Java Programming: Syntax and Utilities II

Originals of slides and source code for examples: http://courses.coreservlets.com/Course-Materials/java.html
Also see Java 8 tutorial: http://www.coreservlets.com/java-8-tutorial/ and many other Java EE tutorials: http://www.coreservlets.com/
Customized Java training courses (onsite or at public venues): http://courses.coreservlets.com/java-training.html

© 2015 Marty Hall

For customized Java-related training at your organization, email hall@coreservlets.com

Marty is also available for consulting and development support

Taught by lead author of *Core Servlets & JSP*, co-author of *Core JSF* (4th Ed), & 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 2.2, PrimeFaces, servlets/JSP, Ajax, JavaScript, jQuery, Android, Java 7 or 8 programming, GWT, 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 MVC, Core Spring, Hibernate/JPA, Hadoop, HTML5, RESTful Web Services

Contact hall@coreservlets.com for details

Topics in This Section

- **Data structures**
  - ArrayList
  - LinkedList
  - HashMap
- **Generic types**
  - Using classes and methods that already support generics
  - Defining your own classes and methods that support generics
- **The Arrays utility class**
- **printf**
- **varargs**
- **String vs. StringBuilder**

Lists

Lists

- **Problem**
  - You want to make an ordered list of objects. But, even after you get the first few elements, you don't know how many more you will have.
    - Thus, you can't use an array, since the size of arrays must be known at the time that you allocate it. (Although Java arrays are better than C++ arrays since the size does not need to be a compile-time constant)
- **Solution**
  - Use ArrayList or LinkedList: they stretch as you add elements to them
    - The two options give the same results for the same operations, but differ in performance

ArrayList and LinkedList: Basic Methods

- **Create empty list**
  - `List<Type> var = new ArrayList<>()` or
    `List<Type> var = new LinkedList<>()`
    - We will explain shortly why we say List instead of ArrayList for the variable type
- **Add entry to end**
  - `add(value)` (adds to end) or
    `add(index, value)`
- **Retrieve *n*th element**
  - `get(index)`

ArrayList Example

```
import java.util.*;

public class ListTest2 {
  public static void main(String[] args) {
    List<String> entries = new ArrayList<>();
    double d;
    while((d = Math.random()) > 0.1) {
      entries.add("Value: " + d);
    }
    for(String entry: entries) {
      System.out.println(entry);
    }
  }
}
```

Before Java 7, you had to repeat the type here, using new ArrayList<String>() instead of new ArrayList<>()

This tells Java your list will contain only strings. More on this in a few slides in section on generics.

ArrayList Example: Output

```
> java ListTest2
Value: 0.6374760850618444
Value: 0.9159907384916878
Value: 0.8093728146584014
Value: 0.7177611068808302
Value: 0.9751541794430284
Value: 0.2655587762679209
Value: 0.3135791999033012
Value: 0.44624152771013836
Value: 0.7585420756498766
```

ArrayList and LinkedList: More Methods

- **Check if element exists in list**
  - `contains(element)`
- **Remove element**
  - `remove(index)` or `remove(element)`
- **Find the number of elements**
  - `size()`
- **Print a list**
  - `System.out.println(someList);`
    - Automatically uses square brackets and puts commas between elements

More List Methods: Code

```
public class MoreWithLists {
  public static void main(String[] args) {
    List<String> names = new ArrayList<>();
    // or List<String> names = new LinkedList<>();
    names.add("Marty");
    names.add("Cay");
    names.add("David");
    int numEntries = names.size();
    System.out.println("Num entries: " + numEntries);
    String secondEntry = names.get(1);
    System.out.println("2nd entry: " + secondEntry);
    boolean containsJoe = names.contains("Joe");
    System.out.println("Contains Joe? " + containsJoe);
    names.remove("Marty");
    numEntries = names.size();
    System.out.println("Num entries: " + numEntries);
  }
}
```

More List Methods: Output

- **Output when using ArrayList**

```
Num entries: 3
2nd entry: Cay
Contains Joe? false
Num entries: 2
```

- **Output when using LinkedList (same!)**

```
Num entries: 3
2nd entry: Cay
Contains Joe? false
Num entries: 2
```


List Operation Performance

Comparing ArrayList and LinkedList Performance

	Array with Copying (`ArrayList`)	List of Pointers (`LinkedList`)
Insert at beginning	O(N)	O(1)
Insert at end	O(1) if space O(N) if not O(1) amortized time	O(1)
Access Nth Element	O(1)	O(N)

Hang on to your hats! We will walk carefully through what this all means in the lecture. But the whole reason that Java supplies both LinkedList and ArrayList is because of the different performance characteristics, so understanding this slide is the key to deciding which class to use in real life.

Using Generic Types

Using Generics

- **General steps**
  - Find a data structure that accepts Object(s)
    - ArrayList, LinkedList, HashMap, HashSet, Stack
  - Declare the data structure with the type(s) in angle brackets immediately after class name
    - List<String> names = new ArrayList<>();
    - Map<String,Person> employees = new HashMap<>();
  - Insert objects of the appropriate type
    - names.add("Some String");
    - employees.put(person.getEmployeeId(), person);
  - No typecast required on removal
    - String firstName = names.get(0);
    - Person p1 = employees.get("a1234");

ArrayList Example: Explicit Typecasts (Java 1.4 and Earlier)

```
import java.util.*;

public class ListTest1 {
  public static void main(String[] args) {
    List entries = new ArrayList();
    double d;
    while((d = Math.random()) > 0.1) {
      entries.add("Value: " + d);
    }
    String entry;
    for(int i=0; i<entries.size(); i++) {
      entry = (String)entries.get(i);
      System.out.println(entry);
    }
  }
}
```

ArrayList Example: Generics (Java 7 and Later)

```
import java.util.*;

public class ListTest2 {
  public static void main(String[] args) {
    List<String> entries = new ArrayList<>();
    double d;
    while((d = Math.random()) > 0.1) {
      entries.add("Value: " + d);
    }
    for(String entry: entries) {
      System.out.println(entry);
    }
  }
}
```

This tells Java your list will contain only strings. Java will check at *compile* time that all additions to the list are Strings. You can then use the simpler looping construct because Java knows ahead of time that all entries are Strings.

ArrayList Example: No Diamond Operator (Java 5 and 6)

```
import java.util.*;

public class ListTest3 {
  public static void main(String[] args) {
    List<String> entries = new ArrayList<String>();
    double d;
    while((d = Math.random()) > 0.1) {
      entries.add("Value: " + d);
    }
    for(String entry: entries) {
      System.out.println(entry);
    }
  }
}
```

Before Java 7, you had to use <String> instead of <> here. In Java 7 and later, the compiler will do type inferencing. But you still need List<String> in the variable declaration.

Autoboxing: Idea

- **Objects vs. Primitives**
  - Primitives are stored directly. Objects use pointers (references). Primitives start with lowercase letters (int, double, boolean, etc.). Object classes start with uppercase letters (Integer, Double, Boolean, String, etc.).
- **List, Map, etc. expect Object**
  - So you cannot insert int, double, and other primitives. You must use Integer, Double, etc.
- **Java (Java 5+) converts automatically**
  - You can assign int to Integer or Integer to int

Autoboxing: Old (Java 1.4) vs. New (Java 5+)

Old (Java 1.4 and Earlier)	New (Java 5 and Later)

`List nums = new ArrayList();`
`int i = someCalculation();`
`nums.add(new Integer(i));`
`...`
`Integer val =`
`  (Integer)nums.get(someIndex);`
`int num = val.intVal() + 1;`
`nums.add(new Integer(num));`
	`List<Integer> nums =`
	`  new ArrayList<>();`
	`nums.add(someCalculation());`
	`...`
	`int num =`
	`  nums.get(someIndex);`
	`nums.add(num + 1);`

Generic types were a win in every way: shorter and more typesafe code, identical performance.
Autoboxing is a tradeoff: more convenient but slightly worse performance in some situations.

Maps
(Also called "Lookup Tables" or "Associative Arrays")

HashMap: Overview

- **Idea**
  - HashMap provides simple lookup table
- **Usage**
  - Use "put" to store data

```
Map<String,Person> employees =
  new HashMap<>();
Person p1 =
  new Person("a1234", "Larry", "Ellison");
employees.put(p1.getEmployeeId(), p1);
```

The table keys will be Strings; the associated values will be Persons.

  - Use "get" to retrieve data

```
Person p = employees.get("a1234");
```

    - Returns null if no match

HashMap: Performance

- **Inserting a value associated with a key**
  - Independent of the number of entries in the table, i.e., O(1).
- **Finding the value associated with a key**
  - Independent of the number of entries in the table, i.e., O(1).
- **Finding the key associated with a value**
  - Requires you to look at every entry, i.e., O(N)
- **Assumptions**
  - This performance makes some assumptions about the hash function, which is usually a safe assumption
    - But Java has other Map types with different performance characteristics and features

HashMap Example: Finding State Abbreviations Based on State Names

```
public class StateMap {
  private Map<String,String> stateMap;

  public StateMap() {
    stateMap = new HashMap<>();
    for(String[] state: stateArray) {
      stateMap.put(state[0], state[1]);
    }
  }
  public Map<String,String> getStateMap() {
    return(stateMap);
  }

  public String[][] getStateArray() {
    return(stateArray);
  }
  private String[][] stateArray =
    { { "Alabama", "AL" },
      { "Alaska", "AK" },
      { "Arizona", "AZ" }, ….
}
```

HashMap Example: Finding State Abbreviations Based on State Names

```
public class MapTest {
  public static void main(String[] args) {
    StateMap states = new StateMap();
    Map<String,String> stateAbbreviationTable =
      states.getStateMap();
    Scanner inputScanner = new Scanner(System.in);
    System.out.println("Enter state names. " +
                       "Hit RETURN to quit");
    String stateName;
    String abbreviation;
```


HashMap Example: Finding State Abbreviations Based on State Names

```
    while(true) {
      System.out.print("State: ");
      stateName = inputScanner.nextLine();
      if (stateName.equals("")) {
        System.out.println("Come again soon.");
        break;
      }
      abbreviation =
        stateAbbreviationTable.get(stateName);
      if (abbreviation == null) {
        abbreviation = "Unknown";
      }
      System.out.println(abbreviation);
    }
  }
}
```

A Few Other Simple Map Methods

- **getOrDefault(key, defaultValue)**
  - If there is value already associated with the key, returns it. Otherwise returns the default value.
- **keySet()**
  - Returns a Set of the keys. A Set is like a Map with boolean values (i.e., does key exist or not?)
- **values()**
  - Returns a Collection of the values. You can loop over a Collection just as with a List, but you cannot access entries by index.
- **size()**
  - Returns the number of entries.

A Few Advanced Map Methods

- **forEach(function)**
  - For each entry, passes key and value to the bi-function
    - `map.forEach((k,v) -> System.out.printf("(%s,%s)%n", k, v);`
- **computeIfAbsent(key, function)**
  - If there is no value associated with the key, pass the key to the function and store the result in the Map
- **merge(key, value, function)**
  - If there is no value associated with the key, inserts the value. Otherwise replaces existing value with the function applied to old and new values
    - `map.merge(key, message, String::concat)`
- **Notes**
  - Lambda functions (like the argument to forEach) and method references (String::concat) covered in later sections
  - printf (used in forEach example) covered later in this section

Building Genericized Methods and Classes

Generic Classes and Methods: Overview

- **Using existing genericized methods and classes**
  - Even beginning Java programmers need to know how to *use* classes that support generics:
    - List<Employee> workers = …;
    - Map<String, Employee> workerDatabase = …;
- **Creating genericized methods and classes**
  - Intermediate Java developers should also to be able to *define* classes or methods that support generics.

Generic Classes and Methods: Syntax Overview

- **Using <TypeVariable>**
  - If you put variables in angle brackets in the class or method definition, this tells Java that future uses of those variables refer to types, not instances.
  - It is conventional to use short names in upper case, such as T, R (input type, result type) or T1, T2 (type1, type2), or E (element)
- **Examples**

```
public class ArrayList<E> … {
  …
}
public static <T> T randomElement(T[] array) {
  …
}
```


Generic Classes and Methods: Syntax Details

- **Declaring classes that support generics**
  - public class SomeClass<T> { … }
    - Methods in the class can now refer to T for arguments and return values
      - public T getSomeValue(int index) { … }
    - Java will figure out the type of T by your declaration
      - SomeClass<String> blah = new SomeClass<>();
- **Declaring methods that support generics**
  - public static <T> T myMethod(T[] array, …) { … }
    - Java will figure out the type of T by the method call
      - Person[] people = …;
      - myMethod(people, …);

Example: Generic Method

```
public class RandomUtils {
  private static Random r = new Random();

  public static int randomInt(int range) {
    return(r.nextInt(range));
  }

  public static int randomIndex(Object[] array) {
    return(randomInt(array.length));
  }

  public static <T> T randomElement(T[] array) {
    return(array[randomIndex(array)]);
  }
}
```

In rest of method, T refers to a type

This says that the method takes in an array of T's and returns a T. For example, if you pass in an array of Strings, you get out a String; if you pass in an array of Employees, you get out an Employee. No typecasts required in any of the cases.

Using RandomUtils

- **Examples**

```
String[] names = { "Joe", "John", "Jane" };
String name = RandomUtils.randomElement(names);
Color[] colors = { Color.RED, Color.GREEN, Color.BLUE };
Color color = RandomUtils.randomElement(colors);
Person[] people =
    { new Person("Larry", "Page"), new Person("Larry", "Ellison"),
      new Person("Larry", "Bird"), new Person("Larry", "King") };
Person person = RandomUtils.randomElement(people);
Integer[] nums = { 1, 2, 3, 4 };    // Integer[], not int[]
int num = RandomUtils.randomElement(nums);
```

- **Points**
  - No typecast required to convert to String, Color, Person, Integer
  - Autoboxing lets you assign entry from Integer[] to an int, but array must be Integer[] not int[], since generics work only with Object types, not primitive types

Example: Generic Class (Simplified)

```
public class ArrayList<E> {

  public E get(int index) { ... }

  ...
}
```

In rest of class, E refers to a type

This says that get returns an E. So, if you created ArrayList<Employee>, get returns an Employee. No typecast required.

This is a highly simplified version of the real java.util.ArrayList class.
That class implements multiple interfaces, and the generic support comes from the interfaces.

The Arrays Class

Arrays.asList

- **Idea**
  - Takes an array or a number of individual entries and produces a List
- **Example: passing in individual entries**

```
List<String> wordList1 =
  Arrays.asList("word1", "word2", ...);
```

- **Example: passing in array**

```
String[] wordArray = { "word1", "word2", ...};
List<String> wordList2 = Arrays.asList(wordArray);
```


Other Static Methods in Arrays Class

- **Arrays.sort**
  - Sorts an array based on a comparator. Details on Comparator in first section on lambdas.
    - Arrays.sort(wordArray, someComparator);
- **More**
  - Arrays.binarySearch, Arrays.copyOf, Arrays.deepEquals, Arrays.fill, Arrays.parallelSort, Arrays.setAll, Arrays.stream

printf

Formatted Output: printf

- **Takes a variable number of arguments**
  - System.out.printf("Formatting String", arg1, arg2, …);
- **The formatting string has %_ placeholders**
  - %s for anything to be displayed as string, %f for floating point numbers, %d for whole numbers, etc.
    - System.out.printf("Value1: %s, value2: %s%n", val1, val2);
- **%n means newline**
  - Both printouts on same line
    - System.out.printf("blah");
    - Sytem.out.printf("blah");
  - Two printouts on different lines
    - System.out.printf("blah%n");
    - Sytem.out.printf("blah%n");

Formatted Output: printf

- **Advantages**
  - Lets you insert values into output without much clumsier String concatenation.
  - Lets you control the width of results so things line up
  - Lets you control the number of digits after the decimal point in numbers, for consistent-looking output
  - Applies to any PrintWriter or PrintStream, not just System.out
    - In File IO and Networking sections, we will make our own PrintWriter, then use ourWriter.printf
- **Very similar to C/C++ printf function**
  - If you know printf in C/C++, you can probably use Java's printf immediately without reading any documentation
    - Although some additions in time formatting and locales

Simple Example: printf vs. println

- **Example**

```
public static void printSomeStrings() {
  String firstName = "John";
  String lastName = "Doe";
  int numPets = 7;
  String petType = "chickens";
  System.out.printf("%s %s has %s %s.%n",
                    firstName, lastName, numPets, petType);
  System.out.println(firstName + " " + lastName +
                     " has " + numPets + " " +
                     petType + ".");
}
```

- **Result:**

```
John Doe has 7 chickens.
John Doe has 7 chickens.
```

Controlling Formatting

- **Different flags**
  - %s for strings, %f for floats/doubles, %t for dates, etc.
    - Unlike in C/C++, you can use %s for *any* type (even numbers)
- **Various extra entries can be inserted**
  - To control width, number of digits, commas, justification, type of date format, and more
- **Details**
  - printf uses mini-language
    - Complete coverage would take an entire lecture
    - However, basic usage is straightforward
  - For complete coverage, see
    http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax

Printf Formatting Options

	Stands For	Options	Example
%s	String. Can output any data type. If arg is Object, toString is called.	%*width*s Gives min num of chars. Spaces added to left if needed.	printf("%8s", "Hi") outputs " Hi"
%d	Decimal. Outputs whole number in base 10. Also %x and %o for hex and octal.	%*width*d %,*width*d Gives min width; inserts commas.	printf("%,9d", 1234) outputs " 1,234"
%f	Floating point. Lets you line up decimal point and control precision.	%*width.precision*f %,*width.precision*f width includes comma and decimal point.	printf("%6.2f", Math.PI) outputs " 3.14"
%t*x*	Time (or date). %tA for day, %tB for month, %tY for year, and many more.	Date now = new Date(); printf("%tA, %tB ,%tY", now, now, now) outputs "Thursday, November 17, 2005"	
%n	Outputs OS-specific end of line (linefeed on Linux, CR/LF pair on Windows)		

Printf Example: Controlling Width and Precision

```
public static void printSomeSalaries() {
  CEO[] softwareCEOs =
    { new CEO("Steve Jobs", 3.1234),
      new CEO("Scott McNealy", 45.5678),
      new CEO("Jeff Bezos", 567.982323),
      new CEO("Larry Ellison", 6789.0),
       new CEO("Bill Gates", 78901234567890.12)};
  System.out.println("SALARIES:");
  for(CEO ceo: softwareCEOs) {
    System.out.printf("%15s: $%,8.2f%n",
            ceo.getName(), ceo.getSalary());
}}
SALARIES:
     Steve Jobs: $    3.12
  Scott McNealy: $   45.57
     Jeff Bezos: $  567.98
  Larry Ellison: $6,789.00
     Bill Gates: $78,901,234,567,890.12
```


Printf Example: Controlling Width and Precision

```
public class CEO {
  private String name;
  private double salary; // In billions

  public CEO(String name, double salary) {
    this.name = name;
    this.salary = salary;
  }

  public String getName() { return(name); }

  public double getSalary() { return(salary); }
}
```

printf vs. String.format

- **printf**
  - *Outputs* a formatted String
- **String.format**
  - *Returns* a formatted String
- **Equivalent code**

```
System.out.printf("Blah %s", 7);

String s = String.format("Blah %s", 7);
System.out.print(s);
```

- **Note**
  - For consistency with String.format, System.out.format is synonymous with System.out.printf

Common printf Errors

- **Using + instead of , between arguments**
  - printf uses varargs, println uses a single String
- **Forgetting to use %n**
  - printf does not add a newline automatically, println does
- **Forgetting how to insert a literal %**
  - You use %%, not \%

Varargs

Variable-Length Arguments

- **printf takes any number of arguments**
  - You could use overloading to define a few versions of printf with different argument lengths, but it takes *any* number of arguments
- **To do this yourself, use "*type* ... *variable*"**
  - variable becomes an array of given type
  - Only legal for final argument of method
  - Examples
    - public void printf(String format, Object ... arguments)
    - public int max(int ... numbers)
      - Can call max(1, 2, 3, 4, 5, 6) or max(someArrayOfInts)
- **Use sparingly**
  - You usually know how many arguments are possible

Varargs: Example

```
public class MathUtils {
  public static int min(int ... numbers) {
    int minimum = Integer.MAX_VALUE;
    for(int number: numbers) {
      if (number < minimum) {
        minimum = number;
      }
    }
    return(minimum);
  }

  public static void main(String[] args) {
    System.out.printf("Min of 2 nums: %d.%n",
                      min(2,1));
    System.out.printf("Min of 7 nums: %d.%n",
                      min(2,4,6,8,1,2,3));
  }
}
```


StringBuilder

String vs. StringBuilder

- **Strings are immutable (unmodifiable)**
  - Thus what appears to be String concatenation really involves copying the string on the left (oldString below)
    - String newString = oldString + "some extra stuff"
  - Never do String concatenation inside a loop that could be very long (i.e., more than about 100)
- **StringBuilder is mutable**
  - Build a StringBuilder from a String with new StringBuilder(someString)
  - Call append to append data to the end
  - Call toString to turn back into a string
  - Other methods: insert, replace, substring, indexOf, reverse

Performance Comparison: Using String

- **Code**

```
public static String padChars1(int n, String orig) {
  String result = "";
  for(int i=0; i<n; i++) {
    result = result + orig;
  }
  return(result);
}
```

- **Usage**
  - padChars(5, "x") returns "xxxxx" for this and for the upcoming StringBuilder version
- **Performance**
  - $O(N^2)$. Why?

Performance Comparison: Using StringBuilder

- **Code**

```
public static String padChars2(int n, String orig) {
  StringBuilder result = new StringBuilder("");
  for(int i=0; i<n; i++) {
    result = result.append(orig);
  }
  return(result.toString());
}
```

- **Performance**
  - O(N)

Wrap-Up

Summary

- **Stretchable lists: ArrayList and LinkedList**
  - Different performance characteristics, same methods (mostly) and output
  - Declare variables to be of type List, not ArrayList or LinkedList
- **Generics let you avoid tedious typecasts**
  - List<SomeType> entries = new ArrayList<>();
    - Java 5 and 6: List<SomeType> entries = new ArrayList<SomeType>();
- **HashMap supports large and fast lookup tables**
  - someTable.put(key, value);
  - SomeType value = someTable.get(key);
- **More**
  - Arrays class has many useful static methods for arrays and Lists
  - Java stole printf from C. Yay!
  - Varargs provide for flexible argument lists
  - Use StringBuilder for string concatenation in loops

Questions?

More info:
http://courses.coreservlets.com/Course-Materials/java.html – General Java programming tutorial
http://www.coreservlets.com/java-8-tutorial/ – Java 8 tutorial
http://courses.coreservlets.com/java-training.html – Customized Java training courses, at public venues or onsite at *your* organization
http://coreservlets.com/ – 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
Many additional free tutorials at coreservlets.com (JSF, Android, Ajax, Hadoop, and lots more)