Basic Java Syntax

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/java.html

For live Java-related training,
see http://courses.coreservlets.com/
or email hall@coreservlets.com.

Taught by the author of *Core Servlets and JSP*, *More Servlets and JSP*, and this tutorial. Available at public venues, or customized versions can be held on-site at your organization.

- Courses developed and taught by Marty Hall
  - JSF 2, PrimeFaces, servlets/JSP, Ajax, jQuery, Android development, Java 7 or 8 programming, custom mix of topics
  - Courses available in any state or country. Maryland/DC area companies can also choose afternoon/evening courses.
- Courses developed and taught by coreservlets.com experts (edited by Marty)
  - Spring, Hibernate/JPA, GWT, Hadoop, HTML5, RESTful Web Services

Contact hall@coreservlets.com for details

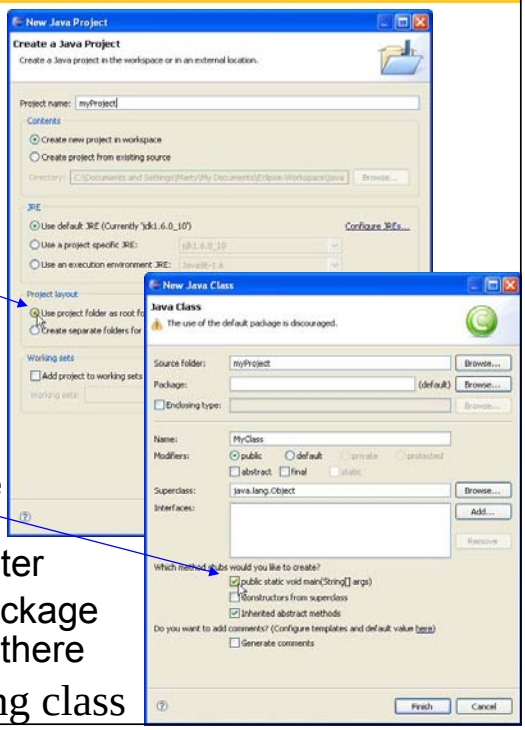
Topics in This Section

- **Basics**
  - Creating, compiling, and executing simple Java programs
- **Accessing arrays**
- **Looping**
- **Indenting Code**
- **Using if statements**
- **Comparing strings**
- **Building arrays**
  - One-step process
  - Two-step process
  - Using multidimensional arrays
- **Performing basic mathematical operations**
- **Reading command-line input**

Basics

Eclipse: Making Projects

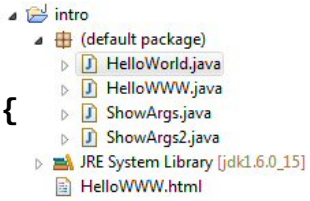
- **Creating new project**
  - File → New → Project → Java → Java Project
    - Pick any name
    - To simplify applets later, choose Sources/Classes in same folder
- **Creating new class**
  - R-click → New → Class
    - You can have Eclipse make "main" when class created, or use shortcut to insert it later
    - Eventually you will make package (subdir) first, then put class there
  - Can also copy/rename existing class



Getting Started: Syntax

- **Example**

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, world.");
  }
}
```

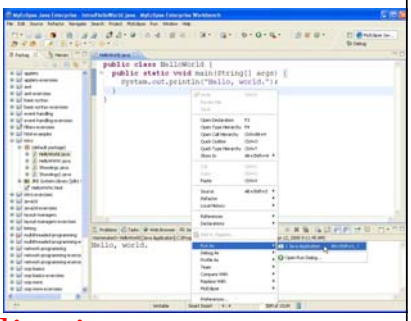


- **Details**
  - Processing starts in main
    - Eclipse can create main automatically
      - When creating class: choose main as option
      - Eclipse shortcut inside class: type "main" then hit Control-space
    - Routines usually called "methods," not "functions."
  - Printing is done with System.out.print...
    - System.out.println, System.out.print, System.out.printf
    - Eclipse shortcut: type "sysout" then hit Control-space

Getting Started: Execution

- **File: HelloWorld.java**

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, world.");
  }
}
```



- **Compiling**
  - Eclipse: just save file

```
DOS> javac HelloWorld.java
```

- **Executing**
  - Eclipse: R-click, Run As, Java Application

```
DOS> java HelloWorld
Hello, world.
```


Packages

- **Idea**
  - Packages are subdirectories used to avoid name conflicts
  - Java class must have "package subdirname;" at the top
    - But Eclipse puts this in automatically when you right-click on a package and use New → Class
- **Naming conventions**
  - Package names are in all lower case
  - Some organizations use highly nested names
    - com.companyname.projectname.projectcomponent
- **Creating packages in Eclipse**
  - R-click project, New → Package
  - Then R-click package and New → Class

HelloWorld with Packages (in src/mypackage folder)

```
package mypackage;

public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, world (using packages)");
  }
}
```

Run from Eclipse in normal manner: R-click, Run As → Java Application. Running from the command line is a pain: you must go to parent directory and do "java mypackage.HelloWorld". Run from Eclipse and it is simple to use packages.

More Basics

- **Use + for string concatenation**
- **Arrays are accessed with [ ]**
  - Array indices are zero-based
  - The argument to **`main`** is an array of strings that correspond to the command line arguments
    - args[0] returns first command-line argument
    - args[1] returns second command-line argument, etc.
    - Error if you try to access more args than were supplied
- **The `length` field**
  - Gives the number of elements in *any* array
    - Thus, **`args.length`** gives the number of command-line arguments
    - Unlike in C/C++, the name of the program is not inserted into the command-line arguments

Command-line Arguments

- **Are useful for learning and testing**
  - Command-line args are helpful for practice
  - But, programs given to end users should almost never use command-line arguments
    - They should pop up a GUI to collect input.
- **Eclipse has poor support**
  - Entering command-line args via Eclipse is more trouble than it is worth
  - So, to test with command-line args:
    - Save the file in Eclipse (causing it to be compiled)
    - Navigate to folder on desktop (not within Eclipse)
    - Open command window (Start icon, Run… → cmd)
    - Type "java *Classname* arg1 arg2 …"

Example: Command Line Args and the length Field

- **File: ShowTwoArgs.java (naïve version)**

```
public class ShowTwoArgs {
  public static void main(String[] args) {
    System.out.println("First arg: " +
                       args[0]);
    System.out.println("Second arg: " +
                       args[1]);
  }
}
```

Oops! Crashes if there are less than two command-line arguments. The code should have checked the length field, like this:

```
if (args.length > 1) {
  doThePrintStatements();
} else {
  giveAnErrorMessage();
}
```

Example (Continued)

- **Compiling (automatic on save in Eclipse)**

```
DOS> javac ShowTwoArgs.java
```

- **Manual execution**

```
DOS> java ShowTwoArgs Hello Class
First args Hello
Second arg: Class

DOS> java ShowTwoArgs
[Error message]
```

- **Eclipse execution (cumbersome)**
  - To assign command line args: R-click, Run As, Run Configurations, click on "Arguments" tab

Loops

Looping Constructs

- for/each
```
for(variable: collection) {
  body;
}
```
- for
```
for(init; continueTest; updateOp) {
  body;
}
```
- while
```
while (continueTest) {
  body;
}
```
- do
```
do {
  body;
} while (continueTest);
```


For/Each Loops

```
public static void listEntries(String[] entries) {
  for(String entry: entries) {
    System.out.println(entry);
  }
}
```

- Result
```
String[] test = {"This", "is", "a", "test"};
listEntries(test);

This
is
a
test
```

For Loops

```
public static void listNums1(int max) {
  for(int i=0; i<max; i++) {
    System.out.println("Number: " + i);
  }
}
```

- **Result**

```
listNums1(4);

Number: 0
Number: 1
Number: 2
Number: 3
```


While Loops

```
public static void listNums2(int max) {
  int i = 0;
  while (i < max) {
    System.out.println("Number: " + i);
    i++; // "++" means "add one"
  }
}
```

- **Result**

```
listNums2(5);

Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
```

Do Loops

```
public static void listNums3(int max) {
  int i = 0;
  do {
    System.out.println("Number: " + i);
    i++;
  } while (i < max);
                  // ^ Don't forget semicolon
}
```

- **Result**

```
listNums3(3);
Number: 0
Number: 1
Number: 2
```


Class Structure and Formatting

Defining Multiple Methods in Single Class

```
public class LoopTest {
  public static void main(String[] args) {
    String[] test =
      { "This", "is", "a", "test"};
    listEntries(test);
    listNums1(5);
    listNums2(6);
    listNums3(7);
  }

  public static void listEntries(String[] entries) {…}
  public static void listNums1(int max) {…}
  public static void listNums2(int max) {…}
  public static void listNums3(int max) {…}
}
```

These methods say "static" because they are called directly from "main". In the next two sections on OOP, we will explain what "static" means and why most regular methods do not use "static". But for now, just note that methods that are *directly* called by "main" must say "static".

Indentation: blocks that are nested more should be indented more

- **Yes**

```
blah;
blah;
for(...) {
  blah;
  blah;
  for(...) {
    blah;
    blah;
  }
}
```

- **No**

```
blah;
blah;
for(...) {
blah;
blah;
for(...) {
blah;
blah;
}
}
```

Indentation: blocks that are nested the same should be indented the same

- Yes

```
blah;
blah;
for(...) {
  blah;
  blah;
  for(...) {
    blah;
    blah;
  }
}
```

- No

```
blah;
 blah;
for(...) {
   blah;
  blah;
  for(...) {
    blah;
     blah;
  }
  }
```


Indentation: number of spaces and placement of braces is a matter of taste

- OK

```
blah;
blah;
for(...) {
  blah;
  blah;
  for(...) {
    blah;
    blah;
  }
}
```

- OK

```
blah;
blah;
for(...) {
    blah;
    blah;
    for(...) {
        blah;
        blah;
    }
}
```

- OK

```
blah;
blah;
for(...)
{
  blah;
  blah;
  for(...)
  {
    blah;
    blah;
  }
}
```

Conditionals and Strings

If Statements

- **Single option**

```
if (boolean-expression) {
  statement1;
  ...
  statementN;
}
```

- **Two options**

```
if (boolean-expression) {
  ...
} else {
  ...
}
```

- **Multiple options**

```
if (boolean-expression) {
  ...
} else if (boolean-expression) {
  ...
} else if (boolean-expression) {
  ...
} else {
  ...
}
```

The value inside parens must strictly be boolean, unlike C, C++, and JavaScript.

A widely accepted best practice is to use the braces even if there is only a single statement inside the if or else.

Switch Statements

- **Example**

```
int month = ...;
String monthString;
switch(month) {
  case 1: monthString = "January";  break;
  case 2: monthString = "February"; break;
  case 3: monthString = "March";    break;
  ...
  default: monthString = "Invalid month"; break;
}
```

- **Syntax is mostly like C and C++**
  - Types can be primitives, enums, and (Java 7+ only) Strings

Boolean Operators

- **==, !=**
  - Equality, inequality. In addition to comparing primitive types, == tests if two objects are identical (the same object), not just if they appear equal (have the same fields). More details when we introduce objects.
- **<, <=, >, >=**
  - Numeric less than, less than or equal to, greater than, greater than or equal to.
- **&&, ||**
  - Logical AND, OR. Both use short-circuit evaluation to more efficiently compute the results of complicated expressions.
- **!**
  - Logical negation.

Example: If Statements

```
public static int max(int n1, int n2) {
  if (n1 >= n2) {
    return(n1);
  } else {
    return(n2);
  }
}
```


Strings

- **Basics**
  - String is a real class in Java, not an array of characters as in C and C++.
  - The String class has a shortcut method to create a new object: just use double quotes
    - This differs from normal objects, where you use the **new** construct to build an object
- **Use `equals` to compare strings**
  - Never use == to test if two Strings have same characters!
- **Many useful builtin methods**
  - contains, startsWith, endsWith, indexOf, substring, split, replace, replaceAll
    - Note: can use regular expressions, not just static strings
  - toUpperCase, toLowerCase, equalsIgnoreCase

Common String Error: Comparing with ==

```
public static void main(String[] args) {
  String match = "Test";
  if (args.length == 0) {
    System.out.println("No args");
  } else if (args[0] == match) {
    System.out.println("Match");
  } else {
    System.out.println("No match");
  }
}
```

- **Prints "No match" for *all* inputs**
  - Fix:

```
if (args[0].equals(match))
```


Arrays

Building Arrays: One-Step Process

- **Declare and allocate array in one fell swoop**

```
type[] var = { val1, val2, ... , valN };
```

- **Examples:**

```
int[] values = { 10, 100, 1000 };
String[] names = {"Joe", "Jane", "Juan"};
Point[] points = { new Point(0, 0),
                   new Point(1, 2),
                   new Point(3, 4) };
```


Building Arrays: Two-Step Process

- **Step 1: allocate an array of references:**

  type[] var = new *type*[size];
  - E.g.:

    int[] primes = new int[x]; *// x is positive integer*

    String[] names = new String[someArray.length];

- **Step 2: populate the array**

primes[0] = 2;	names[0] = "Joe";
primes[1] = 3;	names[1] = "Jane";
primes[2] = 5;	names[2] = "Juan";
primes[3] = 7;	names[3] = "John";
etc.	etc.

- **If you fail to populate an entry**
  - Default value is 0 for numeric arrays
  - Default value is **`null`** for Object arrays

Two-Step Process: Examples

```
public static Circle[] makeCircles1(int numCircles) {
  Circle[] circles = new Circle[numCircles]; // Empty array of proper size
  for(int i=0; i<circles.length; i++) {
    circles[i] = new Circle(Math.random() * 10);  // Populate array
  }
  return(circles);
}

public static Circle[] makeCircles2(int numCircles) {
  Circle[] circles = new Circle[numCircles]; // Empty array of proper size
  for(int i=0; i<circles.length; i++) {
    circles[i].setRadius(Math.random() * 10); // NullPointerException
  }
  return(circles);
}

public static Circle[] makeCircles3(int numCircles) {
  Circle[] circles = new Circle[numCircles];
  for(Circle c: circles) {
    c = new Circle(Math.random() * 10);; // Fails to store c in array
  }
  return(circles); // Array still contains only null pointers
}
```

This approach is correct!

This approach fails: crashes with NullPointerException because circles[i] is null.

This approach fails: array is still empty after the loop.

Array Performance Problems

- **For very large arrays, undue paging can occur**
  - Array of references (pointers) allocated first
  - Individual objects allocated next
  - Thus, for very large arrays of objects, reference and object can be on different pages, resulting in swapping for each array reference
  - Example

```
String[] names = new String[10000000];
for(int i=0; i<names.length; i++) {
  names[i] = getNameFromSomewhere();
}
```

- **Problem does not occur with arrays of primitives**
  - I.e., with arrays of **`int`**, **`double`**, and other types that start with lowercase letter
  - Because system stores values directly in arrays, rather than storing references (pointers) to the objects

Multidimensional Arrays

- **Multidimensional arrays**
  - Implemented as arrays of arrays

```
int[][] twoD = new int[64][32];

String[][] cats = {{ "Caesar",  "blue-point" },
                   { "Heather", "seal-point" },
                   { "Ted",     "red-point"  }};
```

- **Note:**
  - Number of elements in each row need not be equal

```
int[][] irregular = { { 1 },
                      { 2, 3, 4},
                      { 5 },
                      { 6, 7 } };
```


TriangleArray: Example

```
public class TriangleArray {
  public static void main(String[] args) {

    int[][] triangle = new int[10][];

    for(int i=0; i<triangle.length; i++) {
      triangle[i] = new int[i+1];
    }

    for (int i=0; i<triangle.length; i++) {
      for(int j=0; j<triangle[i].length; j++) {
        System.out.print(triangle[i][j]);
      }
      System.out.println();
    }
  }
}
```

TriangleArray: Result

```
> java TriangleArray

0
00
000
0000
00000
000000
0000000
00000000
000000000
0000000000
```


Math and Input

Basic Mathematical Routines

- **Very simplest routines use builtin operators**
  - +, -, *, /, ^, %
  - Be careful with / on **`int`** and **`long`** variables
- **Static methods in the Math class**
  - So you call Math.cos(...), Math.random(), etc.
    - Most operate on double precision floating point numbers
  - Simple operations: Math.pow(), etc.
    - pow (x^y), sqrt ($\sqrt{x}$), cbrt, exp (e^x), log ($\log_e$), log10
  - Trig functions: Math.sin(), etc.
    - sin, cos, tan, asin, acos, atan
      - Args are in radians, not degrees, (see toDegrees and toRadians)
  - Rounding and comparison: Math.round(), etc.
    - round/rint, floor, ceiling, abs, min, max
  - Random numbers: Math.random()
    - random (Math.random() returns from 0 inclusive to 1 exclusive).
    - See Random class for more control over randomization.

More Mathematical Routines

- **Special constants**
  - Double.POSITIVE_INFINITY
  - Double.NEGATIVE_INFINITY
  - Double.NAN
  - Double.MAX_VALUE
  - Double.MIN_VALUE
- **Unlimited precision libraries**
  - BigInteger, BigDecimal
    - Contain the basic operations, plus BigInteger has isPrime

Reading Simple Input

- **For simple testing, use standard input**
  - If you want strings, just use args[0], args[1], as before
    - To avoid errors, check args.length first
  - Convert if you want numbers. Two main options:
    - Use Scanner class
      - Note that you need import statement. See next slide!

```
Scanner inputScanner = new Scanner(System.in);
int i = inputScanner.nextInt();
double d = inputScanner.nextDouble();
```

    - Convert explicitly (Integer.parseInt, Double.parseDouble)

```
String seven = "7";
int i = Integer.parseInt(seven);
```

- **In real applications, use a GUI**
  - Collect input with textfields, sliders, combo boxes, etc.
    - Convert to numeric types with Integer.parseInt, Double.parseDouble, etc.

Example: Printing Random Numbers

```
import java.util.*;

public class RandomNums {
  public static void main(String[] args) {
    System.out.print("How many random nums? ");
    Scanner inputScanner = new Scanner(System.in);
    int n = inputScanner.nextInt();
    for(int i=0; i<n; i++) {
      System.out.println("Random num " + i +
                         " is " + Math.random());
    }
    inputScanner.close();
  }
}
How many random nums? 5
Random num 0 is 0.22686369670835704
Random num 1 is 0.0783768527137797
Random num 2 is 0.17918121951887145
Random num 3 is 0.3441924454634313
Random num 4 is 0.6131053203170818
```

Wrap-Up

Summary

- **Basics**
  - Loops, conditional statements, and array access is similar to C/C++
    - But new "for each" loop: for(String s: someStrings) { … }
  - Indent your code for readability
  - `String` is a real class in Java
    - Use **`equals`**, not ==, to compare strings
- **Allocate arrays in one step or in two steps**
  - If two steps, loop down array and supply values
- **Use Math.*blah*() for simple math operations**
- **Simple input from command window**
  - Use command line for strings supplied at program startup
  - Use Scanner to read values after prompts
    - Neither is very important for most real-life applications

Questions?

JSF 2, PrimeFaces, Java 7 or 8, Ajax, jQuery, Hadoop, RESTful Web Services, Android, HTML5, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training. Also see the Java 8 tutorial and general Java programming tutorial.