

String Formatting and Regular Expressions



Jim Wilson

MOBILE SOLUTIONS DEVELOPER & ARCHITECT

@hedgehogjim blog.jwhh.com



Overview



Moving beyond string concatenation

StringJoiner class

Creating strings with format specifiers

Writing formatted content to a stream

Regular expressions

String class support for regular expressions

Regular expression classes



More Powerful Solutions to Creating String Representations

The need for more powerful string creation

- Concatenating strings is often not enough
 - Very focused on creation details
 - Numeric conversions awkward
- StringBuilder has the same issues

Options for more powerful string creation

- StringJoiner
 - Simplifies joining a sequence of values
- String formatting
 - Can specify desired appearance without dealing with creation details



StringJoiner

StringJoiner has a specific purpose

- Simplify composing a string comprised of a sequence of values

How it works

- Construct the StringJoiner
 - Specify string to separate values
 - Optionally specify start/end strings
- Add values
- Retrieve the resulting string



StringJoiner with Separator

```
StringJoiner sj = new StringJoiner(", ");  
sj.add("alpha");  
sj.add("theta");  
sj.add("gamma");  
String theResult = sj.toString();
```

alpha, theta, gamma



StringJoiner Chaining Method Calls

```
StringJoiner sj = new StringJoiner(", ");  
sj.add("alpha")  
String theResult = sj.toString();
```

alpha, theta, gamma



StringJoiner with Start and End Values

```
StringJoiner sj = new StringJoiner(", " );  
sj.add("alpha");  
sj.add("theta");  
sj.add("gamma");  
String theResult = sj.toString();
```

```
{alpha, theta, gamma}
```



StringJoiner with More Involved Separator

```
StringJoiner sj = new StringJoiner("[", "[", " ");  
sj.add("alpha");  
sj.add("theta");  
sj.add("gamma");  
String theResult = sj.toString();
```



[alpha], [theta], [gamma]



StringJoiner Edge Case Handling

toString when only one value added

- When constructed with separator only
 - Returns the added value
- When constructed with start/end strings
 - Returns added value within start/end



Handling a Single Value

```
StringJoiner sj1 = new StringJoiner(", ");
```

```
sj1.add("alpha");
```

```
String theResult1 = sj1.toString();
```

alpha

```
StringJoiner sj2 = new StringJoiner(", ", "{", "}");
```

```
sj2.add("alpha");
```

```
String theResult2 = sj2.toString();
```

{alpha}

StringJoiner Edge Case Handling

toString when no values added

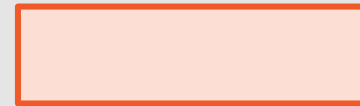
- When constructed with separator only
 - Returns empty string
- When constructed with start/end strings
 - Returns string with start/end only



Handling No Added Values

```
StringJoiner sj1 = new StringJoiner(", ");
```

```
String theResult1 = sj1.toString();
```



```
StringJoiner sj2 = new StringJoiner(", ", "{", "}");
```

```
String theResult2 = sj2.toString();
```



Can Customize Empty Handling

Can specify a special string for empty case

- Specified with setEmptyValue method
- Used only when add method not called



Customizing Empty Handling

```
StringJoiner sj1 = new StringJoiner(", ");
```

```
sj1.setEmptyValue("EMPTY");
```

```
String theResult1 = sj1.toString();
```

EMPTY

```
StringJoiner sj2 = new StringJoiner(", ", "{", "}");
```

```
sj2.setEmptyValue("EMPTY");
```

```
String theResult2 = sj2.toString();
```

EMPTY



Custom Empty Handling

```
StringJoiner sj1 = new StringJoiner(", ");
```

```
sj1.setEmptyValue("EMPTY");
```

```
sj1.add(" ");
```

```
String theResult1 = sj1.toString();
```



```
StringJoiner sj2 = new StringJoiner(", ", "{", "}");
```

```
sj2.setEmptyValue("EMPTY");
```

```
sj2.add(" ");
```

```
String theResult2 = sj2.toString();
```



Constructing Strings with Format Specifiers

Format specifiers

- Focus is on describing the desired result
 - Not concerned with the how
- Can control many aspects of appearance
 - Positioning
 - Decimal places
 - Representation

Some methods supporting format specifiers

- `String.format`
- `System.out.printf`
- `Formatter.format`



Concatenation vs. Formatting

```
int david = 13, dawson = 11, dillon = 4, gordon = 2;
```

```
My nephews are 13, 11, 4, and 2 years old
```

```
String s1 =  
    "My nephews are " + david + ", " + dawson  
    + ", " + dillon + ", and " + gordon + " years old";
```

```
String s2 = String.format(  
    "My nephews are %d, %d, %d, and %d years old",  
    david, dawson, dillon, gordon);
```



Concatenation vs. Formatting

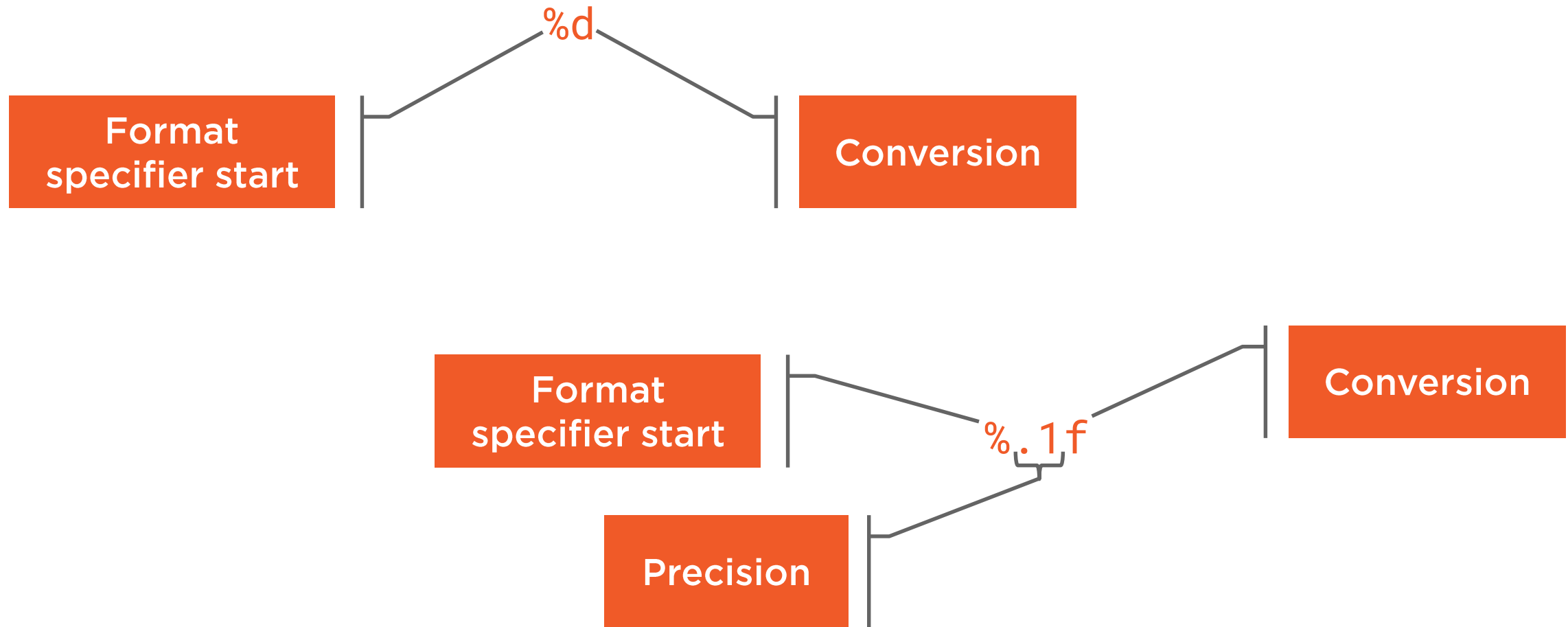
```
int david = 13, dawson = 11, dillon = 4, gordon = 2;  
double avgDiff = ((david - dawson) +  
    (dawson - dillon) + (dillon - gordon)) / 3.0d;
```

```
String s3 =  
    "The average age between each is "  
    "The average age between each is 3.66666666666666666665 years"
```

```
String s4 = String.format(  
    "The average age between each is %.1f years"  
    "The average age between each is 3.7 years"
```



Format Specifiers



Parts of a Format Specifier

Decimal places to
display

% [argument index] [flags] [width] [precision] conversion

Minimum characters to
display
(Space-padded, right-
justified by default)



Common Format Conversions

	Meaning	Type	Example Value	Result
d	Decimal	Integral	32	32
o	Octal	Integral	32	40
x X	Hex	Integral	32	20
f	Decimal	Floating Point	123.0	123.000000
e E	Scientific Notation	Floating Point	123.0	1.230000e+02
s	String	General	“Hello”	Hello
			<i>Implements Formattable</i>	<i>Return value of format method</i>
			<i>Other classes</i>	<i>Return value of toString method</i>



Format Flags

Flag	Meaning
#	Include radix



Format Flag:

String s1 = String.format("%d", 32); 32

String s2 = String.format("%o", 32); 40

String s3 = String.format("%x", 32); 20

String s4 = String.format("%#o", 32); 040

String s5 = String.format("%#x", 32); 0x20

String s6 = String.format("%#X", 32); 0X20



Format Flags

Flag	Meaning
#	Include radix
0	Zero-padding
-	Left justify



Format Flags: 0 and -

```
s1 = String.format("W:%d X:%d", 5, 235);
```

W:5 X:235

```
s2 = String.format("Y:%d Z:%d", 481, 12);
```

Y:481 Z:12

```
s3 = String.format("W:%4d X:%4d", 5, 235);
```

W: 5 X: 235

```
s4 = String.format("Y:%4d Z:%4d", 481, 12);
```

Y: 481 Z: 12

```
s5 = String.format("W:%04d X:%04d", 5, 235);
```

W:0005 X:0235

```
s6 = String.format("Y:%04d Z:%04d", 481, 12);
```

Y:0481 Z:0012

```
s7 = String.format("W:%-4d X:%-4d", 5, 235);
```

W:5 X:235

```
s8 = String.format("Y:%-4d Z:%-4d", 481, 12);
```

Y:481 Z:12



Format Flags

Flag	Meaning
#	Include radix
0	Zero-padding
-	Left justify
,	Include grouping separator



Format Flag: ,

```
s1 = String.format("%d", 1234567);
```

1234567

```
s2 = String.format("%,d", 1234567);
```

1,234,567

```
s3 = String.format("%,.2f", 1234567.0);
```

1,234,567.00



Format Flags

Flag	Meaning
#	Include radix
0	Zero-padding
-	Left justify
,	Include grouping separator
<i>space</i>	Leave space for positive numbers
+	Always shown sign
(Enclose negative values in parenthesis



Format Flags: Space, +, and (

s1 = String.format("%d", 123);	123
s2 = String.format("%d", -456);	-456
s3 = String.format("% d", 123);	123
s4 = String.format("% d", -456);	-456
s5 = String.format("%+d", 123);	+123
s6 = String.format("%+d", -456);	-456
s7 = String.format("%(d", 123);	123
s8 = String.format("%(d", -456);	(456)
s9 = String.format("% (d", 123);	123



Argument Index

Index	Meaning
<i>Not specified</i>	Corresponds sequentially to argument
<i>\$index</i>	Index of argument to use
<	Corresponds to same argument as previous format specifier



Argument Index

```
s1 = String.format("%d %d %d", 100, 200, 300);
```

100 200 300

```
s2 = String.format("%3$d %1$d %2$d", 100, 200, 300);
```

300 100 200

```
s3 = String.format("%2d %<04d %1d", 100, 200, 300);
```

200 0200 100



Writing Formatted Content to a Stream

Formatter class

- Provides formatting capabilities
- Writes content to any type that implements Appendable interface

Writer stream class

- Implements Appendable interface



Writing Formatted Content to a Stream

```
void doWrite(int david, int dawson, int dillon,  
            int gordon, double avgDiff) throws IOException {  
    BufferedWriter writer =  
        Files.newBufferedWriter(Paths.get("myFile.txt"));  
    try(Formatter f = new Formatter(writer)) {  
        f.format("My nephews are %d, %d, %d, and %d years old",  
                david, dawson, dillon, gordon);  
        f.format("The average age between each is %.1f years",  
                avgDiff);  
    }  
}
```



For More on
Formatting

Java Formmater documentation

- Detailed format specifier information
 - <http://bit.ly/java8formatter>

Java 8 date/time formatting

- Pluralsight What's New in Java 8
 - <http://bit.ly/psjava8whatsnew>



String Matching with Regular Expressions

Regular expressions

- A powerful pattern matching syntax
- Finds/excludes groups of characters
 - `a` -match the letter a
 - `xyz` -match the sequence xyz
 - `\w+` -match 1+ word characters
(letter, digit, underscore)
 - `\b` -match word breaks

Java support for regular expressions

- Methods on the String class
- Dedicated classes



String Class Support for Regular Expressions

replaceFirst, replaceAll methods

- Returns a new updated strings
- Pattern identifies which parts to change

split method

- Splits string into an array
- Pattern is the separator between values

match method

- Identifies if string matches the pattern



Using the replaceAll Method

```
String s1 = "apple, apple and orange please";
```

```
String s2 = s1.replaceAll("ple", "ricot");
```

apricot, apricot and orange ricotase

```
String s3 = s1.replaceAll("ple\\b", "ricot");
```

apricot, apricot and orange please



Using the split and match Methods

```
String s1 = "apple, apple and orange please";
```

```
String[] parts = s1.split("\\b");
```

```
for(String thePart:parts)
    if(thePart.matches("\\w+"))
        System.out.println(thePart);
```

apple
apple
and
orange
please

"apple"
" , "
"apple"
" "
"and"
" "
"orange"
" "
"please"



Dedicated Regular Expression Classes

Regular expression considerations

- Compilation is processing intensive
- String methods repeat compilation on every use

Pattern class

- Compiles a regular expression
- Factory for Matcher class instances

Matcher class

- Applies compiled expression to a string



Using Pattern and Matcher Classes

```
String value1 = "apple, apple and orange please";
```

```
Pattern pattern = Pattern.compile("\\w+");
```

```
Matcher matcher = pattern.matcher(value1);
```

```
while(matcher.find())
```

```
    System.out.println(matcher.group());
```

```
apple  
apple  
and  
orange  
please
```



For More on Working with Regular Expression Syntax

Java Pattern class documentation

- Overview of regular expression syntax
 - <http://bit.ly/java8pattern>

Java tutorial on regular expressions

- <http://bit.ly/javaregextutorial>

Interactive regular expression console

- <https://regex101.com/>
- Includes syntax quick reference



Summary



StringJoiner class

- Simplifies combining sequence of values
- Construct with value separator
 - Optionally specify start/end strings
- Add the values and retrieve string
- Can specify special value for empty
 - Empty means no values added

Summary



Format specifiers

- Focus on describing the desired result
- Parts of a specifier
 - % (required)
 - Conversion (required)
 - Precision
 - Flags
 - Argument index

String class supports format specifiers

Formatter class

- Writes formatted content to any class that implements Appendable interface



Summary



Regular expressions

- Powerful pattern matching syntax

String class support

- replaceFirst/All: Create new string
- split: Split string into an array
- match: Check for matching value

Dedicated classes

- Pattern
 - Compiles regular expression
- Matcher
 - Applies pattern to a string