# Text Processing with Regular Expressions

- A string can be formatted or parsed based on a specified pattern that will be searched in the string.
- The search pattern is described by what is called a regular expression (*regex)* - `java.util.regex.Pattern`
- A piece of text that is found to correspond to the search pattern is called a *match* - `java.util.regex.Matcher`

# Regular Expressions

- Literal strings: `Kant`
- Regular expression constructs:
  - `[A-Za-Z0-9]`: any letter or digit
  - `[^A-Za-z0-9]`: any other character
- Important points
  - \ : an escape character
  - | : logical OR
  - ^ : to match the beginning of a line
  - $ : to match the end of a line
  - ^ : inside [ ] means negation

**Table 9-8.** *Character Classes (Brackets Used as Grouping Mechanism)*

| Construct | Description |
|---|---|
| `[ABC…]` | Any of the characters represented by A, B, C, etc. |
| `[^ABC…]` | Any character except A, B, C, etc. (negation) |
| `[a-zA-Z]` | a through z or A through Z (range) |
| `[…&&…]` | Intersection of two sets (`AND`) |

**Table 9-9.** *Predefined Character Classes*

| Construct | Description |
|---|---|
| `.` (dot) | Any character if the `DOTALL` flag is set, else any character except the line terminators |
| `\d` | A digit: `[0-9]` |
| `\D` | A non-digit: `[^0-9]` |
| `\s` | A whitespace character: `[\f\n\r\t\x0B]` |
| `\S` | A non whitespace character: `[^\s]` |
| `\w` | A word character: `[a-zA-Z0-9]` |
| `\W` | A non-word character: `[^\w]` |

**Table 9-10.** *Greedy Quantifiers (X Represents Regular Expression)*

| Construct | Description |
|---|---|
| X? | X, zero or one time |
| X* | X, zero or more times |
| X+ | X, one or more times |
| X{n} | X, exactly n times |

**Table 9-11.** *Some Other Constructs*

| Construct | Description |
|---|---|
| ^ | The beginning of a line |
| XY | Y following X |
| X\|Y | Either X or Y |
| (?:X) | X, as a noncapturing group |
| (?idmsux-idmsux) | Turns match flag on or off |
| (?idmsux-idmsux:X) | X, as a noncapturing group with a given flag on or off |

## The Typical Process for Pattern Matching

- Compile the regular expression specified as a string into an instance of the Pattern class:

```
Pattern p = Pattern.compile("[^a-zA-Z0-9]");
```

- Create a Matcher object that will contain the specified pattern and the input text to which the pattern will be matched:

```
Matcher m =
    p.matches("thinker@thinkingman.com");
```

- Invoke the find() (or matches()) method on the Matcher object to find if a match is found:

```
boolean b = m.find();
```

```
//Look for for email addresses starting with
//invalid symbols: dots or @ signs.
    Pattern p = Pattern.compile("^\\.+|^\\@+");
   Matcher m = p.matcher(email);
   if (m.find()) {
    System.err.println("Invalid email address: starts with a dot or an @ sign.");
    System.exit(0);
   }
//Look for email addresses that start with www.
p = Pattern.compile("^www\\.");
m = p.matcher(email);
   if (m.find()) {
    System.out.println("Invalid email address: starts with www.");
    System.exit(0);
   }
   p = Pattern.compile("[^A-Za-z0-9\\@\\.\\_]");
  m = p.matcher(email);

  if(m.find()) {
    System.out.println("Invalid email address: contains invalid characters");

   } else{
    System.out.println(args[0] + " is a valid email address.");
   }

   }
}
```

**Table 9-12.** *Some Useful Methods for Pattern Matching*

| Method | Class | Description |
|---|---|---|
| `static Pattern compile (String regex)` | `Pattern` | Compiles the input regular expression passed in as a string into a pattern |
| `boolean find()` | `Matcher` | Scans the input sequence only to find the next sub-sequence that matches the pattern |
| `Matcher matcher (CharSequence input)` | `Pattern` | Creates a `Matcher` object with the input that will be used to match against a pattern |
| `static boolean matches ➥ (String regex, CharSequence input)` | `Pattern` | Attempts to match the entire input sequence against the pattern |
| `boolean matches()` | `Matcher` | Attempts to match the entire input sequence against the pattern |
| `String[ ] split (CharSequence input)` | `Pattern` | Splits the given input sequence around matches of this pattern and returns the pieces as an array of strings |
| `String toString()` | `Pattern` | Returns the pattern as a string |

**Listing 9-8.** *SplitTest.java*

```
import java.util.regex.*;
public class SplitTest {
  public static void main(String[] args)  {
      String input= "www.cs.cornell.edu";
        Pattern p = Pattern.compile("\\.");
     String pieces[] = p.split(input);
     for (int i=0; i<pieces.length; i++){
         System.out.println(pieces[i]);
           }
   }
}
```

```
www
cs
cornell
edu
```

## Formatting and Parsing Streams

- Formatting and parsing are two sides of the same coin.
- On the sending end, an application formats the data into a certain format.
- On the receiving end, it breaks (parses) the data into useful pieces.

## Formatting Streams

- by using the `format()` method in the `Formatter`, `PrintWriter`, and `String` classes

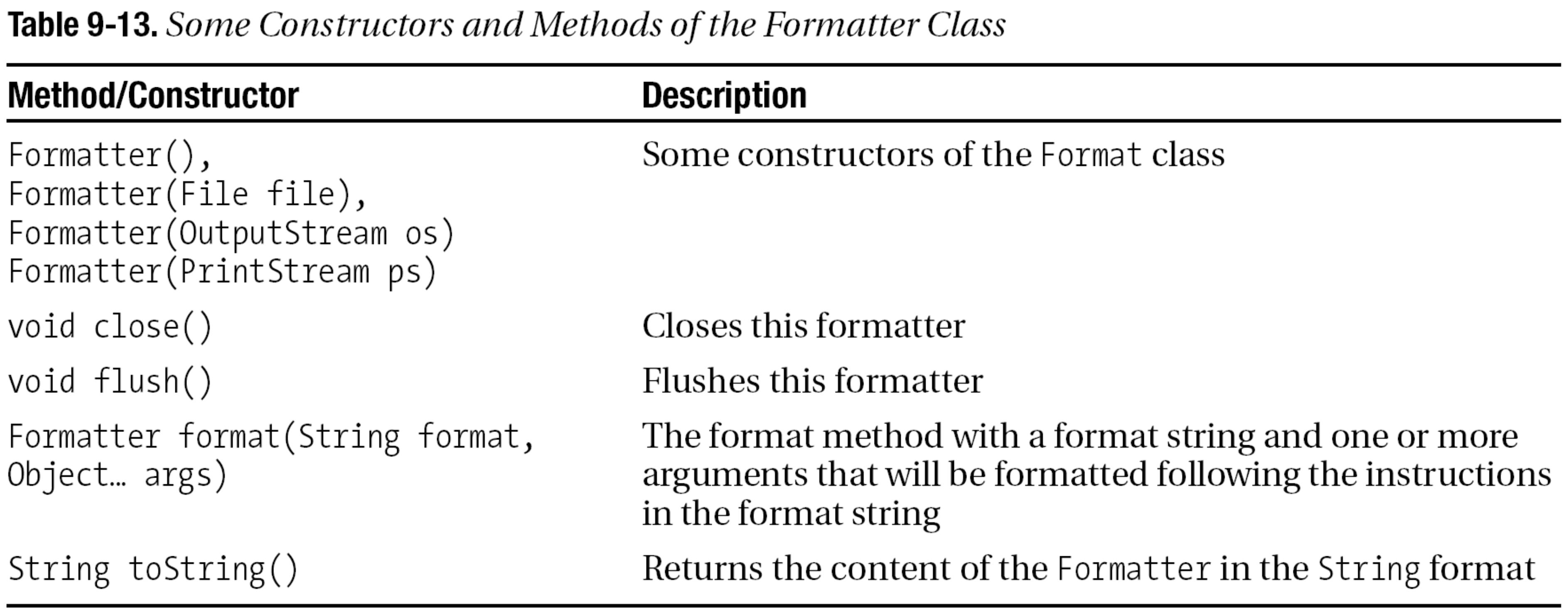
**Table 9-13.** *Some Constructors and Methods of the Formatter Class*

| Method/Constructor | Description |
|---|---|
| `Formatter()`, `Formatter(File file)`, `Formatter(OutputStream os)` `Formatter(PrintStream ps)` | Some constructors of the `Format` class |
| `void close()` | Closes this formatter |
| `void flush()` | Flushes this formatter |
| `Formatter format(String format, Object… args)` | The format method with a format string and one or more arguments that will be formatted following the instructions in the format string |
| `String toString()` | Returns the content of the `Formatter` in the `String` format |

# The format() method

```
format(<format specifier>, <argument>);
```

**`<format specifiers>:`**

```
%[<argumentIndex>$][<flags>][<width>][.<precisi
  on>] <type>
```

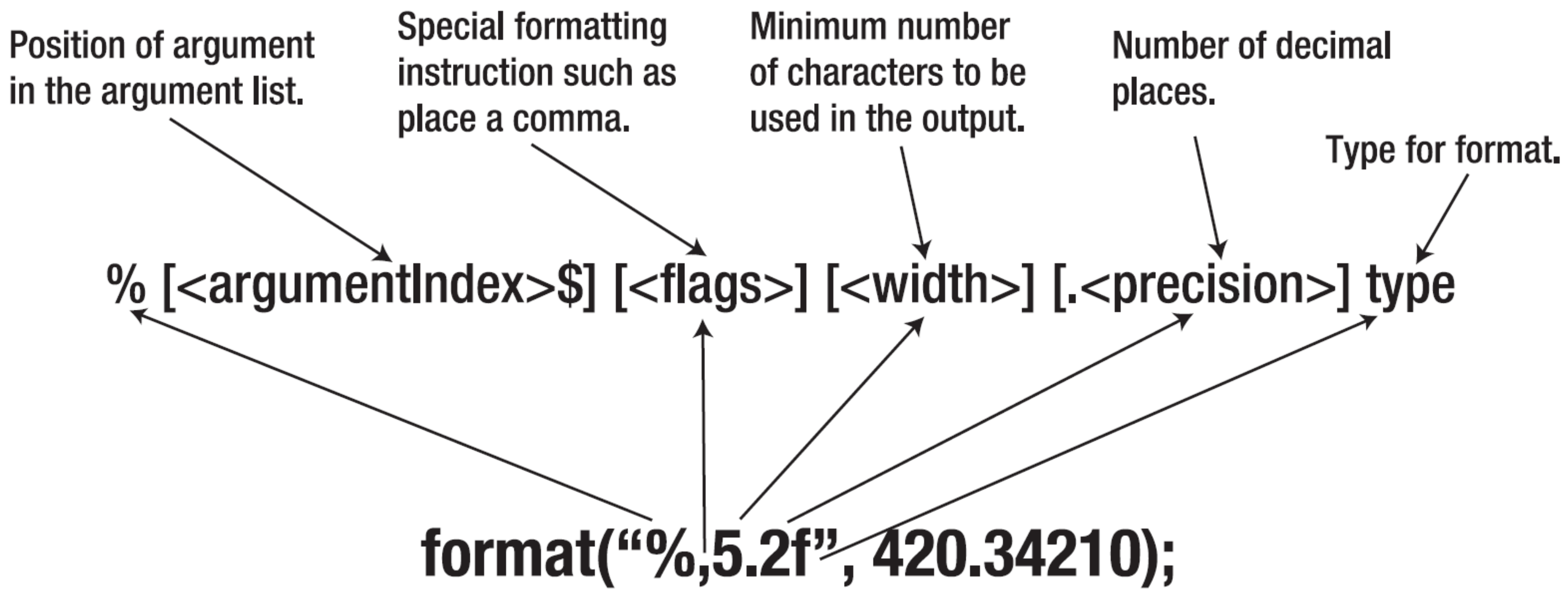
Position of argument in the argument list.
Special formatting instruction such as place a comma.
Minimum number of characters to be used in the output.
Number of decimal places.
Type for format.
% [<argumentIndex>$] [<flags>] [<width>] [.<precision>] type
format("%,5.2f", 420.34210);

**Table 9-14.** *Partial List of Format Types*

| Parameter | Type | Description |
|---|---|---|
| %b | Boolean | If the argument is `null`, the result is `false`. If the argument is `boolean` or `Boolean`, the result is `String.valueOf()`, else the result is `true`. |
| %c | Character | The result is a Unicode character. The argument must be a `byte`, `short`, `char`, or `int` (either primitives or wrappers). Wrappers are discussed later in this chapter. |
| %d | Decimal | The result is formatted as a decimal integer. The argument must be compatible with an `int`; that is, it must be `byte`, `short`, `char`, or `int` (either primitives or wrappers). |
| %f | Floating point | The result is formatted as a decimal number. The argument must be a `float` or a `double` (primitive or wrapper). |
| %s | String | If the argument is `null`, the result is `null`. The result is obtained by invoking `arg.toString()`. |

**Listing 9-9.** *FormatterTest.java*

```
import java.util.*;
 public class FormatterTest {
 public static void main(String[] args)  {
   Formatter formatter = new Formatter();
   System.out.println(formatter.format("%c", 33).toString());
   System.out.println(formatter.format("%8.2f", 420.23).toString());
   System.out.println(formatter.format("%8.2f", new Double(4234.23)).toString());
   System.out.println(formatter.format("%5b", " ").toString());
   System.out.println(formatter.format("%20d", 42042042).toString());
    System.out.println(formatter.format("%,20d", 42042042).toString());
   System.out.println(formatter.toString());
  }
 }
```

The output from Listing 9-9 follows:

```
!
!  420.23
!  420.23 4234.23
!  420.23 4234.23 true
!  420.23 4234.23 true            42042042
!  420.23 4234.23 true            42042042          42,042,042
!  420.23 4234.23 true            42042042          42,042,042
```

**Listing 9-10.** *StringFormatTest.java*

```java
public class StringFormatTest {
    public static void main(String[] args)  {
        System.out.println(String.format("%c", 33));
        System.out.println(String.format("%8.2f", 420.23));
        System.out.println(String.format("%8.2f", new Double(4234.23)));
        System.out.println(String.format("%5b", " "));
        System.out.println(String.format("%20d", 42042042));
        System.out.println(String.format("%,20d", 42042042));
    }
}
```

```
!
  420.23
4234.23
true
  42042042
42,042,042
```

**Listing 9-11.** *FormatterStreamTest.java*

```
import java.util.*;
import java.io.*;
 public class FormatterStreamTest {
  public static void main(String[] args) throws IOException {
      Formatter formatter = new Formatter("c:\\tmp\\formatterTest.txt");
           //The path to the file must exist before the code is executed.
      formatter.format("%c", 33);
      formatter.format("%8.2f", 420.23);
      formatter.format("%8.2f", new Double(4234.23));
      formatter.format("%5b", " ");
      formatter.format("%20d", 42042042);
      formatter.format("%,20d", 42042042);
      formatter.flush();
  }
}
```

**Listing 9-12.** *PrintFormatTest.java*

```
import java.io.*;
public class PrintFormatTest {
  public static void main(String[] args) throws IOException {
   PrintWriter pw = new PrintWriter("C:\\tmp\\writerTest.txt");
      //The path to the file must exist before executing this code.
      pw.format("%c", 33);
      pw.format("%8.2f", 420.23);
      pw.format("%8.2f", new Double(4234.23));
      pw.format("%5b", " ");
      pw.format("%20d", 42042042);
      pw.format("%,20d", 42042042);
      pw.flush();
  }
}
```

## Parsing Streams

- You can use the `Scanner` class as a simple text scanner to parse primitive types and strings using regular expressions.
- The input text that needs to be parsed can be passed to the `Scanner` constructor as a `String`, `File`, or an `InputStream`.
- The individual tokens can be converted into values of different types by using a suitable `next()` method.

**Table 9-15.** *Some Constructors and Methods of the Scanner Class*

| Methods/Constructors | Description |
|---|---|
| `Scanner(File source)`<br>`Scanner(InputStream source)`<br>`Scanner(String source)` | Constructors |
| `void close()` | Closes the scanner |
| `boolean hasNext()` | Returns `true` if this scanner has another token in its input |
| `boolean hasNext(String pattern)` | Returns `true` if the next token matches the pattern passed in as a string argument |
| `boolean hasNextBoolean()`<br>`boolean hasNextByte()`<br>`boolean hasNextDouble()`<br>`boolean hasNextFloat()`<br>`boolean hasNextInt()`<br>`boolean hasNextLong()`<br>`boolean hasNextShort()` | Returns `true` if the next token in the input of this scanner can be interpreted as the type spelled by the last word of the method name; for example, `byte` for `hasNextByte()` |
| `boolean hasNextLine()` | Returns `true` if there is another line in the input of this scanner |

**Table 9-15.** *Some Constructors and Methods of the Scanner Class*

| Methods/Constructors | Description |
|---|---|
| `boolean nextBoolean()`<br>`byte nextByte()`<br>`double nextDouble()`<br>`float nextFloat()`<br>`int nextInt()`<br>`long nextLong()`<br>`short nextShort()` | Returns the next token as a data type specified by the last word of the name (and the return type) of the method |
| `String next()` | Returns the next complete token from the scanner's input, in `String` format |
| `String nextLine()` | Returns the current line from the scanner's input and advances the scanner to the next line in `String` format |
| `String toString()` | Returns the scanner content in `String` format |
| `Scanner useDelimiter(String regex)` | Sets the passed-in regular expression as a pattern delimiter to parse (break) the scanner's input |

**Listing 9-13.** *ScannerTest.java*

```
import java.util.*;
  public class ScannerTest {
  public static void main(String[] args)  {
     String input = "cheque from publisher um 2000 dollars um buy
        diet pepsi um and peanuts";
     Scanner sc = new Scanner(input);
     System.out.println("Parsing round 1:");
     System.out.println(sc.next());
     System.out.println(sc.next());
      System.out.println(sc.next());
     System.out.println(sc.next());
     int salary = sc.nextInt();
     System.out.println("Advance:" + salary);
     sc.useDelimiter("um");
     System.out.println(sc.next());
     System.out.println(sc.next());
     System.out.println(sc.next());
     sc.close();
     System.out.println("Parsing round 2:");
     sc = new Scanner(input).useDelimiter("um");
      while(sc.hasNext()){
         System.out.println(sc.next());
      }
     }
 }
```

## Wrapping the Primitives

- Creating Objects of Wrapper Classes
  - Creating Wrapper Objects with the `new` Operator
  - Wrapping Primitives Using a `static` Method
- Methods to Extract the Wrapped Values
- The Instant Use of Wrapper Classes

## Creating Objects of Wrapper Classes

- Corresponding to each primitive data type in Java is a class called a *wrapper class.*
- Encapsulates a single value for the primitive data type.
- Created in one of two ways:
  - instantiate the wrapper class with the `new` operator
  - invoke a `static` method on the wrapper class

## Creating Wrapper Objects with the new Operator

**Table 9-16.** *Primitive Data Types and Corresponding Wrapper Classes*

| Primitive Data Type | Wrapper Class | Constructor Arguments |
|---|---|---|
| `boolean` | `Boolean` | `boolean` or `String` |
| `byte` | `Byte` | `byte` or `String` |
| `char` | `Character` | `char` |
| `short` | `Short` | `short` or `String` |
| `int` | `Integer` | `int` or `String` |
| `long` | `Long` | `long` or `String` |
| `float` | `Float` | `double`, `float`, or `String` |
| `double` | `Double` | `double` or `String` |

# Wrapping Primitives Using a static Method

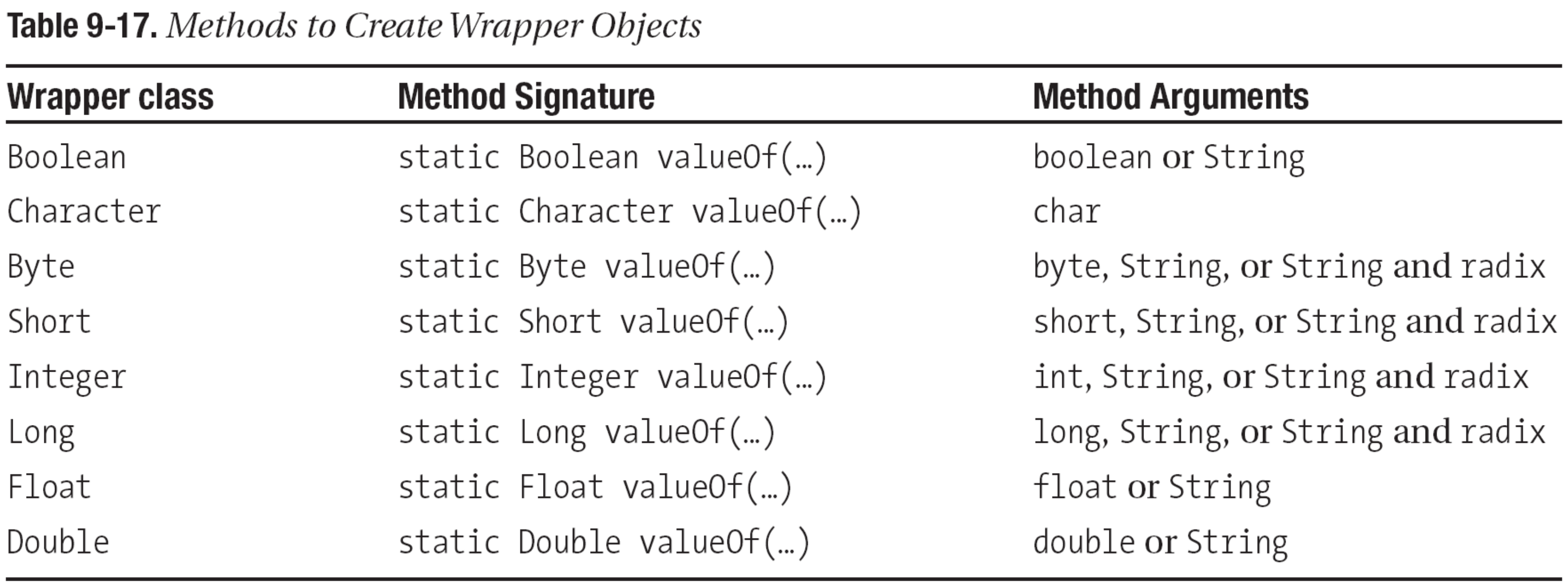

**Table 9-17.** *Methods to Create Wrapper Objects*

| Wrapper class | Method Signature | Method Arguments |
|---|---|---|
| `Boolean` | `static Boolean valueOf(…)` | `boolean` or `String` |
| `Character` | `static Character valueOf(…)` | `char` |
| `Byte` | `static Byte valueOf(…)` | `byte`, `String`, or `String` and `radix` |
| `Short` | `static Short valueOf(…)` | `short`, `String`, or `String` and `radix` |
| `Integer` | `static Integer valueOf(…)` | `int`, `String`, or `String` and `radix` |
| `Long` | `static Long valueOf(…)` | `long`, `String`, or `String` and `radix` |
| `Float` | `static Float valueOf(…)` | `float` or `String` |
| `Double` | `static Double valueOf(…)` | `double` or `String` |

# Methods to Extract the Wrapped Values

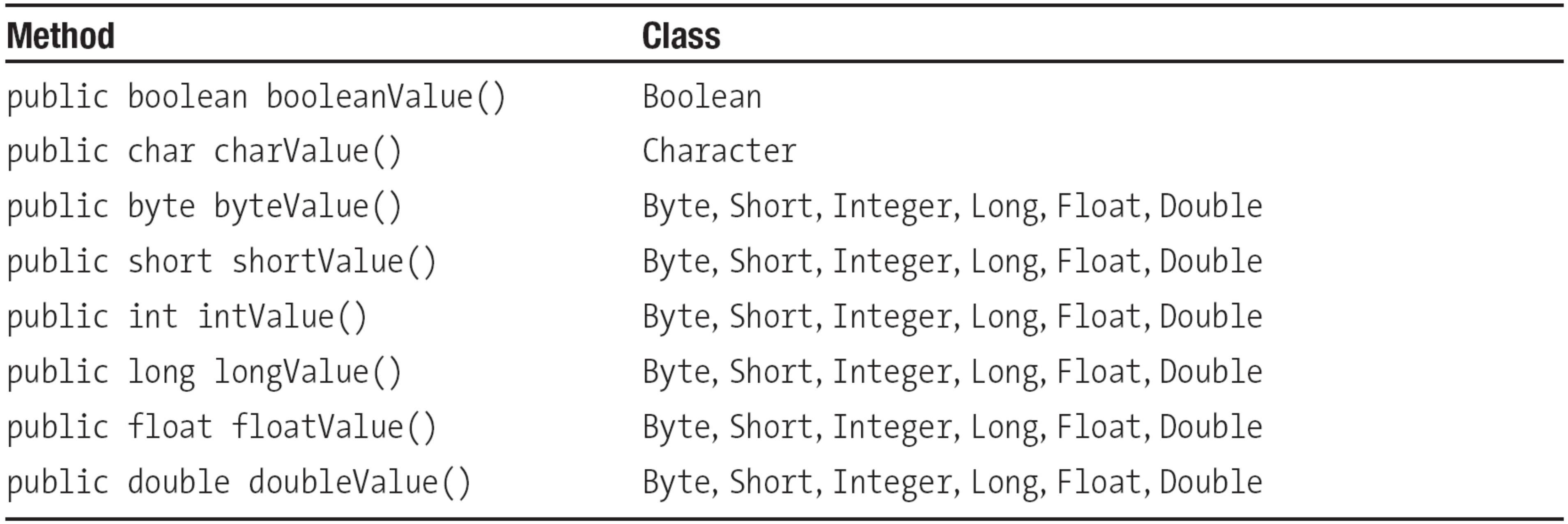
**Table 9-18.** *Methods to Retrieve Primitives from Wrapper Classes (All Methods Are No-Argument Methods)*

| Method | Class |
|---|---|
| `public boolean booleanValue()` | Boolean |
| `public char charValue()` | Character |
| `public byte byteValue()` | Byte, Short, Integer, Long, Float, Double |
| `public short shortValue()` | Byte, Short, Integer, Long, Float, Double |
| `public int intValue()` | Byte, Short, Integer, Long, Float, Double |
| `public long longValue()` | Byte, Short, Integer, Long, Float, Double |
| `public float floatValue()` | Byte, Short, Integer, Long, Float, Double |
| `public double doubleValue()` | Byte, Short, Integer, Long, Float, Double |

**Listing 9-14.** *ConversionMachine.java*

```
public class ConversionMachine{
  public static void main(String[] args){
    Byte b = 4;
    Byte wbyte = new Byte(b);
    double d = 354.56d;
    Double wdouble = new Double(d);
    System.out.println("wrapped Inside Byte: " + b);
    System.out.println("double value extracted from Byte: " +
        wbyte.doubleValue());
    System.out.println("Wrapped Inside Double: " + d);
    System.out.println("byte value extracted from Double: " +
        wdouble.byteValue());
  }
}
```

The output from Listing 9-14 follows:

```
wrapped Inside Byte: 4
double value extracted from Byte: 4.0
Wrapped Inside Double: 354.56
byte value extracted from Double: 98
```

# The Instant Use of Wrapper Classes

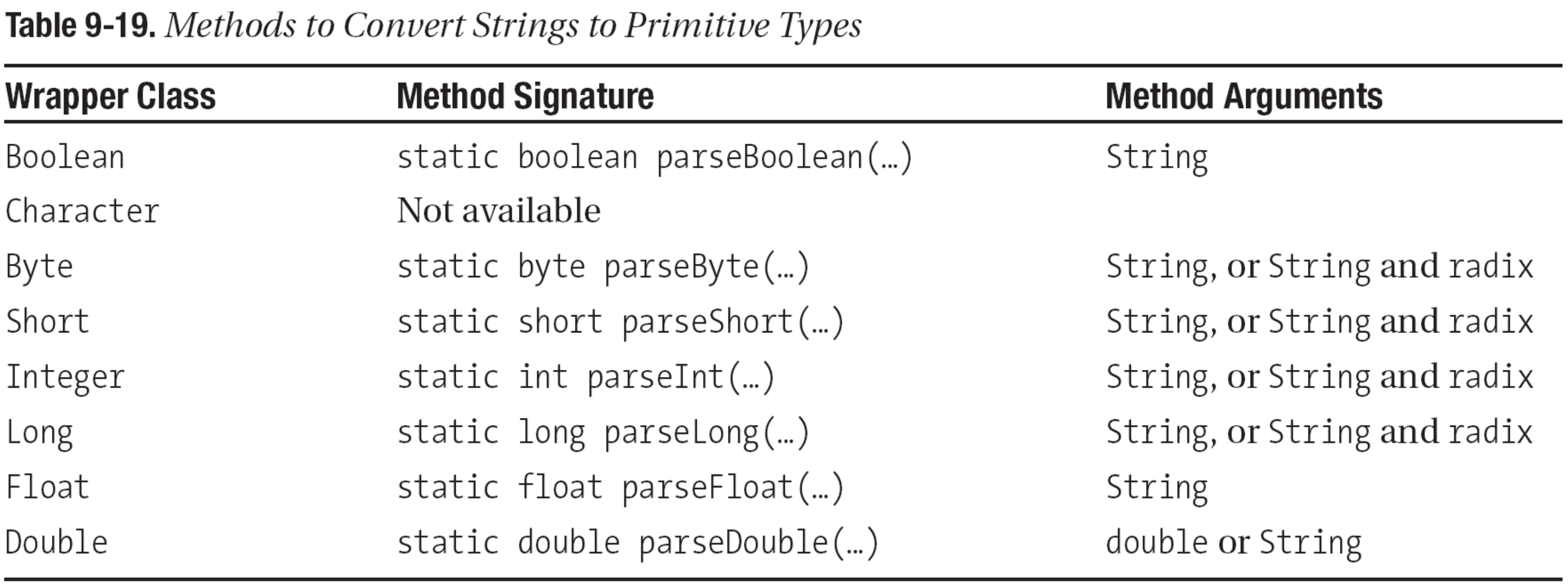

**Table 9-19.** *Methods to Convert Strings to Primitive Types*

| **Wrapper Class** | **Method Signature** | **Method Arguments** |
|---|---|---|
| `Boolean` | `static boolean parseBoolean(…)` | `String` |
| `Character` | Not available | |
| `Byte` | `static byte parseByte(…)` | `String`, or `String` and `radix` |
| `Short` | `static short parseShort(…)` | `String`, or `String` and `radix` |
| `Integer` | `static int parseInt(…)` | `String`, or `String` and `radix` |
| `Long` | `static long parseLong(…)` | `String`, or `String` and `radix` |
| `Float` | `static float parseFloat(…)` | `String` |
| `Double` | `static double parseDouble(…)` | `double` or `String` |