DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 281 NOTES
Spring 2019: Amber Settle

Wednesday, April 17, 2019

Announcements

- Discuss the first assignment
- The second assignment is due tonight at 9 pm – questions?
- The third assignment is due Wednesday, April 24th

Using predefined classes

There are a (large) number of classes defined in Java. A class is defined to create objects, which can be used in our program to help us solve problems.

An **object** has three things:

1. **Identity**: We can distinguish multiple instances of the same kind ("class") of object; this is done using the identifier for the object
2. **State**: Every object has certain properties that characterize it; these are the value(s) of the object
3. **Behavior**: Every object has certain actions that it can perform; these are the method(s) of the object

To see what we mean by this, we will consider several classes.

The String class

Strings are objects representing character strings.

To **create a String object**:

```
String stringName = new String("some character string");
```

Note: We must supply the text for the contents of the string. This is also a declaration and assignment of a variable, although not one of a primitive type.

Examples:

- String name = new String("Amber Settle");
- String address = new String("243 S. Wabash Ave.");

Once we have a string, what can we do with it?

Some **methods of the String** class:

- `int length()` : returns the number of characters in the String

- `char` **`charAt`**`(int index)` : returns the character in the input position 'index'
- `int` **`indexOf`**`(char target)` : returns the index of the first occurrence of the input character 'target' or –1 if it does not occur in the String; `indexOf` can also take a String as a parameter and it will return the index of the first occurrence of the parameter in the object.
- `String` **`substring`**`(int start, int end)` : returns a String that represents a range of characters from positions 'start' to 'end'-1
- `String` **`toLowerCase`**`():` returns the String that is the same as the calling String except that all upper case letters of the String have been replaced with the equivalent lower case letters
- `String` **`toUpperCase`**`():` returns the String that is the same as the calling String except that all lower case letters of the String have been replaced with the equivalent upper case letters
- `boolean` **`startsWith`**`(String prefix, int toffset):` returns True if the String begins with prefix, starting at the specified index toffset and False otherwise.
- `boolean` **`endsWith`**`(String suffix):` returns True if the String ends with suffix and False otherwise.
- `boolean` **`contains`**`(CharSequence s):` returns True if the String contains the specified sequence of char values and False otherwise.
- `boolean` **`equals`** `(Object anObject):` compares anObject with the calling String; returns true if the argument is not null and a String object that represents the same sequence of characters as the calling object, and false otherwise.
- `boolean` **`equalsIgnoreCase`** `(String anotherString):` returns the same result as equals but ignoring case for the two Strings.
- `int` **`compareTo`**`(String anotherString):` compares two Strings lexicographically; returns a negative value if the first String comes before the second String, 0 if the Strings are equal, and a positive value if the first String comes after the second String.
- `int` **`compareToIgnoreCase`**`(String str):` performs the same way as compareTo, but ignores case.

In general, the form for describing a method (the header of the method) is:

```
returnType methodName(types and names of parameters)
```

Note: Indexing in strings (like all arrays) starts at 0.

Obtaining characters as input

We can use String method charAt **to obtain a single character of input** from the user of our programs.

To do this, you input a String using the Scanner class and then use the charAt method to obtain the first character of the String.

The Character class

The Character class provides methods that allow programs to manipulate characters.

It is located in the java.lang package. You use methods from the class the same way you would use methods from the Math class, by either writing the name of the class, followed by the dot operator, followed by the name of the method or by using a static import statement.

The methods of the class include:

- **isDigit**(char ch): returns true if ch is a digit and false otherwise
- **isLetter**(char ch): returns true if ch is a letter, lower or upper case, and false otherwise
- **isLowerCase**(char ch): returns true if ch is a lowercase letter and false otherwise
- **isUpperCase**(char ch): returns true if ch is an uppercase letter and false otherwise
- **isSpaceChar**(char ch): returns true is ch is the space character (not the tab or newline) and false otherwise
- **isWhitespace**(char ch): returns true if ch is any white space character (including the tab, return, newline, or space) and false otherwise
- **toLowerCase**(char ch): returns the character that is the lower case version of ch or ch itself if there is no lower case equivalent
- **toUpperCase**(char ch): the same as above but for upper case

See an example using the Character and String methods in **StringDemo.java**.

The StringBuilder class

StringBuilder objects are like String objects, except that **they can be modified**.

Internally, these objects are **treated like variable-length arrays that contain a sequence of characters**. At any point, the length and content of the sequence can be changed through method invocations.

A string builder object has two properties that are important as well as methods to access them:

- The **length**, accessed by the `length()` method, returns the length of the character sequence in the builder.
- The **capacity**, accessed by the `capacity()` method, returns the number of character spaces that have been allocated.

The capacity is always larger than the length and will expand as necessary to accommodate additions to the string builder.

The **constructors** for the StringBuilder class can take a variety of parameters:

- `StringBuilder()`: Creates an empty string builder with a capacity of 16 characters.
- `StringBuilder(CharSequence cs):` Constructs a string builder containing the same characters as the specified CharSequence, plus an extra 16 empty elements.
- `StringBuilder(int initCapacity)`: Creates an empty string builder with the specified initial capacity.
- `StringBuilder(String s)`: Creates a string builder whose value is initialized by the specified string, plus an extra 16 empty elements trailing the string.

The **important methods** for the StringBuilder class are as follows:

- `StringBuilder append(type o)`: Append o to the end of the string builder where type can be one of Boolean, char, char[], double, float, int, long, Object, and String.
- `StringBuilder delete(int start, int end)`: Delete the subsequence from start to end-1 (inclusive).
- `StringBuilder delete(int index)`: Delete the character at the specified index.
- `StringBuilder insert(int offset, type o)`: Insert the object o into the string builder at the index immediately following the offset.
- `String toString()`: Return the string builder as a String.

Complete the required code in the **StringBuildEx.java** file.

Arrays of objects

The elements of an array do not have to be a primitive data type.
They can also be objects from any class that we have available to us.
(E.g. Strings, etc.)

Consider **an array of String objects**:

```
String[] sentence = new String[20];
```

This sets up an array of references to 20 String objects. The objects themselves have not been created. As it stands, this is an array of 20 null references.

You must set up each of these objects individually:

```
for (int i = 0; i < sentence.length; i++)
```

```
sentence[i] = new String("default");
```

Note: For Strings we can actually use an initializer list of 20 constant Strings. This is not true in general for objects, particularly the ones we define ourselves.

Objects as parameters to methods

When you pass arrays or objects to methods you need to clearly understand what is possible with those calls.

We discuss a couple of the examples now.

Arrays as parameters

When we pass an array as a parameter to a method, what is passed is a reference to where the array is stored. This value cannot be changed, but the individual values stored as the elements of the array can.

Example: **PassingArrays.java**

1. Basic function of the code
2. Static method to print the array
3. Create the array trashing method to demonstrate call-by-reference

Visualize the execution using http://pythontutor.com.

On the other hand, when we pass a single array element to a method, it is like passing any other single piece of data to a method.

It is copied and passed by value if it is a primitive type.
It is passed by reference if it is an object.

Example: **PassingArrays.java**

Add the trashElement method

Visualize the execution using http://pythontutor.com.

Strings as parameters

When we pass a String to a method, it is like passing any other single piece of data to a method.

Example: **PassingStrings.java**

1. Change the calls to the ones using a String
2. The method that destroys the String doesn't impact the String in the main method.

Visualize the execution using http://pythontutor.com.

Debugging Java programs

For the rest of the time this week we will focus on debugging Java programs.

There are **two general types of mistakes** we will consider:

1. **Compilation errors**: mistakes that prevent your program from being compiled
2. **Run-time errors**: mistakes that prevent your compiled code from executing correctly.

Both types of mistakes are important to be able to overcome, and you will be expected in CSC 300 (and beyond) to handle them.

Compilation errors

A compilation error is a mistake in syntax: you have failed to write correct Java syntax and the IDE will force you to fix it before you can run the program.

In this class (and others) you are expected to submit programs that compile. Programs that do not compile typically earn little to no points.

The following are very common compilation problems and worth remembering:

1. Missing semicolons
2. Unclosed parentheses
3. Unclosed brackets
4. Missing types for variables
5. Incorrect headers and modifiers
6. Incorrect return types
7. Failure to import necessary classes

Consider the mistakes in the **CompileError.java** program and fix them one-by-one, noting the mistakes in the comments.

Note **that Eclipse is helpful** in that it examines your code as you type it and tries to warn you of problems you will encounter when trying to compile.

It describes those problems in red Xs next to the problematic lines.

Run-time errors

A run-time error occurs in a program that has correct syntax but incorrect logic (in some way) that prevents the program from executing as intended.

The rest of this section describes some approaches to use when debugging your program.

Many of these ideas come from a YouTube video that you might find helpful: https://www.youtube.com/watch?v=9gAjIQc4bPU

Note though that the video discusses classes, which we haven't discussed yet. But will be more useful in about two weeks.

We will illustrate these ideas on the **RuntimeErrors.java** program.

Setting breakpoints

A simple way to step through your program is to set breakpoints.

To set a breakpoint in your source code, right-click in the small left margin in your source code editor and select *Toggle Breakpoint*.

Alternatively, you can double-click on this position.

Example: Illustrate on the printArray method in the **RuntimeErrors.java** file. Set a breakpoint at the top of the for loop and the body of the for loop.

After you've set a breakpoint you can **begin debugging your program by clicking on the debug option on the main menu**. It looks like a tiny bug.

Once you've done this you'll move into the debug perspective. The first time you do it, it will prompt you as to whether you want to move into that perspective. Say yes!

When you enter the debug perspective, unless your first breakpoint is at the first line of the program, you'll need to **move to the point where you've set your breakpoint**.

To do that you click Step Into (or F5).

You can now see the variables that are relevant to that piece of code.

Example: So what is the problem with the printArray method in the **RuntimeErrors.java** file?

Debugger basics

The following are basic commands that are useful when you use the debugger:

- To **terminate** the debugger you press the red (stop) button.
- To move into a line with a break point, press the **Step Into** button (F5)

- To move past a line of code you're convinced is correct, press the **Step Over** button (F6)
- To complete the current method/function quickly, you click the **Step Return** button
- To switch back to the source code view, click the **Resource** button at the top right of the window

Useful settings in the debugger

To ensure that you don't try to debug code that's built into Java, you need to set your preferences.

To do that go to Window → Preferences → Java → Debug → Step Filtering

Check the box that says Use Step Filters and then check all of the associated boxes. Click Ok to save these changes.

Tracking expressions

If there is an expression that is important to watch in trying to understand your program, you can set a watch for the expression.

To do this, highlight the expression and right click. Choose Watch from the options on the menu.

You will then see another section pop up in the menu on the top left which provides you with expressions of interest.

Example: What is the problem with the fillArray method in the **RuntimeErrors.java** file?