Generic Types, printf, and Miscellaneous Java Utilities

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

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

coreservlets.com – custom onsite training

For customized training related to Java or JavaScript, please email hall@coreservlets.com

Marty is also available for consulting and development support

The instructor is author of several popular Java EE books, two of the most popular Safari videos on Java and JavaScript, and this tutorial.

Courses available at public venues, or
custom versions can be held on-site at your organization.

- **Courses developed and taught by Marty Hall**
  - JSF 2.3, PrimeFaces, Java programming (using Java 8, for those new to Java), Java 8 (for Java 7 programmers), JavaScript, jQuery, Angular 2, Ext JS, Spring Framework, Spring MVC, Android, GWT, custom mix of topics.
  - Java 9 training coming soon.
  - Courses available in any state or country.
  - Maryland/DC companies can also choose afternoon/evening courses.
- **Courses developed and taught by coreservlets.com experts (edited by Marty)**
  - Hadoop, Spark, Hibernate/JPA, HTML5, RESTful Web Services

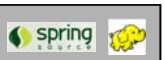
Contact hall@coreservlets.com for details

Topics in This Section

- **Supporting generic types in your own code**
  - Idea
  - Methods
  - Classes or interfaces
- **printf**
- **varargs**
- **String vs. StringBuilder**





Building Generic Methods and Classes: Overview

For additional materials, please see http://www.coreservlets.com/. The Java tutorial section contains complete source code for all examples in this tutorial series, plus exercises and exercise solutions for each topic.

Using Existing Generic Methods and Classes

- **Basic capability**
  - Even beginning Java programmers need to know how to *use* classes that support generics
  - You cannot properly use Lists, Maps, Sets, etc. without this
  - Covered in earlier section

```
List<Employee> workers = ...;
workers.add(new Employee(...)); // Type checked @ compile time
Employee e = workers.get(someIndex); // Return is Employee

Map<String,Employee> workerTable = ...;
workerTable.put(someId, someEmployee);
Employee employeeWithId = workerTable.get(someId);
```


Creating Your Own Generic Methods and Classes

- **Intermediate capability**
  - Intermediate Java developers should also to be able to *define* classes or methods that support generics
  - In Java 7 and earlier, being able to do this was mostly reserved for advanced developers, but it is done much more commonly in Java 8
    - Because lambda functions and generic types work together for same goal: to make code more reusable

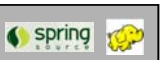
```
public interface Map<K,V> { ... }
public static <T> T lastElement(List<T> elements) { ... }
```

Generic Classes and Methods: Syntax Overview

- **Using <TypeVariable>**
  - If you put variables in angle brackets in the class or method definition, it tells Java that uses of those variables refer to types, not to values
  - 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 type)
- **Examples**

```
public class ArrayList<E> ... {
  ...
}

public static <T> T randomElement(T[] array) {
  ...
}
```



Generic Methods

Generic Classes and Methods: Syntax Details

- **Declaring methods that support generics**

```
public static <T> T best(List<T> entries, ...) { ... }
```

  - This says that the best method takes a List of T's and returns a T. The <T> at the beginning means T is not a real type, but a type that Java will figure out from the method call.

- **Java will figure out the type of T by looking at parameters to the method call**

```
List<Person> people = ...;
Person bestPerson = Utils.best(people, ...);
List<Car> cars = ...;
Car bestCar = Utils.best(cars, ...);
```


Partial Example: randomElement

```
public class RandomUtils {
  ...

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

- In rest of method, T refers to a type.
- Java will figure out what type T is by looking at the parameters of the method call.
- Even if there is an existing class actually called T, it is irrelevant here.

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.

Complete Example: randomElement

```
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)]);
  }
}
```


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 passed to randomElement must be Integer[] not int[], since generics work only with Object types, not primitive types





Generic Classes or Interfaces

Generic Classes and Methods: Syntax Details

- **Declaring classes or interfaces that support generics**

```
public class SomeClass<T> { ... }
```

- **Methods in the class can now refer to T both for arguments and for return values**

```
public T getSomeValue(int index) { ... }
```

- **Java will figure out the type of T by your declaration**

```
SomeClass<Person> blah = new SomeClass<>();
```

Example: Generic Class (Simplified)

```
public class ArrayList<E> {

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

  public boolean add(E element) { ... }

  ...
}
```

In rest of class, E does not refer to an existing type. Instead, it refers to whatever type was defined when you created the list. E.g., if you did ArrayList<String> words = ...; then E refers to String.

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

This says that add takes an E as a parameter. So, if you created ArrayList<Circle>, add can take only a Circle.

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.







printf – Formatted Output

printf: Quick Overview

- **Values replace %s placeholders (%n means end-of-line)**

```
String name = "Jane";
double num = 1234.567;
System.out.printf("%s's number is %s.%n", name, num);
```

```
Jane's number is 1234.567.
```

- **Use %*n*s to control spacing**

```
Name (8 chars): '    Jane'.
```

```
System.out.printf("Name (8 chars): '%8s'.%n", name);
```

- **For numbers, use %f to control decimal places and more**

```
Num (rounded to 2 places): 1234.57.
```

```
System.out.printf("Num (rounded to 2 places): %.2f.%n", num);
```


printf: A Few Details

- **Takes a variable number of arguments**

```
System.out.printf("Formatting String", arg1, arg2, ...);
```

  - First a string, then one extra arg for each %_ placeholder (not counting %n)

- **The formatting string has %_ placeholders**
  - %s for anything to be treated as string, %f for floating point numbers, %d for whole numbers, %t for times, etc.

```
System.out.printf("Value1: %s, value2: %s%n", val1, val2);
```

- **%n means newline**
  - Both printouts on same line

```
System.out.printf("blah");
System.out.printf("blah");
```

  - Two printouts on different lines

```
System.out.printf("blah%n");
System.out.printf("blah%n");
```

Motivation

- **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 to System.out
    - In File IO and Networking sections, we will make our own PrintWriter, then use ourWriter.printf

Java printf vs. C++ printf

- **They are very similar**
  - If you know printf in C/C++, you can probably use Java's printf immediately without reading any documentation
- **Key differences in Java version**
  - %s can be used for any type, even numbers. You only need number-specific placeholders like %f and %d when you are doing number-specific formatting like controlling digits after decimal point, inserting commas, etc.
  - You use %n for newlines. You can also use \n as in C++, but %n is slightly better
    - It inserts the newline of the current OS (e.g., LF on Unix, CR/LF pair on Windows)
  - There are a few new options for times and locales
    - But these are not important to learn at the beginning

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 name, %td for day number, %tB for month, %tY for year, and many more.	Date now = new Date(); System.out.printf("%tA, %tB %td, %tY", now, now, now, now); outputs "Tuesday, April 12, 2016"	
%n	OS-specific end of line (linefeed on Linux, CR/LF pair on Windows)		

Most Common Flag: %s

- **Overview**
  - Treat entry as a String
  - Any type is legal, not just Strings
- **Usage and options**
  - **%s** – prints the same way as System.out.print would: for doubles prints all the digits, and for objects prints the exact result of toString
    - `System.out.printf("%s", valueOfAnyType);`
  - **%*n*s** – prints the value the same as above, but if the output is less than *n* characters long, pads with spaces on the left so that total output is exactly *n* characters
    - `System.out.printf("%15f", valueOfAnyType);`
- **Reminder: printf does not add carriage return automatically**
  - Use %n to add carriage return
    - `System.out.printf("%s", valueOfAnyType);  // Same as print`
    - `System.out.printf("%s%n", valueOfAnyType); // Same as println`

Second Most Common Flag: %f

- **Overview**
  - For printing floating point numbers. Lets you control number of decimal points, total spacing, and commas in the main part of the number.
    - Often used to line up rows of numbers on the decimal point
- **Usage and options**
  - **%f** – prints all digits and with 0 at the end
    - `System.out.printf("%f", someDouble);`
  - **%.*d*f** – prints exactly *d* digits after the decimal point; the final digit is rounded
    - `System.out.printf("%.2f", someDouble);`
  - **%*n*.*d*f** – prints exactly *d* digits after the decimal point as above, and if the total output is less than *n* characters, pads with spaces on the left
    - `System.out.printf("%7.2f", someDouble);`
  - **%,*n*.*d*f** – Same as above, but adds commas every 3 digits in main part of number
    - `System.out.printf("%,7.2f", someDouble);`

Printf Example: Using %s and %f

```
double num = 1234.56722;
System.out.printf("num is '%s' (using %%s)%n", num);
System.out.printf("num is '%12s' (using %%12s)%n", num);
System.out.printf("num is '%f' (using %%f)%n", num);
System.out.printf("num is '%.2f' (using %%.2f)%n", num);
System.out.printf("num is '%4.2f' (using %%4.2f)%n", num);
System.out.printf("num is '%3.2f' (using %%3.2f)%n", num);
System.out.printf("num is '%10.3f' (using %%10.3f)%n", num);
System.out.printf("num is '%,10.3f' (using %%,10.3f)%n", num);
```

```
num is '1234.56722' (using %s)
num is '  1234.56722' (using %12s)
num is '1234.567220' (using %f)
num is '1234.57' (using %.2f)
num is ' 1234.57' (using %8.2f)
num is '1234.57' (using %6.2f)
num is '  1234.567' (using %10.3f)
num is ' 1,234.567' (using %,10.3f)
```

Printf Example: Controlling Width and Precision

```
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 never throws away significant digits, so the salary of Bill Gates is not lined up properly. Conclusion: you must know something about the range of possible values if you want to line things up on the decimal points.

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**

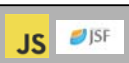
```
System.out.printf("Blah %s", 7);

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

- **Forgetting %s can be used for any type**
  - In Java, %s can be used for strings, objects, numbers, etc.
    - You only need number-specific placeholders like %f and %d when you are doing number-specific formatting like controlling decimal points, inserting commas, etc.
  - In C++, %s can be used only for strings
- **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 %%
  - You do not use \%






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 printf 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));
  }
}
```


Rare But Tricky Problem: Primitive Arrays with Varargs

- **Problem**
  - If you pass ints one at a time to a method that uses Object..., each one is converted to Integer, and things work as you expect
  - If you group the same numbers into an Integer[], you get the same result, as expected
  - But, if you pass an array of primitives to the same method, the entire array is considered a single element
- **This comes up later with Streams**
  - Passing ints one at time to Stream.of results in a Stream<Integer> containing each number separately
  - Passing an Integer[] to Stream.of also results in a Stream<Integer> containing each number separately
  - Passing an int[] to Stream.of results in a one-element stream, where the one element is the int[]

Object... Method

```
public class PrintUtilities {
  public static void printAll(Object... entries) {
    for(Object o: entries) {
      System.out.println(o);
    }
  }
}
```


Test Code

```
public class VarArgsTest {
  public static void main(String[] args) {
    PrintUtilities.printAll(1, 2, 3);
    Integer[] nums1 = { 1, 2, 3 };
    PrintUtilities.printAll(nums1);
    int[] nums2 = { 1, 2, 3 };
    PrintUtilities.printAll(nums2);
  }
}
```

```
1
2
3
1
2
3
[I@2a139a55
```





StringBuilder

Overview

- **Strings are immutable**
  - Once a String object is allocated, it cannot be modified.
  - However, a variable that refers to a String can be changed to refer to a new String that was derived from old one
- **String concatenation**
  - Results in copying the String that is before the "+"
- **Performance implications**
  - For a few fixed concatenations: no problem
  - For repeated concatenation in a loop: could be a problem
- **StringBuilder**
  - Alternative that can be directly modified
  - Also supports useful reverse method

Strings are Mutable?

```
public class StringsAppearMutable {
  public static void main(String[] args) {
    String s = "Hello";
    s = s + ", World";
    System.out.println(s);
  }
}
```

The string was modified here, right?

You actually cannot tell from this test. Maybe the original String object was changed, or maybe a new String object was created, and the original one was copied. You need a better test.

```
Hello, World
```


Strings are Immutable

```
public class StringsAreImmutable {
  public static void main(String[] args) {
    String s1 = "Hello";
    String s2 = s1;
    s1 = s1 + ", World";
    System.out.println(s1);
    System.out.println(s2);
  }
}
```

Since s2 is still “Hello”, this shows that the line that did concatenation really copied s1, allocated a new String object, and assigned that new object to s1.

```
Hello, World
Hello
```

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 by passing the String to the constructor
    ```
    StringBuilder b = new StringBuilder(someString);
    ```
  - Call append to append data to the end
    ```
    b.append("more");
    ```
  - Call toString to turn back into a string
    ```
    String s = b.toString();
    ```
  - 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 also for the upcoming StringBuilder version
- **Performance**
  - $O(N^2)$. Why?

Performance Comparison: Using StringBuilder

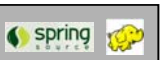
- **Code**

```
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

- **Generic types in your code**
  - `public class MyClass<E> { ... }`
  - `public interface MyInterface<E1,E2> { ... }`
  - `public static <T> T someMethod(T[] entries) { ... }`
- **printf**

```
System.out.printf("%s, %s, and %s.%n", v1, v2, v3);
```

- **Varargs**

```
public static int min(int... nums) { ... }
```

  - You can call min(int1, int2, int3) or min(intArray)
  - nums above is int[]
- **StringBuilder**
  - Better performance than String if many repeated concatenations
  - Some convenient methods like reverse and insert