

"Hello".concat(" there")
Compare Strings using `equals()`

```java
String s0 = "Hello" ;
String s1 = "Hello" ;

boolean b = s0.equals( s1 ); // true
```

The `equals()` method goes through each character of the two strings and verifies that they are the same.

_A common mistake made by Java programmers to compare strings using the "==" operator instead of `equals()`_. See next slide(s).
As we will see, when the "==" operator compares objects, it checks if two objects are the same.

So, you might expect the following expressions to evaluate to false i.e. the left and right side are different (but equal) strings.

"surprise" == "surprise"       // evaluates to true
"sur" + "prise" == "surprise"  // evaluates to true

The reason the first result is true is that the Java compiler creates a list of constants that the program will need, and it only makes one copy of each constant.
For the second example, the compiler does the concatenation "sur" + "prise" in advance, so again there is just one string "surprise".

ASIDE: why not compare Strings using == operator?
ASIDE: why not compare Strings using == operator?

Consider a different example in which a string is computed at run time.

String s = "sur";

s + "prise" == "surprise" evaluates to false

The reason is that there are two String objects created at runtime.

(s + "prise").equals("surprise") evaluates to true which is what we want.

Bottom line: it is always safer to use equals when comparing strings.
String  name  =  “Suzanne Fortier” ;
       name = name.toUpperCase();

The second line assigns name the string “SUZANNE FORTIER”

A common mistake is to write just

       name.toUpperCase()

and assume that this changes the string that name references. It doesn’t. Rather, a new (upper case) string is created and returned. You need to write that returned string somewhere.
Strings are “immutable”

String objects cannot be changed.

```java
String s = "cats";

s.charAt(0) = 'r';  // compile-time error!
```

You cannot use the `charAt` method in this way.

There is no `String` method that allows us to set the value of a particular character. Rather, one would have to make a new string.
(There are various `String` methods that can help you do that. Details omitted here.)
Primitive types

boolean b  false
char c      N
int i       -2498
double x   34.679

Reference types (so far...)

char[] charArray

-2498
34.679
0  1  2  3  4
L  U  N  C  H

String s  “LUNCH”

Let’s look at more examples of reference types.
Wrapper classes

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Byte</td>
</tr>
<tr>
<td>short</td>
<td>Short</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
</tbody>
</table>

How do we use these?
One way we use wrapper classes is to define constants:

Byte.MAX_VALUE has value $2^7 - 1$

Short.MAX_VALUE has value $2^{15} - 1$

Integer.MAX_VALUE has value $2^{31} - 1$

Long.MAX_VALUE has value $2^{63} - 1$

Float.MAX_VALUE and Double.MAX_VALUE have the largest (finite) values that you can represent with a float or double, respectively.

Use MIN_VALUE instead of MAX_VALUE to get the smallest negative values.
Another way we use wrapper classes is to convert from a String to a number:

To convert from a String to an int, use:

```java
int i = Integer.parseInt("54");
```

To convert from a String to an Integer, use:

```java
Integer j = Integer.valueOf("54");
```

To convert from a String to a double, use:

```java
double z = Double.parseDouble("2.7");
```

To convert from a String to a Double, use:

```java
Double y = Double.valueOf("2.7");
```

The names of these methods don’t clarify at all what is the difference between them!

I can’t think of better name for them. Calling them “convertToPrimitiveInt()” and “convertToWrapperInteger()” would have been awkward :/.
Initializing a wrapper class variable

Boolean b = new Boolean(true)   Boolean.valueOf(true);
Integer i = new Integer(-45)   Integer.valueOf(-45);
Double x = new Double(3.75);  Double.valueOf(3.75);

The wrapper classes constructors were “deprecated” as of Java 8.
Autoboxing and Unboxing ("wrapper")

Alternatively, we can write:

```java
Boolean b = true;
Integer i = -45;
Double x = 3.75;
```

The compiler replaces this code with what I wrote on the previous slide. This is called **autoboxing**. (It is reminiscent of casting but it is not the same thing.)

Going in the opposite direction is called **unboxing**.

```java
boolean b1 = b;
integer i1 = i;
double x1 = x;
```
Check out fields & methods for wrapper classes at the Java API e.g. **Integer**

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>static int</strong></td>
<td><code>bitCount(int i)</code></td>
</tr>
<tr>
<td></td>
<td>Returns the number of one-bits in the two's complement binary representation of the specified int value.</td>
</tr>
<tr>
<td>byte</td>
<td><code>byteValue()</code></td>
</tr>
<tr>
<td></td>
<td>Returns the value of this Integer as a byte after a narrowing primitive conversion.</td>
</tr>
<tr>
<td><strong>static int</strong></td>
<td><code>compare(int x, int y)</code></td>
</tr>
<tr>
<td></td>
<td>Compares two int values numerically.</td>
</tr>
<tr>
<td>int</td>
<td><code>compareTo(Integer anotherInteger)</code></td>
</tr>
<tr>
<td></td>
<td>Compares two Integer objects numerically.</td>
</tr>
<tr>
<td><strong>static int</strong></td>
<td><code>compareUnsigned(int x, int y)</code></td>
</tr>
<tr>
<td></td>
<td>Compares two int values numerically treating the values as unsigned.</td>
</tr>
<tr>
<td>static <code>Integer</code></td>
<td><code>decode(String nm)</code></td>
</tr>
<tr>
<td></td>
<td>Decodes a String into an Integer.</td>
</tr>
<tr>
<td><strong>static int</strong></td>
<td><code>divideUnsigned(int dividend, int divisor)</code></td>
</tr>
<tr>
<td></td>
<td>Returns the unsigned quotient of dividing the first argument by the second where each argument and the result is interpreted as an unsigned value.</td>
</tr>
<tr>
<td>double</td>
<td><code>doubleValue()</code></td>
</tr>
<tr>
<td></td>
<td>Returns the value of this Integer as a double after a widening primitive conversion.</td>
</tr>
<tr>
<td>boolean</td>
<td><code>equals(Object obj)</code></td>
</tr>
<tr>
<td></td>
<td>Compares this object to the specified object.</td>
</tr>
<tr>
<td>float</td>
<td><code>floatValue()</code></td>
</tr>
<tr>
<td></td>
<td>Returns the value of this Integer as a float after a widening primitive conversion.</td>
</tr>
</tbody>
</table>
Big Picture Brief Summary

**Primitive types**

- char \( c \) = \('3'\)
- int \( i \) = -4598
- double \( x \) = 3.75

**Reference types**

- String \( s \)
  - “3.75”
- int[] \( intArray \)
  - 0 = 32
  - 1 = -5
  - 2 = 4335
  - 3 = 234
  - 4 = -12

- Double \( x \)
  - 3.75
- Integer \( i \)
  - -4598

...and other wrapper class examples
Reference variables and objects

The arrows are *references to “objects”*. The value of a reference variable can be thought of as the address of an object in memory. That’s what we mean by the arrow. (More generally, it is some id or number that uniquely identifies the object.)

...and other wrapper class examples
COMP 250

Lecture 6

Objects & Classes 1: String, wrapper classes, Math, defining our own classes, constructors this

Jan. 19, 2022
Math

- Math.PI is the value $\pi$ ← this is a field, not a method

Suppose that we declare a variable double x;

- Math.sqrt(x) returns the value $\sqrt{x}$.

- Math.random() returns a random number in (0,1).

- Math.log(x) returns the value $\log_{e}x$ or $\ln(x)$.

- Math.log10(x) returns the value $\log_{10}x$.  (There is no method for taking log to a given base b.)

- Math.sin(x) returns the value $\sin(x)$.  

As we saw in lecture 4, Java has many pre-defined reference types, or “classes”.

They are organized into packages.

Examples of packages from the “standard Java library”:

java.awt
java.util
java.lang
Defining your own class

class ClassName {
    // field declarations
    // method declarations
}

Example:

class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
Java naming conventions

Class names begin with an upper case letter (String, Integer, Math, ...).

Constants should be all upper case, e.g. Math.PI

Variables, methods, package names (and some other things) begin with a lower case character.

e.g. Integer j = Integer.valueOf("54");
Constructors

class ClassName {

    // field declarations

    // method declarations

    ClassName( ) {  // constructor methods have no return type
        // their method name is same as class name

        // instructions in constructor method
    }

}
A constructor with no arguments is called a “no-argument constructor”. It could have an empty body, or it could have instructions in the body such a print statement, or it might assign default values to the fields, e.g. `x = 5;` ...
Default Constructor

class Point2D {
    int x;
    int y;

    // Point2D(){ };
    The compiler would essentially create this method.
}

If you don’t explicitly define any constructor for your class, then the compiler makes a “default constructor” for you, namely a no-argument constructor.

The no-argument and default constructors both initialize the fields to a default value of 0, 0.0, ‘\u0000’, false, or null depending on the type.
new keyword

To create (construct) an object, use the `new` keyword and a constructor method which has the name of the object’s class (except for wrapper classes – see earlier).

Some method might have the following instruction:

```java
Point2D p1 = new Point2D();
```

![Diagram of creating a Point2D object with reference variable `p1` and its int x and y values set to 0.](image)
Point2D p1 = new Point2D();

The method can then change the values in the object’s $x$ and $y$ fields:

```java
p1.x = 23;
p1.y = 85;
```
Constructors with arguments

We can define constructors that have arguments, for example, that assign values to the fields of the object.

```java
class Point2D {
    int x;
    int y;

    Point2D(int x0, int y0)
    {
        x = x0;
        y = y0;
    }
}
```

We can call this constructor as follows:

```java
Point2D p1 = new Point2D(23, 85);
```
Non-default constructors & “overloading”

class Point2D {
    int x;
    int y;

    Point2D(){ } // “no argument” constructor

    Point2D(int x0, int y0){
        x = x0;
        y = y0;
    }
}

If we define a (non-default) constructor that has some parameter(s), and if we also want to have a no-argument constructor, then we must explicitly define the no-argument constructor. Otherwise, the no-argument constructor won’t exist.
class Point2D {
    int x;
    int y;

    Point2D(int x0, int y0) {
        x = x0;
        y = y0;
    }

    void moveTo(int x0, int y0) {
        x = x0;
        y = y0;
    }

    void moveBy(int deltaX, int deltaY) {
        x = x + deltaX;
        y = y + deltaY;
    }
}

other method declarations
class Point2D {
    int x;
    int y;

    Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }

    void moveTo(int x, int y) {
        this.x = x;
        this.y = y;
    }

    void moveBy(int deltaX, int deltaY) {
        this.x += deltaX;
        this.y += deltaY;
    }
}

‘this’ allows having variable names that are the same as the field name, making the code easier to read.

this refers to the Point2D object being constructed.

this refers to the Point2D object that is calling ("invoking") the method.
public class AnotherClass {

    public static void main ( String[] args ) {

        Point2D   p1 = new Point2D(3, 4);
        p1.moveTo( 7, 7 );

        Point2D   p2 = new Point2D(8, 2);
        p2.moveBy( 2, 0 );
    }
}
Coming up...

Lectures

Fri. Jan. 21  objects & classes 2  aliasing, static, scope, ...

Homework (TODO)

Basic Java coding exercises (with solutions)

Assignment 1 to be posted Fri. Jan. 28  (2 weeks).