CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today:
Fields vs local variables and scope
Program Structure; the keyword static
Classes vs objects
Creating and using objects

Next time: Creating Java programs with multiple files; public vs private;
Object-Oriented Design; Abstract data types; Stacks and Queues.

Reading assignments are be posted on the web site!

Java Program Structure: Class = Container

A Java class can be thought of as a container for methods:

```java
public class OverloadTest {
    static int sum(int n, int m) {
        System.out.println("Calling sum(int n, int m)...");
        return (n+m);
    }
    static double sum(double x, double y) {
        System.out.println("Calling sum(double x, double y)...");
        return (x+y);
    }
    public static void main(String[] args) {
        System.out.println("\nTry sum(2,3)...");
        int n = sum(2, 3);
        System.out.println("Returns " + n);
        System.out.println("\nTry sum(2.3, 3.1)...");
        double x = sum(2.3, 3.1);
        System.out.println("Returns " + x);
        System.out.println("\nTry sum(2, 3.1)...");
        double y = sum(2, 3.1);
        System.out.println("Returns " + y);
    }
}
```
Java Program Structure: Class = Container

The contents of a class can be in any order, as far as execution is concerned: usually main is last, and the other members of the class are organized for readability: put related methods next to each other.

```java
public class OverloadTest {
    public static void main(String[] args) {
        System.out.println("inTry sum2,3,3,1...;");
        int n = sum(2, 3);
        System.out.println("Returns " + n);
        System.out.println("inTry sum2,3,3,1...;");
        double x = sum(2.3, 3.1);
        System.out.println("Returns " + x);
        System.out.println("inTry sum2,3,3,1...;");
        double y = sum(2, 3.3);
        System.out.println("Returns " + y);
    }
    static int sum(int n, int x) {
        System.out.println("Calling sum(int n, int x)...;");
        return (n*x);
    }
}
```

Java Program Structure: Class = Container

A class can also hold variables, which are called fields (go figure!), and can even hold other class definitions (called inner classes). We will focus on fields for now. Let's consider a class MyMath, which will provide some basic math functions:

```java
public class MyMath {
    static double add(double x, double y) {
        return (x + y);
    }
    public static void main(String[] args) {
        System.out.println("add(2,3) => 5.0");
    }
}
```
Java Program Structure: Class = Container

Suppose we add a method which calculates the \( \log_2(\cdot) \) of a double:

```java
public class MyMath {
    static double add(double x, double y) {
        return (x + y);
    }
    static double log2(double x) {
        return Math.log(x) / Math.log(2.0);
    }
    public static void main(String[] args) {
        System.out.println("add(2.3) => "+add(2.3));
        System.out.println("log2(2.3) => "+log2(2.3));
    }
}
```

Recall:

\[
\log_a(B) * \log_b(C) = \log_a(C)
\]

so if \( A = e \) and \( B = 2 \):

\[
\log_2(C) = \log(C) / \log(2)
\]

But it is inefficient to calculate Math.log(2.0) each time, so we add it as a field to the container:

```java
public class MyMath {
    static double logOfTwo = Math.log(2.0);
    static double add(double x, double y) {
        return (x + y);
    }
    static double log2(double x) {
        return Math.log(x) / logOfTwo;
    }
    public static void main(String[] args) {
        System.out.println("add(2.3) => "+add(2.3));
        System.out.println("log2(2.8) => "+log2(2.8));
    }
}
```
Java Program Structure:  Class = Container

Since we can put it anywhere, we put it near its only use the program:

```java
public class MyMath {
    static double add(double x, double y) {
        return (x + y);
    }
    static double logOfTwo = Math.log(2.0);
    static double log2(double x) {
        return Math.log(x) / logOfTwo;
    }
    public static void main(String[] args) {
        System.out.println("add(2,3) => "+ add(2,3));
        System.out.println("log2(8.0) => "+ log2(8.0));
    }
}
```

Java Program Structure:  Class = Container

One more refinement: logOfTwo is actually being used as a constant value, which should never change: to make sure we don’t change it, we make it final:

```java
public class MyMath {
    static double add(double x, double y) {
        return (x + y);
    }
    static final double logOfTwo = Math.log(2.0);
    static double log2(double x) {
        return Math.log(x) / logOfTwo;
    }
    public static void main(String[] args) {
        System.out.println("add(2,3) => "+ add(2,3));
        System.out.println("log2(8.0) => "+ log2(8.0));
    }
}
```
Java Program Structure: Class = Container

One more refinement: `logOfTwo` is actually being used as a constant value, which should never change: to make sure we don’t change it, we make it `final`, so that if we accidentally try to modify it, we will get an error:

```
static final double logOfTwo = Math.log(2.0);
static double log2(double x) {
  logOfTwo = 0.693;//logOfTwo;
  return Math.log(x) / logOfTwo;
}
```

Summary: a Java class is a container for methods (including main), fields, and final fields (constants). Fields can be initialized just like local variables, but final fields can not be modified after initialization:

```
public class MyMath {
  static double add(double x, double y) { return (x + y); }
  static final double logOfTwo = Math.log(2.0);
  static double log2(double x) {
    return Math.log(x) / logOfTwo;
  }
  static double z = 8.0; // just an example
  public static void main(String[] args) {
    System.out.println("add(2,3) => " + add(2,3));
    System.out.println("log2(8.0) => " + log2(z));
  }
}
```
Scope of the members of a class: Since order does not matter, the scope of a method or a field is the entire class:

```java
public class MyMath {
    static double add(double x, double y) {
        return (x + y);
    }
    static final double logOfTwo = Math.log(2.0);
    static double log2(double x) {
        return Math.log(x) / logOfTwo;
    }
    static double z = 8.0; // just an example
    public static void main(String[] args) {
        System.out.println("add(2.3) => " + add(2.3));
        System.out.println("log2(8.0) => " + log2(z));
    }
}
```

Scope of all members:
- Add
- logOfTwo
- log2
- z
- main
Java Program Structure: Scope of fields and methods

Scope of the members of a class: Since order does not matter, the scope of a method or a field is the entire class:

```java
public class MyMath {
    static double add(double x, double y) {
        return (x + y); // Scope of x, y
    }
    static double log2(double x) {
        return Math.log(x) / logOfTwo; // Scope of x
    }
    public static void main(String[] args) {
        System.out.println("add(2,3) => " + add(2,3));
        System.out.println("log2(8.0) => " + log2(8.0));
    }
    static final double logOfTwo = Math.log(2.0); // Scope of logOfTwo
    static double z = 8.0; // just an example
}
```

The scope rule for members means you can call a method to initialize a field!

```java
> run MyMath
add(2,3) => 5.0
log2(8.0) => 3.0
```

Java Program Structure: Scope of fields and methods

Scope of the members of a class: Since order does not matter, the scope of a method or a field is the entire class:

```java
public class MyMath {
    static double add(double x, double y) {
        return (x + y); // Scope of x, y
    }
    static double log2(double x) {
        return Math.log(x) / logOfTwo; // Scope of x
    }
    public static void main(String[] args) {
        System.out.println("add(2,3) => " + add(2,3));
        System.out.println("log2(8.0) => " + log2(8.0));
    }
    static final double logOfTwo = Math.log(2.0); // Scope of logOfTwo
    static double z = log2(256.0); // just an example
    public static void main(String[] args) {
        System.out.println("add(2,3) => " + add(2,3));
        System.out.println("log2(8.0) => " + log2(8.0));
    }
}
```
Summary:

A Java class is an unordered container for its members (methods and fields);

The scope of a member declaration is the entire class (more on this later),
unlike local variables inside methods, whose scope is from the declaration to
the next unmatched right curly brace;

Fields can be initialized just like local variables, even using methods in the
class, but final fields can not be modified after initialization.

Java Program Structure: Static Containers

What about the keyword static?

There are two different ways classes can be used to compute:

The first is as a static container for its members.

When an entity is static it:

1. Is created when you first run the program;

2. Has a single instance which exists during the entire run of your program;
   and

3. Is destroyed only when your program terminates.
There is in fact a static region of your program in memory: this region contains all static members of classes; other regions of memory are dynamic and store values of local variables in method calls and entities created by new:

- **RAM**
  - Static Region: holds your code and all static members of classes.
  - The Heap: holds entities created by `new` (e.g., arrays).
  - Run-Time Stack: holds values of local variables during method calls

- **Static Memory**: Only thing that can change here is values of fields.

- **Dynamic Memory**: for anything else that changes during execution.

So we can think of a running program as existing in two different “worlds,” static and dynamic:
When you run your program, a static instance of your class is created, exists for the entire run, and is destroyed only when your program terminates:

Java Program Structure: Static Containers

Static World | Dynamic World
---|---
MyMath
- add
- logOfTwo
- log2
- z
- main

When you create a new entity using `new`, it exists in the Dynamic world; entities in the Static World are called Classes and entities in the Dynamic world are called Objects.

Java Program Structure: Static Containers

Static World | Dynamic World
---|---
MyMath
- int[] A = new int[5];

The array A:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>
A Java class can be simply used as a Static container for its members, and used by other programs; this is a way of creating your own libraries (such as Math); just like the Math library, you refer to the methods using the name of the class:

Java Program Structure: Programs spread over multiple files.

Static World

```
MyMath
  add
  log2
  logOfTwo
  main

ClientOfMyMath
  main .... MyMath.add(2,3)
```

Dynamic World

MyMath.java

```java
public class MyMath {
    static double add(double x, double y) {
        return (x + y);
    }
    static final double logOfTwo = Math.log2(2.0);
    static double log2(double x) {
        return Math.log(x) / logOfTwo;
    }
    static double z = 8.8;  // just an example
    public static void main(String[] args) {
        System.out.println("add(2,3) => " + add(2,3));
        System.out.println("log2(8.8) => " + log2(8.8));
    }
}
```

ClientOfMyMath.java

```java
public class ClientOfMyMath {
    public static void main(String[] args) {
        System.out.println("MyMath.add(2,3) => " + MyMath.add(2,3));
        System.out.println("MyMath.log2(8.8) => " + MyMath.log2(8.8));
    }
}
```
Note that you call your own static container library program just like you call other static libraries in Java (String, Character, Math): you use the name of the class plus a dot ".".

```java
public class MyMath {
    static double add(double x, double y) {
        return (x + y);
    }
    static final double log10Two = Math.log(10) / Math.log(2);
    static double log(double x) {
        return Math.log(x) / log10Two;
    }
    static double x = 5.0;  // just as an example
    public static void main(String[] args) {
        System.out.println("MyMath.add(2,3) => ")
            + MyMath.add(2,3));
        System.out.println("MyMath.log(8.0) => ")
            + MyMath.log(8.0));
        System.out.println("MyMath.log10Two => ")
            + MyMath.log10Two);
    }
}
```

Notice also that the library program still contains a `main(…)` method, and can still be run like a normal program. Usually, the main method of a library is used for testing code.

Java Program Structure: Creating and using Objects

Creating Objects: Recall that when we declare an array in a method, we are creating a new object that lives in dynamic memory:

```
Static World
Dynamic World
```

```
MyMath

\[
\begin{align*}
\text{add} & : (\mathbb{R} \times \mathbb{R}) & \rightarrow & \mathbb{R} \\
\text{log} & : \mathbb{R} & \rightarrow & \mathbb{R} \\
\text{log10Two} & : \mathbb{R} & \rightarrow & \mathbb{R}
\end{align*}
\]

The array A:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>
```
Java Program Structure: Creating and using Objects

But, we can create new "classes" which contain members just like static classes....

Static World

Dynamic World

Geometry

main ....
Point p = new Point()

p
x: 0.0
y: 0.0

But, we can create new "classes" which contain members just like static classes.....

CS112 Homework

Geometry.java

```java
public class Geometry {
    public static void main(String[] args) {
        Point p = new Point();
        p.x = 2.3;
        p.y = 4.5;
        System.out.println("p = (" + p.x + "," + p.y + ")");
    }
}
```

Point.java

```java
public class Point {
    double x = 8.0; // create two doubles
    double y = 8.0; // and initialize to 8.0
}
```

> run Geometry
p = (2.3,4.5)
But, we can create new "classes" which contain members just like static classes.....

Static World

Dynamic World

Geometry

```
Point p = new Point();
p.x = 2.3;
p.y = 4.5
```