Java Libraries

Lecture 8
CGS 3416 Spring 2017

February 13, 2017

Intro to Libraries

- We've barely scratched the surface of Java, but before we proceed with programming concepts, we need to talk about the Java API.
- We would also like to be able to start using existing libraries in the Java SDK as quickly as possible.
- To that aim, this outline will provide "just enough" to illustrate basic core usage of existing Java class libraries.
- The Java API can be found at `https://docs.oracle.com/javase/8/docs/api/`

User vs. Builder

With any re-usable programming construct, we have two points-of-view that should always be considered:

- The **builder** is responsible for declaring and defining how some module works.
- The **user** (or **caller**) is somebody (i.e. some portion of code, often some other module) that makes use of an existing module to perform a task.
- For the purposes of this topic (Using Java Libraries), we are looking at things from the user's perspective.
- In other words, what do we need to know to **use** an existing Java library from the SDK, along with it's various already-defined features.

We will look at how to **build** things like functions, classes, interfaces, etc. later on.

What's in the Java SDK?

- There are multiple kinds of library constructs to consider, including:
  - classes
  - interfaces
  - packages
  - classes and interfaces with generic type parameters
- Classes and interfaces are grouped into packages.
- packages are named into categories and subcategories, separated by the dot-operator. Examples of packages:
  - `java.lang`
  - `java.util`
  - `java.util.concurrent`

What's in the Java SDK?

- If a class is inside a package, we can refer to the whole name by referring to the package name, dot-operator, then class name. Examples:
  - `java.lang.String`
  - `java.util.Scanner`
- classes and interfaces can contain:
  - fields (i.e. data variables)
  - methods (i.e. member functions)

Right now, we will focus on the usage of class libraries.

The import Statement

- If you are using any item from the package `java.lang`, you don't need to do anything special.
- Everything from `java.lang` is automatically imported for use into every Java program you write.
- For a class out of *any other* package, you need to put an import statement at the top of your file, to let the Java tools (compiler and loader) know what libraries to pull in.
- Basic form:

```
import <package_name>.<class_name>;
```

- Examples:

```
import java.util.Scanner;
import javax.swing.JFrame;
import java.awt.geom.GeneralPath;
```

The import Statement

- **Wildcards** - if you are going to be using many classes from the same package, you can tell the compiler to import all classes from a single package with the * wildcard character (meaning "all"), in place of a single class name. Examples:

```
import javax.swing.*; // imports all classes in
          javax.swing package
import java.util.*; // imports all classes in
          java.util package
```

- Note that in this last one, for example, it does not import all classes in the sub-package `java.util.concurrent`. It only imports classes directly inside the base package that is specified.

API Descriptions

- The API description for a Java class gives all of the information you need to be able to syntactically use it correctly.
- Starts with description of the class, in a general documentation format.
- **Field Summary**
  - This section lists data fields that you might want to use
  - Often, these are constants, but not always
  - This chart lists the variable names, descriptions, and their types
- **Constructor Summary**
  - This section lists the constructor methods that are available for this class
  - Constructors are related to the creation of objects
  - This chart provides the parameter list for each constructor option

API Descriptions

- **Method Summary**
  - This section lists the methods that are available for this class
  - For general class usage, this will typically be the most relevant set of features that you will want to call upon
  - This chart provides the full prototype, or declaration, of each method
  - first column shows the return type, and whether the method is static or not (more on this later)
  - Second column provides method name, as well as list of expected parameters, and a short description
- For all of these items, the names (of the variables, constructors, and methods) are also links to more detailed descriptions of the items, which are further down the page.

static fields and methods

- Some fields and methods are declared as `static`
- In the Field Summary and/or Method Summary, this information would show up in the left column.
- If a variable or method is **not** declared with the word static, then we call it an instance variable or method.
- To call upon variables or methods from a class, we use the dot-operator syntax. There is a difference between static and instance items, though.
- For a static variable or method, we use this format:

```
className.fieldName                  // fields
className.methodName(arguments)          // methods
```

java.lang.Math Library

- API: java.lang.Math
- Note that all fields and methods in this class are `static`
- class Math has two fields, which are common mathematical constants. Sample usage:

```
double area = Math.PI * radius * radius;
          // compute area of a circle
```

- Sample calls to static methods from Math:

```
area = Math.PI * Math.pow(radius, 2);
           // area of circle, using power method
y = Math.abs(x);
           // computes absolute value of x
System.out.print(Math.random());
           // prints random value in range [0,1)
int die = (int)(Math.random() * 6) + 1;
           // roll a standard 6-sided die
```

Instance Fields and Methods

- Recall that an *instance* field or method is one that is *not* declared to be static. Instance is the default.
- To call upon instance fields or methods in a class library, you have to create one or more *objects* from that class
  - A class is a blueprint for building objects.
- Syntax for building an object:
  `className variable = new className(parameter(s));`
- In this format, the first part is the declaration of a *reference variable*
  `className variableName`
- new is a keyword of the language, and that part of the statement builds a "new" object, and runs a special initialization function called a *constructor*. This is what the parameters are for.

Examples

```
Scanner input = new Scanner(System.in);
JButton myButton = new JButton("Click Me");
String s1 = new String();
```

Once you have declared one or more objects, call upon fields and methods with the dot-operator, as before, but for instance members, use the object's name (i.e. the reference variable) on the left of the dot:

```
objectName.fieldName                  // fields
objectName.methodName(arguments)      // methods
```

Example uses:

```
int x = input.nextInt();
myButton.setText("Stop clicking me!");
System.out.print(s1.toUpperCase());
```

java.util.Random Library

- API: java.util.Random
- This library is for generating pseudo-random numbers
- How computers do "random" number generation
  - It's really a "pseudo-random" generator
  - Start with a "seed" value
  - The seed is used as the input to an algorithm, which generates a seemingly randomized number
  - Each "random" value generated becomes the seed for the next one
  - Start with the same seed, and you'll get the same random numbers!

Some Examples

- Creating objects of this type:

```
Random r1 = new Random();
      // uses the system time to create seed
Random r2 = new Random(1234);
      // uses 1234 as the seed
```

- In the above statements, r1 and r2 refer to objects of type Random – they both can generate a pseudo-random sequence of values

- Sample calls to these objects:

```
int x = r1.nextInt();     // gets a random integer
int y = r1.nextInt(10);
           // gets a random integer from 0-9
double z = r1.nextDouble();
           // gets a random double in range [0,1)
```