

Lecture 06: Structure

Sierra College
CSCI-12
Spring 2015
Weds 02/11/15

Announcements

- **Schedule**
 - Reminder: no classes NEXT Monday 2/16 (President's Day)
- **Past due assignments**
 - HW03: Why Code, accepted thru Fri 2/13 @ 11pm
 - LAB04: Hello Again, accepted thru Tues 2/17 @ 11pm
- **Current assignments**
 - HW05: Numbers, due Fri 2/13 @ 11pm (*lab time today*)
- **New assignments**
 - HW06: Template, due **Weds** 2/18 @ 11pm (*lab time today*)
 - Create a simple Java program skeleton for future programs

Lecture Topics

- **Last time:**
 - A short history of computing language evolution
 - Initial overview of object-orientation
 - An intro to Java (history, features, advantages)
- **Today:**
 - Beginnings of the language
 - Basic structural elements of Java

Program Building Blocks

- We begin a more detailed treatment of Java by looking at the language's "**building blocks**"
 - More formally in CS, these are known as "**lexical elements**"
- These are the **fundamental language elements** that we'll use all semester as we examine and create Java classes
 - They are the content we put into "containers" such as classes or methods
- These language elements are common to most any other programming language
 - Terminology may differ between languages
 - But, the concepts remain identical
 - Learn it well once, then it will look familiar in your next language

Program Building Blocks List

- In the approximate following order over next 4 lectures:
 - **Structure**
 - Application shell, containers
 - Identifiers, reserved words
 - Statements, expressions, blocks
 - Whitespace, comments
 - **Data**
 - Variables, datatypes
 - Literals, constants
 - **Operations**
 - Operators
 - Precedence, mixed types
 - **Methods**
 - Structure
 - Usage
- *“Tear Java down, then build it back up”*

What's An Application Shell?

- An **application shell** is the simplest possible outline of some Java code
- It is a Java **class**
 - It is the container in which other Java statements reside
 - Every Java program requires at least one class
- It may or may not contain a **main()** method
 - If it is a top-level standalone application, it requires a **main() method**
 - If it is a reusable component, it does not require a **main()** method

Revised Structure for Hello Again

HelloWorld.java

```
main() {  
}
```

```
printGreeting() {  
}
```

HelloAgain.java

```
main() {  
}
```

Hello.java

```
firstName  
lastName
```

```
printGreeting() {  
}
```

```
setFirstName() {  
}
```

```
setLastName() {  
}
```

Application Shell Concepts

- Everything in Java is a **class**
- Each class must be contained in a source .java file matching the class name
- Every Java application contains one or more classes
- Every standalone Java application requires at least one **main() method**, which indicates where execution is to begin
- A reusable Java class component does NOT require a main() method
 - But you might want to add one for testing purposes

Application Shell Examples

- Both of these code snippets will compile, as they are valid Java code

```
public class TemplateClassRL {  
    }L
```

```
public class TemplateMainRL {  
    public static void main(String [] args) {  
    }  
}L
```

Hello World

- A standard first test app in many languages or in many programming environments
- Trivially simple and not terribly useful
- YET, it proves some important things:
 - You can get something simple compiled and executed
 - Interaction between language and tool is OK
 - How to echo simple text to the screen
 - Gets something simple working FIRST, so you can then “complexify” it
 - Gives you a working starting point for the next program

Our Hello World Example

```
1  /*
2  * Name:      Rob Lapkass
3  *
4  * Course:    CS-12, Fall 2014
5  *
6  * Date:      08/27/14
7  *
8  * Filename:   HelloWorldRL.java
9  *
10 * Purpose:    Simple "Hello World" Java application to test the language.
11 */
12
13 public class HelloWorldRL {
14
15     public static void main(String [] args) {
16
17         printGreeting("Rob", "Lapkass");
18
19     }
20
21     private static void printGreeting(String firstName, String lastName) {
22
23         System.out.println("Hello " + firstName +
24
25                             " " + lastName +
26
27                             ", good to have you in class");
28
29     }
30
31 }
```

Containers

- We said earlier that all information to a computer is just binary 1s and 0s
- All information represents one of only two things:
 - Data
 - Instructions
- In Java, there are 3 types of containers to hold such information:
 - **Variables** contain data
 - **Methods** contain instructions
 - **Classes** contain/combine data and instructions
 - A class can contain just data or just instructions
 - But, both together is more typical

Container Examples



```
13 public class HelloRL {
14
15     // instance variables: what the class IS
16     private String firstName;
17     private String lastName;
18
19     // constructor: initializes the class
20     public HelloRL() {
21         firstName = "Anonymous";
22         lastName = "Student";
23     }
24
25     // methods: what a class DOES
26
27     // prints a greeting given first and last names
28     public void printGreeting() {
29         System.out.println("Hello " + firstName +
30                             " " + lastName +
31                             ", good to have you in class.");
32     }
33
34     // updates the first name
35     public void setFirstName(String first) {
36         firstName = first;
37     }
38
39     // updates the last name
40     public void setLastName(String last) {
41         lastName = last;
42     }
43
44 }
```

Identifiers

- Java **identifiers** are user-specified **symbolic names**
- They are given to new containers that WE create, such as:
 - Variables (and constants)
 - Methods
 - Classes
- Identifiers must follow certain Java **naming requirements**
- Identifiers should follow certain standard Java **naming conventions**
 - We'll cover conventions for “good” names as we introduce the various language elements (variables, methods, classes)

Identifier Requirements

- Per the Java language specification, identifiers:
 - **Must** be composed of **Java letters** in any combination
 - a-z, A-Z, 0-9, _ (underscore), \$ (dollar sign)
 - Also many Unicode characters
 - **Must not** contain any spaces
 - **Must not** begin with a digit 0-9
 - **Must not** be Java **reserved words** (coming up...)
 - Are **case-sensitive**: Name1 ≠ name1
 - Length: essentially unlimited

Identifier Examples

Valid Java identifiers

- x
- Abc1
- student1, Student1
- taxes2013
- voidCheck
- classSize
- _systemInfo, system_info
- \$systemInfo
- aReallyReallyLongName

Invalid Java identifiers

- 4
- 1abc
- studentID#
- student 1
- 2013taxes
- void
- class
- @systemInfo

Identifier Examples

- This is your first program from Week 1
- Text underlined in RED indicates identifiers

```
public class HelloWorldRL {  
    public static void main(String [] args) {  
        printGreeting("Rob", "Lapkass");  
    }  
    private static void printGreeting(String firstName, String lastName) {  
        System.out.println("Hello " + firstName + " " + lastName + ", good to have you in class");  
    }  
}
```

Reserved Words

- **Reserved words** (or **keywords**) are symbols with special, recognized meanings in the Java language
 - Collectively, they are Java's "vocabulary"
 - Similar to the list of words in a dictionary
- We cannot use any of these as identifiers in any code that we write
 - Even though Java is case-sensitive, stay away from using reserved words, even with upper or mixed case
 - However, as part of a larger identifier is OK
 - **Int** alone would be a bad idea, but **intCounter** is perfectly OK

List of Reserved Words

- See your textbook, or links in today's lecture module on Canvas, for an official list and full descriptions
 - *true*, *false*, and *null* are actually special literal data values
- We will cover **many**, but **not all**, during the semester

abstract	default	goto	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	<i>true</i>
catch	<i>false</i>	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
const	float	new	switch	while
continue	for	<i>null</i>		

Statements

- **Statements** are executable instructions in a Java program
 - Each statement performs some atomic action
 - A statement is like one complete, executable “thought”
 - Comparable to sentences in a piece of writing
- **Statements:**
 - Must be terminated with a semicolon
 - Otherwise, a compiler error results
 - Can span multiple lines if lengthy
 - Should be kept to “reasonable” length
 - Are composed using: variables, objects, operators, methods, literals, etc. (all of which we’ll get to...)

Statement Examples

- `System.out.println("Hello");`
- `int countSheep;`
- `countSheep = 50;`
- `salesTax = price * taxRate;`
- `area = 3.14159 * (2.0 * 2.0);`
- `Student fred;`
- `fred = new Student("Fred", 20);`
- `fred.setMathGrade("A-");`
- `fred.printGrades();`
- `age = fred.getAge();`
- printing
- variable declaration
- variable assignment
- calculation w/variables
- calculation w/literals
- object declaration
- object creation
- object method call w/arg
- object method call w/o args
- object method call w/return value

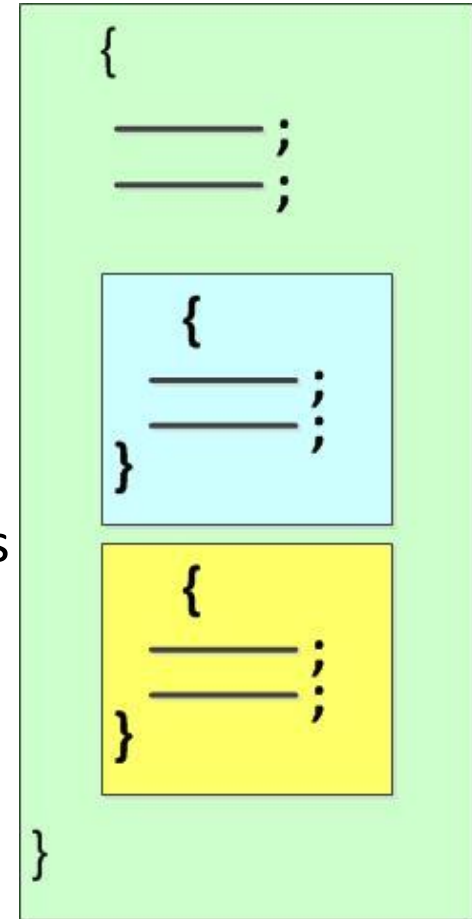
Statement Examples

- This is your Hello World program from Week 1
- Contains two statements
 - One single line, one multi-line

```
13 public class HelloWorldRL {  
14  
15     public static void main(String [] args) {  
16  
17         printGreeting("Rob", "Lapkass");  
18     }  
19  
20  
21     private static void printGreeting(String firstName, String lastName) {  
22  
23         System.out.println("Hello " + firstName +  
24             " " + lastName +  
25             ", good to have you in class");  
26     }  
27  
28  
29 }  
30  
31 }
```

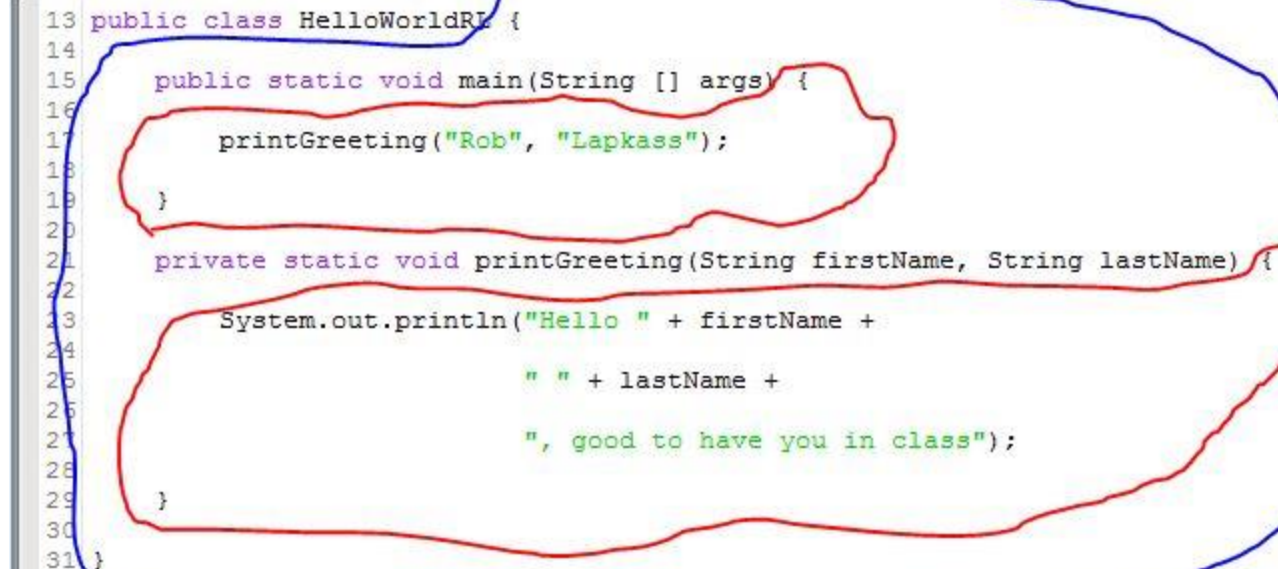
Blocks

- A **block** is a group of (related) Java statements
 - 0, 1, or more statements
 - Enclosed by a pair of curly braces: `{ ... }`
 - Can appear anywhere a single statement can appear (functionally ONE unit of code)
 - A block does not require a closing semicolon
- Blocks represent **containers** for code statements
 - They contain statements, or even other blocks
 - Blocks can be nested, but cannot overlap
- Blocks are required for class and method definitions
 - All their statements appear **inside** the curly braces
 - The curly braces are the **bounds** of that **container**
- Blocks are also used to group statements within looping and branching constructs
 - **if**, **switch**, **else**, **for**, **while**, etc. – we'll get to these...
- Block syntax with `{ ... }` braces is **common across multiple languages**:
 - Java, C, C++, PHP, JavaScript, Processing, ...



Block Examples

- This is your Hello World program from Week 1
- Note that the blocks are nested, but no overlaps



```
13 public class HelloWorldRL {  
14  
15     public static void main(String [] args) {  
16  
17         printGreeting("Rob", "Lapkass");  
18  
19     }  
20  
21     private static void printGreeting(String firstName, String lastName) {  
22  
23         System.out.println("Hello " + firstName +  
24  
25             " " + lastName +  
26  
27             ", good to have you in class");  
28  
29     }  
30  
31 }
```

The image shows a Java code snippet for a Hello World program. The code is enclosed in a blue hand-drawn outline representing the class block. Inside this, there are two red hand-drawn outlines representing method blocks. The first red outline encloses the `main` method, which calls `printGreeting`. The second red outline encloses the `printGreeting` method, which prints a greeting. The nesting is clear: the `printGreeting` method is nested within the `main` method, which is nested within the `HelloWorldRL` class.

More Block Examples

- **class**
- **method**
- **if-else()**
- **switch()**
- **while()**
- **for()**
- All these blocks are various **containers**
- All have similar block structures
- We'll get to all these...

```
public class MyClass {  
    // class content goes here...  
}
```

```
public void myMethod() {  
    // method code goes here...  
}
```

```
if (count <= 100) {  
    // if branch code goes here...  
}  
else {  
    // else branch code goes here...  
}
```

```
switch (keyPressed) {  
    case 'X':  
        // user input X handled here...  
        break;  
    case 'Q':  
        // user input Q handled here...  
        break;  
    default:  
        // default case handled here...  
        break;  
}
```

```
while (input < 1000) {  
    // actions to take while condition is true...  
}
```

```
for (int i=0; i < MAX_I_VALUE; i++) {  
    // actions to take for each i value...  
}
```

Block Conventions For This Course

- **Suggestion:** Above all, make sure to match up your open/close brace pairs
 - Debugging missing braces can be painful!
 - Suggest typing open/close brace pair **right away** when coding, then add contents
- **Requirement 1:** Make sure to use a consistent braces style at all levels
 - Aids in readability of your code
 - Software organizations may have **coding standards** for this
 - For this course, see the two allowable styles at right
 - Pick one style, and use it consistently
- **Requirement 2:** Indent everything inside braces consistently and uniformly
 - 2-4 spaces is typical (tabs not suggested)
 - Use the same indent at EACH level

- Style 1:
public class Foo {

}

- Style 2:
public class Foo
{

}

Whitespace

- **Whitespace** is the use of specific characters to visually separate individual **tokens** and statements within code
- Whitespace characters are:
 - Space, tab, newline
- Whitespace makes your code more readable
 - It gives your code “visual elbow room”
 - Use it liberally, but also reasonably, in your code
- Sometimes, whitespace is required
 - At least one whitespace character is required between keywords and identifiers
- Most of the time, whitespace is just desirable
 - Visual separation between identifiers, operands, operators
 - Blank lines between logical sections of code
 - Consistent nesting within the curly braces of a block
 - Multiple whitespace characters may be “stacked” together
- **Requirement: reasonable use of whitespace is required for all code submitted for grading in this course**

```
public static void main
```

Whitespace → Good

- Code using whitespace is simpler to follow, and easier on the eyes
 - Sometimes it is mandated by organization coding standards
 - Clean code is considered more “professional”
 - Keeps things more positive with your {clients, colleagues, supervisors, instructor}

```
13 public class HelloWorldRL {
14
15     public static void main(String [] args) {
16
17         printGreeting("Rob", "Lapkass");
18
19     }
20
21     private static void printGreeting(String firstName, String lastName) {
22
23         System.out.println("Hello " + firstName +
24
25                             " " + lastName +
26
27                             ", good to have you in class");
28
29     }
30
31 }
```

No Whitespace → Bad

- Here is Hello World with NO whitespace:

```
public class HelloWorldBad{  
public static void main(String[]args){printGreeting("Rob", "Lapkass");}  
private static void printGreeting(String firstName,String lastName){  
System.out.println("Hello "+firstName+" "+lastName+", good to have you in class");}}
```

- It still works, but which would you YOU rather maintain??
 - Which exactly are the methods?
 - Are the braces opened and closed properly?
 - What's subordinate to what here?
 - Harder to interpret the string concatenation
- This version of the same code is just visually “dense”
 - It would never pass a code review in any reputable organization
 - It would NOT be accepted for grading in this course

See [*HelloWorldBad.java*](#) in **Example Source Code**

Whitespace Conventions For This Course

- Use whitespace liberally, but also reasonably
 - Whitespace between variables and operators
 - Indent ALL code within a block consistently (2-4 spaces typically)
 - Nest blocks within each other consistently
 - Use blank lines to visually separate related sections of code
 - Use whitespace between the items of a list
- Adopt personal conventions early on, then stick with them, especially within any one source file
 - The more you write code, the more effortless this will become
 - I will likely publish a coding standard in the near future

Comments

- **Comments** are text lines interspersed among source code, to explain to a human user how that code works or why it was written a certain way (or, at all)
- Comments are **non-executable**, and completely ignored by the Java compiler
- Comments are **solely for the benefit of humans** who write and maintain code
- Comments exist to document code, but indirectly serve other purposes as well
- 2 (actually, 3) types of comments exist

Comments Across Languages

language	Java	C	C++	HTML	Java Script	PHP	VB.NET
single-line	// comment	/* comment */	// comment	<!-- comment -->	// comment	// comment	' comment
multi-line	/* * comment * comment */	/* * comment * comment */	/* * comment * comment */	<!-- comment comment -->	/* * comment * comment */	/* * comment * comment */	' comment comment '

- The format of comments is remarkably identical across languages
- Learn it once in Java, and your next languages will feel very familiar

Comment Types: Single-Line

- **Single-line comments**

- Begin with two slashes: **// comment text...**
- Anything after the slashes is a comment
- Can appear in-line after other executable code
 - Often used to document variables as they are declared/initialized
- Can be used to emulate multi-line comments

- **Examples:**

```
// This looks like a multi-line comment header block
// and it can certainly be done this way
// but it is done here using single-line comments
```

```
// This is a true single-line comment, these are for some physical constants
g = 9.81;      // acceleration due to gravity, SI [m/s^2]
```

- *In 1999, the NASA Mars Climate Orbiter crashed into Mars, creating ~\$300M of Martian space junk (google: “**spacecraft crash units mismatch**”)*
- *The root cause was eventually traced to a units mismatch between contractors, in the software for the acceleration due to gravity*
- *Could this have been averted with some simple units documentation, as shown above??*

Comment Types: Multi-Line

- **Multi-line comments**

- Contained within slash-asterisk pairs: **`/* comment text... */`**
- Anything within these delimiters is a comment
- Can also emulate a single line comment
- Can also appear in-line after executable code

```
/*  
    This is a multi-line comment header block  
    done using multi-line comments  
*/  
/* this is a one line comment using multi-line syntax */  
g = 9.81;      /* acceleration due to gravity [m/s^2] */
```

Comment Types: Javadoc

- A third type of comment is the **Javadoc comment**:

```
/**
```

```
    Javadoc comment text here
```

```
*/
```

- This type of comment is used to generate automated online documentation, using the Javadoc utility
 - The Java API documentation is the prime example of this
 - We will USE the Java API documentation later on in the course
- We probably won't cover Javadoc in this course
 - See textbook if you have more interest

Javadoc Example

- At the right is the **HTML Javadoc output** generated in ONE STEP from our Hello World example
- Try it yourself!
- Just click the Show/Generate Documentation icon on the jGRASP toolbar



Class HelloWorldRL

java.lang.Object
HelloWorldRL

public class HelloWorldRL
extends Object

Constructor Summary

Constructors

Constructor and Description

HelloWorldRL()

Method Summary

Methods

Modifier and Type	Method and Description
static void	main(String[] args)
private static void	printGreeting(String firstName, String lastName)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

HelloWorldRL

public HelloWorldRL()

Method Detail

main

public static void main(String[] args)

printGreeting

private static void printGreeting(String firstName,
String lastName)

Comment Suggestions

- A well-commented piece of code should be relatively easy for a knowledgeable, but unfamiliar, user to follow
 - Should “read like a book”!
- **Requirement: comment ALL code, as it is being written**
 - Summarize all classes, methods, etc. with preceding one-liners
 - Briefly comment the details of your code internally
- **Requirement: precede all source files with a header block**
- Comment code liberally, but reasonably
 - Don’t comment EVERY line, summarize the overall action
- Don’t comment the obvious: tell WHY, not WHAT
 - `qty = qty + 2; // add 2 to qty` (REALLY??)
 - `qty = qty + 2; // 2 free items as startup promo` (BETTER!)

Comment Usage

- All code should be documented as it is written
 - Document the thought processes of creating code
 - Aids in later code maintenance (for YOU or for others)
 - Well-documented code is easier to understand and adapt
- Comments also provide additional possibilities
 - Code development
 - Code debugging
 - Software configuration management (CM)

Comment Usage: Code Development

- Comments can be used to plan your code
 - **Pseudocode**: free-format description of code (not code itself)
 - Map out the flow of execution, without writing a line of code
 - First determine the “what”, without getting stuck in the “how”
 - Similar to “brainstorming” while writing
 - Hardest part often is the “getting started” or the outlining
 - Comments let you preserve your ideas right in the code
 - Gradually replace the comments with working code, bit by bit

```
// first, we need to prompt user for these inputs...  
// then we need to initialize a Java object which will do...  
// then we need to calculate something else...  
// then we need to display the outputs in a certain format...
```

Comment Usage: Debug

- When developing or debugging code:
 - Comments can be used to enable/disable sections of code
 - Temporarily or permanently
- Gradually restore small sections of code, until the code breaks or works

```
/*  
    add code section which doesn't work, now disabled  
*/
```

```
// I think this line of code is OK...  
// This line of code is suspect...
```


Comment Usage: CM

- Real-life applications usually consist of many source code files, which each undergo numerous revisions
- Keeping track of all the versions is a **Configuration Management (CM)** task
- Professional, production code usually always includes a **header block** in each source code file
 - Purpose of file
 - What changes were made, when, and by whom
 - Version number history
- If something breaks, you can always start tracing back in time to when it worked

For Next Time

- **Lecture Prep**
 - Text readings and lecture notes
- **Assignments**
 - See slide 2 for new/current/past due assignments