String Formatting and Regular Expressions

Jim Wilson

Mobile Solutions Developer & Architect

@hedgehogjim | 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 Strings

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 focus on desired appearance

Avoids need to deal with creation details

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

Main.java

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

```
alpha, theta, gamma
```

StringJoiner Chaining Method Calls

Main.java

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

```
alpha, theta, gamma
```

StringJoiner with Start and End Values

Main.java

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

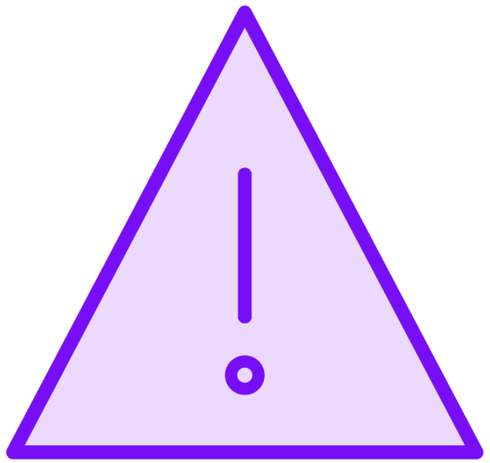
```
{alpha, theta, gamma}
```

StringJoiner with More Involved Separator

Main.java

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

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

Main.java

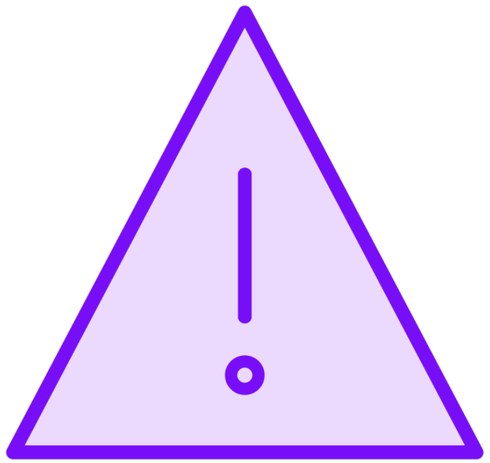
```
StringJoiner sj1 = new StringJoiner(", ");
sj1.add("alpha");
String theResult1 = sj1.toString();

StringJoiner sj2 =
    new StringJoiner(", ", "{", "}");
sj2.add("alpha");
String theResult2 = sj2.toString();
```

```
alpha

{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

Main.java

```
String Joiner sj1 = new StringJoiner(", ");
String theResult1 = sj1.toString();

StringJoiner sj2 =
     new StringJoiner(", ", "{", "}");
String theResult2 = sj2.toString();
```

Returns empty string

{}

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

Main.java

```
StringJoiner sj1 = new StringJoiner(", ");
sj1.setEmptyValue("EMPTY");
String theResult1 = sj1.toString();

StringJoiner sj2 =
    new StringJoiner(", ", "{", "}");
sj2. setEmptyValue("EMPTY");
String theResult2 = sj2.toString();
```

```
EMPTY

EMPTY
```

Custom Empty Handling

Main.java

```
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();
```

Returns empty string

{}

Concatenation vs. Formatting

```
My nephews are 17, 15, 8, and 6 years old
```

```
int david = 17, dawson = 15, dillon = 8, gordon = 6;

String s1 =
  "My nephews are "
   dillon

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

Concatenation vs. Formatting

```
The average age between each is 3.6666666666666665 years
```

```
int david = 17, dawson = 15, dillon = 8, gordon = 6;
double avgDiff = ((david - dawson) +  (dawson - dillon) +
      (dillon - gordon)) / 3.0d;

String s3 =
 "The average age between each is " + avgDiff + " years";
```

Concatenation vs. Formatting

```
The average age between each is 3.7 years
```

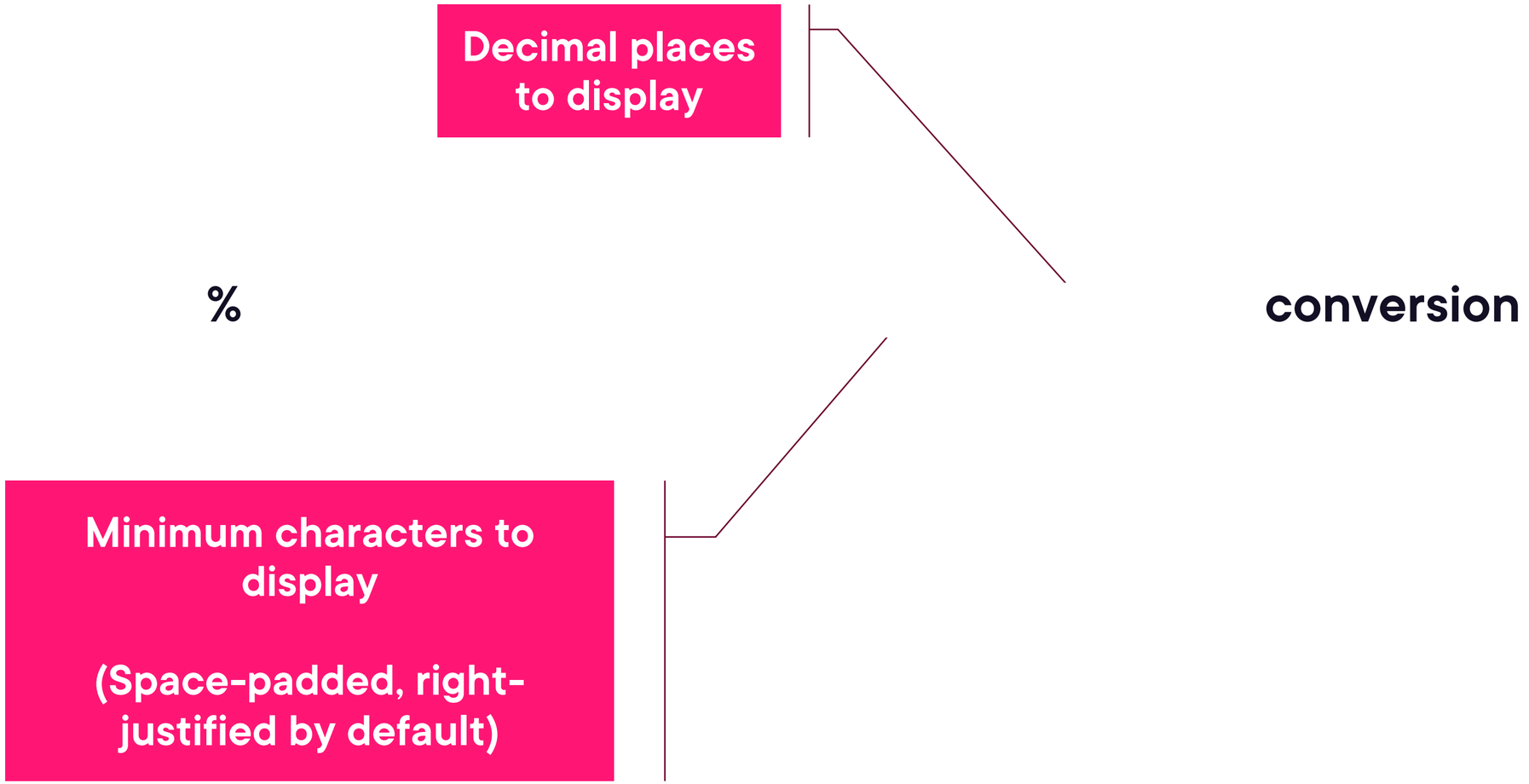
```
int david = 17, dawson = 15, dillon = 8, gordon = 6;
double avgDiff = ((david - dawson) +  (dawson - dillon) +
      (dillon - gordon)) / 3.0d;

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

Parts of a Format Specifier

%

Parts of a Format Specifier



Common Format Conversions

	Meaning	Type	Example Value	Result

Format Flags

Flag	Meaning

Format Flag:

Main.java

```
int iVal = 32;

String s1 = String.format("%d", iVal);
String s2 = String.format("%x", iVal);

String s3 = String.format("%#x", iVal);
String s4 = String.format("%#X", iVal);
```

```
32
20

0x20
0X20
```

Format Flags

Flag	Meaning
#	Include radix

Width and Format Flags: 0 and -

Main.java

```
int w = 5, x = 235, y = 481, z = 12;

s1 = String.format("W:%d X:%d", w, x);
s2 = String.format("Y:%d Z:%d", y, z);

s3 = String.format("W:%4d X:%4d", w, x);
s4 = String.format("Y:%4d Z:%4d", y, z);
```

```
W:5 X:235
Y:481 Z:12

W:   5 X: 235
Y: 481 Z:  12
```

Width and Format Flags: 0 and -

Main.java

```
int w = 5, x = 235, y = 481, z = 12;

s5 = String.format("W:%04d X:%04d", w, x);
s6 = String.format("Y:%04d Z:%04d", y, z);

s7 = String.format("W:%-4d X:%-4d", w, x);
s8 = String.format("Y:%-4d Z:%-4d", y, z);
```

```
W:0005 X:0235
Y:0481 Z:0012

W:5    X:235
Y:481  Z:12
```

Format Flags

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

Format Flags: ,

Main.java

```
int iVal = 1234567;
double dVal = 1234567.0d;

s1 = String.format("%d", iVal);
s2 = String.format("%,d", iVal);
s3 = String.format("%,.2f", dVal);
```

```
1234567
1,234,567
1,234,567.00
```

Format Flags

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

Format Flags: Space, +, and (

Main.java

```
int iPosVal = 123, iNegVal = -456;

s1 = String.format(“%d”, iPosVal);
s2 = String.format(“%d”, iNegVal);

s3 = String.format(“% d”, iPosVal);
s4 = String.format(“% d”, iNegVal);
```

```
123
-456

 123
-456
```

Format Flags: Space, +, and (

Main.java

```
int iPosVal = 123, iNegVal = -456;

s5 = String.format("%+d", iPosVal);
s6 = String.format("%+d", iNegVal);

s7 = String.format("%(d", iPosVal);
s8 = String.format("%(d", iNegVal);
s9 = String.format("% (d", iPosVal);
```

```
+123
-456

 123
(456)
 123
```

Argument Index

Index	Meaning

Argument Index

Main.java

```
int valA = 100, valB = 200, valC = 300;

s1 = String.format("%d %d %d",
     valA, valB, valC);

s2 = String.format("%3$d %1$d %2$d",
     valA, valB, valC);

s3 = String.format("%2$d %<d %1$d",
     valA, valB, valC);
```

```
100 200 300

300 100 200

200 200 100
```

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

String Matching with Regular Expressions

Regular expressions

- A powerful pattern matching syntax
- Finds/excludes groups of characters

Regular expression examples

- 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
```

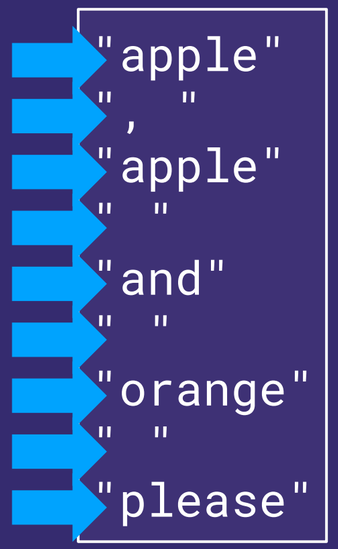
Using the split and match Methods

Main.java

```
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
```

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

Main.java

```
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