

Miscellaneous Java

...or “things we should talk about at some point, and now’s a good time.”

ArrayList

Problems with Arrays

- Array objects must be declared with a fixed size:

```
int[] myArray = new int[1000];
```

 - This can hold at most 1000 items.
 - Trying to use the 1001st element will cause an error
 - Can’t be expanded after declaration.
- Might not need all of the capacity.
 - wastes memory
 - must keep a separate counter with the number of real values in the array.

ArrayList

- One of the classes available in the Java library.
 - a “resizable array” of objects
- some methods implement array-like behaviour:

```
al.set(0, "abc"); // like array[0] = "abc";
x = al.get(0); // like x = array[0];
l = al.size(); // like l = array.length;
```
- also allows shortening/lengthening:

```
al.add("def"); // append to end
al.remove(0); // remove element 0
```

ArrayList

- ArrayList is much more like the Python list type.
- ArrayList can only hold objects
 - no primitive types: int, char, etc.
 - must specify type when creating: the type is like:

```
ArrayList<String>
```
- If we want to store primitive types, they have to be somehow “converted” to objects. (later)
- See Java docs for ArrayList details.

ArrayList Example

```
ArrayList<String> al = new ArrayList<String>();

// add some objects
al.add("zero");
al.add("one");
al.add("two");
System.out.println(al);
// output: [zero, one, two]

// delete an element
al.remove(1);
System.out.println(al);
// output: [zero, two]
```

Objects → Strings

Printing Objects

- When we print an object...
 - ArrayLists gives nice output:
[one, two, three]
 - but when we print a Student, it's not so useful:
Student@82ba41
- The Student is using the default method for printing an object.
 - can be overridden

The toString Method

- When `System.out.print` is given an object, it calls the object's `toString()` method.
 - ie.

```
Student s = new Student(...);  
System.out.print(s);
```


... makes a call to `s.toString()` and prints that.
- Student uses the default `toString()` method.
 - ... but we can write our own.
 - ArrayList already has a nice `toString()` method.

toString Example

```
class Student {  
    ...  
    public String toString() {  
        return Long.toString(studentNumber)  
            + ": " + lastName  
            + ", " + firstName;  
    }  
    ...  
}
```

Using toString

- Now, printing a Student will give output like:
300012345: Simpson, Rudiger
- Can also be called manually, outside of a print:

```
String s = someObject.toString() + "x";
```
- Many classes from the standard library have `toString` methods that can be used (at least) for debugging.

Wrapper Classes

Wrapper Classes

- An ArrayList can only store objects
 - ... not fundamental types (int, char, etc).
- There are other cases when it would be useful to treat fundamental types as objects as well.
- For each fundamental type, there is a corresponding “wrapper class”.
 - holds the same info as the type, but does it in an object.

Example: Integer

- The Integer class is the wrapper for int.
 - constructor for Integer can take an int:
`Integer i = new Integer(234);`
- This can then be used as an object:
`ArrayList<Integer> al = new ArrayList<Integer>();`
`al.add(i);`
- Can be converted back to fundamental type:
`int i2 = i.intValue();`

Wrapper Classes

- All of the fundamental types have a corresponding wrapper class:
 - Byte, Short, Integer, Long, Float, Double, Character, Boolean, Void
- The classes also contain **static** functions to convert Strings to the corresponding class.
 - eg. `Double d = Double.parseDouble("1.3");`

Overloading Methods

Argument Types

- The print method can take many types as its argument:
`System.out.print(16); // int`
`System.out.print(true); // boolean`
`System.out.print("Hello"); // Object (String)`
`System.out.print(s); // Object (Student)`
- How would we define such a function?
`public static void print(???)`
 - must specify a type for the argument: no type will do.

Overloading

- There are actually several different “print” functions, with different argument types.
`public static void print(int i) { ... }`
`public static void print(boolean b) { ... }`
`public static void print(String s) { ... }`
`public static void print(Object obj) { ... }`
- An “overloaded” function/method
 - The compiler matches the arguments you give with the functions available.
 - Possible because Java is strongly typed.

Creating Overloaded Methods

- **Only if** you have a similar operation to do on different types...
- Create separate functions for each set of arguments.
 - must have different “signatures”: different types/numbers of arguments
 - The compiler will try match the arguments you give with the available signatures and “bind” to a particular definition.

Formatting Output

Formatting Output

- The default output from `System.out.print` isn't always formatted the way we want.
 - eg. `System.out.println(3.0/7);`
 - ... output: `0.42857142857142855`
- It's also hard to combine many values.
 - eg. produce “`3 + 4 = 7`” from `a=3` and `b=4`.
 - Would have to print five things separately:
`a, " + ", b, " = ", a+b`

The `printf` Method

- The `System.out.printf` method can output values based on a “format string”.
 - like C's `printf` and Python's `%` operator.
 - new in Java 5.0
- eg.

```
System.out.printf("%d + %d = %d\n",  
a, b, a+b);
```

Format Strings

- A String object, mostly left as-is.
- Replacements are marked with a “`%`”.
- Common types:

<code>%d</code>	int, long, ...
<code>%f</code>	float, double (with decimals)
<code>%e</code>	float, double (scientific notation)
<code>%g</code>	float, double (chooses either <code>%f</code> or <code>%e</code>)
<code>%s</code>	String
<code>%%</code>	create a %

Format Details

- Can control number of characters printed and decimal places
 - eg. `%10.2f` replacement will take 10 characters and have 2 decimal places: “`34.21`”
 - eg. `%8d` takes 8 characters: “`32`”
- Can also control other details
 - eg. `%08d` replacement will take 8 characters, padded with zeroes: “`00000032`”

String.format

- Another way to do string formatting
 - Return a String object, instead of printing to the screen.
- Use the static function in the String class: String.format .
- eg.

```
String s = String.format(
    "%d + %d = %d\n", a, b, a+b);
```

String Formatting Examples

- Print powers of 2 in columns:

```
for(int i=0; i<=10; i++) {
    System.out.printf( "%2d  %6.0f\n",
        i, Math.pow(2, i) );
}
```
- The toString calculation from Student class:

```
return String.format("%09d: %s. %s",
    studentNumber, lastName, firstName);
```