
Additional Java SE Classes



This chapter takes a look at some of the additional classes available in the Java SE library. There are two objectives for this chapter: firstly, to introduce you to the specific details of the classes covered here; and secondly, to encourage you to investigate other classes and capabilities in the Java SE library.

Contents

1. The Console class
2. The StringBuilder class
3. Formatting techniques
4. Regular expressions



Demo project:
`DemoAdditionalJavaSEClasses`

2

Section 1 looks at the `Console` class, which allows you to perform character-based I/O on the console attached to your application.

Section 2 shows how to use the `StringBuilder` class to build up textual content efficiently. We'll also look at a very similar class called `StringBuffer`, and explain the differences between the two.

Section 3 describes how to perform formatted output. For example, we'll show how to format dates, times, numbers, and so on.

Section 4 lifts the lid on regular expressions, a very important feature that allows you to perform pattern matching on your textual content.

The demos for chapter are located in the `DemoAdditionalJavaSEClasses` project.

1. The Console Class

- Overview of the Console class
- Console functionality
- Using the Console class



This section shows how to use the Console class. It's quite simple, so we'll cover this quickly!

Overview of the Console Class

- Java 1.6 introduces the `java.io.Console` class
 - Provides methods to access the character-based console device (if any) associated with the current JVM
 - Provides simplified access to keyboard/screen (unless I/O has been redirected)
- To get the (singleton) `Console` instance for the JVM:

```
import java.io.*;
...
Console console = System.console();
if (console != null) {
    ...
}
```

4

The `Console` class, introduced in Java 1.6, allows you to perform character-based console I/O. By default, the console is attached to the keyboard input device and the screen output device, although you can redirect these devices when you run your application (e.g. by using `>` and `<` when you start up the JVM).

There is a singleton `Console` object associated with the JVM. Your first step is to acquire a reference to this `Console` object, as shown in the slide above. Note that this might fail if the console is unavailable (e.g. in the Eclipse IDE), so be sure to check for a `null` reference.

Console Functionality

- Useful methods in the `Console` class:
 - `printf()` - print formatted string (also `format()`)
 - `writer()` - get a `PrintWriter` for the `Console`
 - `flush()` - force buffered content to be written now
 - `reader()` - get a `Reader` for the `Console`
 - `readLine()` - read a string (with optional prompt)
 - `readPassword()` - read a password (with optional prompt)

5

The `Console` class provides various instance methods for reading and writing to the console. The slide above lists the common methods. For full details, see the following web site:

- <http://docs.oracle.com/javase/7/docs/api/java/io/Console.html>

Using Console

■ Example

```
Console console = System.console();  
  
if (console != null) {  
  
    console.printf("Hello at %s. Please tell me your name: ", new Date());  
  
    String name = console.readLine();  
    String nat = console.readLine("Thanks %s, now tell me your nationality: ", name);  
    char[] pwChars = console.readPassword("Great %s, you're %s! Password? ", name, nat);  
  
    String password = new String(pwChars);  
    console.printf("Thanks. Your password is: %s", password);  
  
} else {  
    System.out.println("Sorry, console unavailable.");  
}  
  
UsingConsole.java
```

```
C:\JavaDev\Demos\DemoAdditionalJavaSEClasses\bin>java demo.additionaljavaseclasses.UsingConsole  
Hello at Thu Feb 20 09:21:10 GMT 2014. Please tell me your name: Andy  
Thanks Andy, now tell me your nationality: Welsh  
Great Andy, you're Welsh! Password?  
Thanks. Your password is: Super$wans
```

6

This slide illustrates how to use the `Console` class. This code is available in the demo project, but you can't run the program via Eclipse (because Eclipse captures the console, so the first `if`-statement in the code will fail immediately).

If you want to run the program, follow these steps:

- Open a command window.
- Change to the following directory:
C:\JavaDev\Demos\DemoAdditionalJavaSEClasses\bin
- Run the program manually through the JVM as follows:
java demo.additionaljavaseclasses.UsingConsole

2. The `StringBuilder` Class

- Overview of the `StringBuilder` class
- `StringBuilder` functionality
- Using `StringBuilder`



The `StringBuilder` class allows you to build up textual content one piece at a time. As you'll see, this is much more efficient than using `String` objects.

Note: Java also includes a class named `StringBuffer`, which offers the same capabilities as `StringBuilder` but in a thread-safe kind of way. We'll discuss this issue on the next slide.

Overview of `StringBuilder` Class

- `String` objects are immutable
 - Once you've created a `String` object, you can't modify it
 - This enables the JVM to optimize memory usage
 - But it can impair performance if you do a lot of string handling
- `StringBuilder` (and `StringBuffer`) are mutable
 - Assigned an initial capacity (16 characters by default)
 - Allows you to change the text
 - Automatically resizes when necessary
- `StringBuilder` vs. `StringBuffer`...
 - `StringBuilder` has the same API as `StringBuffer`
 - `StringBuilder` only available in Java 1.5 onwards
 - `StringBuilder` isn't thread-safe, so faster than `StringBuffer`
 - Recommendation: Use `StringBuilder` where possible

8

As we mentioned earlier in the course, `String` objects are immutable. There are good reasons for this, but the downside is that it makes the `String` class extremely inefficient if you need to build up textual content piecemeal or if you need to change the content of a string in any way. Basically, all the seemingly mutator methods in `String` don't modify the original `String` object at all, but create and return a new `String` object instead.

If you need to do any non-trivial amount of text processing, you are strongly advised to use the `StringBuilder` or `StringBuffer` classes. These two classes manipulate text in-situ, rather than creating a new copy all the time.

`StringBuilder` and `StringBuffer` offer exactly the same methods as each other. The only difference is that `StringBuilder` is not thread-safe (i.e. it doesn't perform any locking) whereas `StringBuffer` is thread-safe (which potentially makes it slower).

Note that `StringBuilder` is only available in Java 1.5 onwards, whereas `StringBuffer` is available in all versions.

StringBuilder Functionality

- Useful methods in the `StringBuilder` class:
 - `constructor` - optionally specify initial capacity or initial text
 - `append()` - append to text (various overloads)
 - `delete()` - delete text from start to end positions
 - `deleteCharAt()` - delete character at specific position
 - `insert()` - insert at specified position (various overloads)
 - `replace()` - replace text from start to end positions
 - `setCharAt()` - set character at specific position
 - `reverse()` - reverse text
 - `toString()` - create immutable `String` from text

- Note:
 - `StringBuffer` has the same methods 😊

9

The `StringBuilder` and `StringBuffer` classes provide various instance methods for creating, editing, and generally manipulating text in-situ. The slide above lists the common methods. After you've created all the text content you need, you can then call `toString()` to create an immutable `String` object as your final result.

For full details about these classes, see the following web sites:

- <http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html>
- <http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html>

Using StringBuilder

■ Example

```
StringBuilder sb1 = new StringBuilder("Hello");  
  
sb1.append(" world");  
sb1.append('!');  
  
sb1.insert(5, " Andy and the rest of the");  
sb1.insert(10, 43);  
  
String str1 = sb1.toString();  
System.out.println(str1);  
  
StringBuilder sb2 = sb1;  
sb2.replace(6, 9, "Fred");  
sb2.reverse();  
  
String str2 = sb2.toString();  
System.out.println(str2);
```

UsingStringBuilder.java

```
Hello Andy43 and the rest of the world!  
!dlrow eht fo tser eht dna 34yderF olleH
```

10

This slide illustrates how to use the `StringBuilder` class. The demo project also contains a similar block of code using `StringBuffer`, to highlight the fact that the two classes have the same API.

This code is available in the demo project, and you can run it without any problems in Eclipse.

3. Formatting Techniques

- Overview of formatting
- Example of using the `Formatter` class



We now turn our attention to the `Formatter` class, which allows you to create formatted output for the locale of your choice.

Overview of Formatting

- Java 1.6 introduced extensive formatting capabilities...
- `java.util.Formatter` class
 - Formats content in a string, stream, or file
 - Locale-sensitive
 - Extremely parameterized
- `String.format()` method
 - Creates a formatted string
- `System.out.format()` method
 - Outputs a formatted string

12

The `Formatter` class, defined in the `java.util` package, allows you to format dates, times, numbers etc. in a locale-specific way. You can work on strings, streams, or files.

Java also supports formatting via the `String.format()` method and the `System.out.format()` method. The formatting capabilities are the same as those of the `Formatter` class.

Note that `Formatter` is only available in Java 1.6 onwards.

Example of Using Formatter Class

- This example shows some simple Formatter techniques

```
private static void demoFormatter() {
    // Use a StringBuilder to accumulate output, using the UK locale.
    StringBuilder sb = new StringBuilder();
    Formatter formatter = new Formatter(sb, Locale.UK);

    // Simple output, with field-width specifiers.
    formatter.format("%2s %2s %2s %2s\n", "A", "B", "C", "D");

    // Reorder items, using explicit indices.
    formatter.format("%4$s %3$s %2$s %1$s\n", "A", "B", "C", "D");

    // Use different locale (One-off).
    formatter.format(Locale.FRANCE, "Here's a big number: %.4f\n", 123456789.1234567);

    // Output negative parameter in parentheses.
    formatter.format("Stocks and shares this week: %(.2f\n", -6217.577);

    System.out.println(sb.toString());
}
```

Formatting.java

```
A B C D
D C B A
Here's a big number: 123456789,1235
Stocks and shares this week: (6,217.58)
```

13

This slide shows how to use a `Formatter` object to format various output, and how to specify a particular locale to influence how the formatting is performed.

The demo project contains additional examples that show how to format numbers, dates, and times. The demo also includes some comments to help you understand the myriad options available.

For full details about the `Formatter` class, see the following web site:

- <http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html>

4. Regular Expressions

- Basic regular expression syntax
- Support for regular expressions in Java
- Performing a one-off pattern match
- Using compiled regular expressions
- Wildcards
- Greedy and reluctant matching
- Capturing groups
- Replacing text using regular expressions
- Additions to the `String` class

14

Regular expressions are templates that describe some text, so that you can:

- Search for text or pattern within a string
- Extract interesting parts of a text
- Optionally perform search and replace

This section shows how to achieve all these tasks.

Basic Regular Expression Syntax

- REs use literal characters and meta-characters
 - Use \ character to make meta-character into literal character

RE	Description
abc	Matches abc
.	Matches any single character
[a-zA-Z]	Matches any character in set
[^a-zA-Z]	Matches any character not in set
^	Matches beginning of line
\$	Matches end of line
x?	Matches 0 or 1 of x
x+	Matches 1 or more of x
x*	Matches 0 or more of x
x{m,n}	Matches between m and n of x's (you can omit <i>m</i> or <i>n</i>)
abc def	Matches abc or def
\.	Matches the literal .

15

Regular expressions use a rather cryptic shorthand notation that allows you to express the exact content you're looking for in some text. This slide summarizes the basic syntax of a regular expression in Java.

Support for Regular Expressions in Java

- Support for REs in Java since Java 1.4
 - `java.util.regex` package
- `CharSequence` interface
 - Encapsulates sequence of characters, implemented by `String`, `StringBuilder`, `StringBuffer`, `CharBuffer`
- `Pattern` class
 - Encapsulates an RE (in "compiled form")
- `Matcher` class
 - Supports matching of `Pattern` objects against a `CharSequence`
- `MatchResult` interface
 - Represents the result of a match operation

16

The `java.util.regex` package defines various classes and interfaces that allow you to use regular expressions in Java:

- The `Pattern` class allows you to create a compiled representation of a regular expression. The basic idea is that you specify your regular expression as a string, and then compile it into an instance of the `Pattern` class.
- The `Matcher` class is an engine that performs matching operations on a character sequence (i.e. a `String`, `StringBuilder`, `StringBuffer`, or `CharBuffer`). We'll describe the exact interplay between the `Pattern` and `Matcher` classes in the next few slides.
- The `MatchResult` interface provides query methods that allow you to determine the results of a regular expression match. We'll describe how this works when we discuss groups later in this section.

Performing a One-Off Pattern Match

- The `Pattern` class has a static `matches()` method
 - Allows you to perform a one-off pattern match, to see if a character sequence matches a regular expression
 - Just returns `true` or `false`
- Example

```
String charseq = "aaaab";  
boolean result = Pattern.matches("a*b", charseq);
```

17

If you just want to do a one-off pattern match on a particular character sequence, the simplest approach is to call the static `Pattern.matches()` method. This method takes two parameters:

- A string representing the regular expression
- A character sequence containing the text you want to search in

The method returns a boolean result, indicating whether the character sequence matched the regular expression.

Using Compiled Regular Expressions (1)

- If you're going to use an RE more than once, you should compile it into a Pattern object
 - Invoke `Pattern.compile()` to build a Pattern object

```
Pattern p = Pattern.compile("a*b");
```
- To match an RE match on a character sequence:
 - Invoke `matcher()` on Pattern object, returns Matcher object

```
Matcher m = p.matcher("aaaab");
```
- Matcher methods:
 - `matches()`
 - `lookingAt()`
 - `find()`
 - `group()`

18

The `Pattern.compile()` method returns a Pattern object that represents your compiled regular expression. You can then invoke the `matcher()` method to match the RE against a specific character sequence.

The `matcher()` method returns a Matcher object, which provides various methods for determining whether you have a match:

- `matches()`
Returns true if the entire character sequence matches the RE.
- `lookingAt()`
Returns true if the character sequence starts with the RE (like an implicit `^` in the RE). Note that there may be trailing characters at the end of the character sequence, but this would still constitute a match.
- `find()`
Returns true if the character sequence somewhere contains the RE. The search commences from where the previous match terminated.
- `group()`
Enables you to see the resultant matching text from the character sequence. We'll discuss groups shortly.

Using Compiled Regular Expressions (2)

■ Example:

```
private static void demoMatching(CharSequence source, String regex) {  
  
    Pattern p = Pattern.compile(regex);  
    Matcher m = p.matcher(source);  
  
    // Call matches() to see if there's an exact match on the entire source.  
    if (m.matches()) {  
        System.out.printf("matches() matches: %s\n", m.group());  
    } else {  
        System.out.printf("matches() doesn't match anything.\n");  
    }  
  
    // Call lookingAt() to see if there's a match from the start of the source.  
    if (m.lookingAt()) {  
        System.out.printf("lookingAt() matches: %s\n", m.group());  
    } else {  
        System.out.printf("lookingAt() doesn't match anything.\n");  
    }  
  
    // Call find() to see if there's a match somewhere in the source. Note reset() call!  
    m.reset();  
    if (m.find()) {  
        System.out.printf("find() matches: %s\n", m.group());  
    } else {  
        System.out.printf("find() doesn't match anything.\n");  
    }  
}
```

RegularExpressions.java

19

This slide shows a complete example of how to use compiled regular expressions. The comments in the code explain what it all means. Run the demo project to see the output.

Using Compiled Regular Expressions (3)

- The `find()` method continues searching from the end of the previous match
 - Makes repeated matches easy
 - Invoke `find(n)` to match the n^{th} occurrence of RE (it starts at 1)
- Example:

```
private static void demoRepeatedFinds(CharSequence source, String regex) {  
  
    Pattern p = Pattern.compile(regex);  
    Matcher m = p.matcher(source);  
  
    // Find each occurrence of RE in source, in a loop.  
    while (m.find()) {  
        System.out.printf("find() matches: %s\n", m.group());  
    }  
  
    // Find the 2nd occurrence of RE in source.  
    if (m.find(2)) {  
        System.out.printf("find(2) matches: %s\n", m.group());  
    }  
}
```

RegularExpressions.java

20

A common scenario when using regular expressions is to search a character sequence for a recurring pattern, such as records in a file. You can use the `find()` method to achieve this effect.

The `find()` method returns `true` if it finds a match. The key point is that the `find()` method remembers where it left off last time, so you can call it repeatedly in a loop to work your way through your character sequence.

Aside: Pattern Compilation Options

- Options for `Pattern.compile()` method 2nd parameter

Option	Description
<code>Pattern.LITERAL</code>	Treat the RE as literal characters (i.e. do not interpret meta-characters)
<code>Pattern.UNIX_LINES</code>	Recognize only <code>\n</code> as newline character (relevant for <code>^</code> and <code>\$</code>)
<code>Pattern.CASE_INSENSITIVE</code>	Case-insensitive matching (assumes only US-ASCII chars used)
<code>Pattern.UNICODE_CASE</code>	Use with <code>CASE_INSENSITIVE</code> for case-insensitive matching with Unicode
<code>Pattern.COMMENTS</code>	Allow whitespaces and comments (starting with <code>#</code> , to end of line) in pattern
<code>Pattern.MULTILINE</code>	Enable multiline mode, whereby <code>^</code> and <code>\$</code> match the beginning/end of each line, rather than the beginning/end of the entire character sequence
<code>Pattern.DOTALL</code>	Allow <code>.</code> to match line separator characters in multiline mode
<code>Pattern.CANON_EQ</code>	Enable Unicode canonical equivalence of characters

As an aside, the `Pattern.compile()` method allows you to fine-tune how the pattern-matching operation is performed. You can specify these options as the second parameter to the `Pattern.compile()` method.

Wildcards

- You can use wildcards to indicate a number of common character types

Wildcard	Description	Equivalent to ...
\d	A digit	[0-9]
\D	A non-digit	[^0-9]
\s	A whitespace character	[\t\n\x0B\f\r]
\S	A non-whitespace character	[^\s]
\w	A word character	[a-zA-Z_0-9]
\W	A non-word character	[^\w]

22

Regular expressions can include wildcards, as shown in the table above. These wildcards represent common types of search in regular expressions, and can considerably simplify the syntax in your regular expression string.

Note the capitalization in these wildcards. For example:

- \d matches a digit
- \D matches a non-digit

Be careful you say what you mean!


Greedy and Reluctant Matching

- An RE can sometimes lead to ambiguity when repeat quantifiers are used

- Default behaviour is to use greedy matching
- Match as much as possible

`([a-z]+)(.*)`

- `.*` could match the letters as well as other chars
- `([a-z]+)` matches as many chars as possible while satisfying the RE




abcdef0123456

- Non-greedy (reluctant) matching is also available

`([a-z]+?)(.*)`

- `([a-z]+?)` matches as little as necessary to satisfy the RE



abcdef0123456

Greedy vs. reluctant matching is quite a tricky topic. The essence of the problem is that a regular expression can theoretically be ambiguous, especially when you use quantifiers such as `+` and `*`.

The question is, just how much text does the regular expression include in its match? There are two options in Java regular expressions, as shown in the slide above. Here's a quick summary:

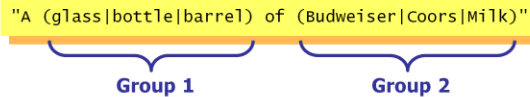
- The first code box shows greedy matching, whereby the regular expression tries to gobble up as much matching text as possible. This is the default mode.
- The second code box shows reluctant matching, whereby the regular expression tries to match as little matching text as possible. The key difference is the `?` at the end of the first part of the regular expression. Subtle eh?

Capturing Groups (1 of 2)

- You can partition an RE into groups

- Parenthesized sub-expressions
- Remember portions of text matched by sub-expressions

"A (glass|bottle|barrel) of (Budweiser|Coors|Milk)"



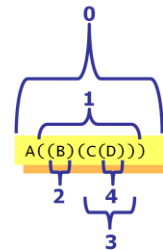
Group 1 Group 2

- Groups are indexed left to right

- Index 0 is the entire capture
- No account taken of nesting

- Use (?...) to prevent capture

- "Pure" groups do not figure in indexing



24

As well as being able to test for matches, regular expressions can also be used to extract portions of matching text. This technique is known as "capturing groups".

To capture groups, you enclose part or all of your regular expression in parentheses. You can have as many sets of parentheses you want - the code snippet at the bottom of the slide shows how each set of parentheses corresponds to a group number.

The following slide shows how to use groups in a full Java application.

Capturing Groups (2 of 2)

- To access a capture group:
 - Invoke `group(n)` method on `Matcher` object
 - Index argument is 1-based
 - Invoke `groupCount()` to determine number of capture groups
- Example:

```
private static void demoGroups(CharSequence source, String regex) {
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(source);

    // Perform the match.
    if (m.matches()) {
        // Iterate through all the capture groups.
        for (int i = 1; i <= m.groupCount(); i++) {
            System.out.printf("Group %d match: %s\n", i, m.group(i));
        }
    }
}
```

RegularExpressions.java

```
demoGroups("01792 123456", "(0\\d{3,4})[\\s-](\\d{6})");
```

Group 1 match: 01792
Group 2 match: 123456

25

This example shows how to use groups in Java regular expressions.

We begin by compiling a regular expression into a `Pattern` object. The regular expression is passed as a parameter into the `demoGroups()` method, e.g. `"(0\\d{3,4})[\\s-](\\d{6})"`. This regular expression represents a UK telephone number, and specifies two groups:

- The first group represents an area code (such as "01792", the area code for Swansea in the UK).
- The second group represents a local number in that area (such as "123456", a fictional number in Swansea).

The `demoGroups()` method matches the character sequence "01792 123456" against the RE. The `matches()` function indicates whether the character sequence matched the RE; if so, we call the `Matcher`'s `group()` method repeatedly to get each group. This will yield the actual text in the character sequence that matches each group in the RE.

The results are shown in the screenshot at the bottom of the slide.

Replacing Text Using Reg Exps

- **Matcher operations for replacing text:**
 - `replaceFirst(replacementString)`
 - Returns new `String` where first match is replaced
 - `replaceAll(replacementString)`
 - Returns new `String` where all matches are replaced
- **Example:**

```
private static void demoReplacement(CharSequence src, String regex, String replacement) {
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(source);

    String result1 = m.replaceFirst(replacement);
    System.out.printf("replaceFirst() gives: %s \tsource: %s\n", result1, src);

    String result2 = m.replaceAll(replacement);
    System.out.printf("replaceAll() gives: %s \tsource: %s\n", result2, src);
}
```

RegularExpressions.java

```
demoReplacement("01222-123456 01222-654321 01222-98765", "01222-", "02920-2");
```

↓

```
replaceFirst() gives: 02920-2123456 01222-654321 01222-98765    source: 01222-123456 01222-654321 01222-98765
replaceAll() gives: 02920-2123456 02920-2654321 02920-298765    source: 01222-123456 01222-654321 01222-98765
```

As well as searching and extracting text, REs can also be used to replace matching text in a character sequence. To do this, call either the `replaceFirst()` or `replaceAll()` method.

These methods don't modify the original character sequence. Rather, they return a string that contains the modified text after the replacement has taken place.

Additions to the String Class

- String class contains some new methods to support REs
 - `boolean matches(String re)`
 - `String[] split(String re) // optional: int limit`
 - `String replaceFirst(String re, String repl)`
 - `String replaceAll(String re, String repl)`
- Example:

```
private static void demoStringRE() {  
    String greeting = "Hello mum";  
    boolean result1 = greeting.matches("Hello mum"); // true  
    boolean result2 = greeting.matches("Hello"); // false  
    System.out.printf("Result1=%b, Result2=%b\n", result1, result2);  
  
    String path = " Hello. How are you? I'm fine! Thanks!";  
    String[] pathComponents = path.split("[?!]");  
    for (String pc : pathComponents) {  
        System.out.println(pc);  
    }  
}
```

```
Result1=true, Result2=false  
Hello  
How are you  
I'm fine  
Thanks
```

RegularExpressions.java

27

To finish our discussion on regular expressions, we point out that the `String` class contains some rudimentary in-built support for regular expressions and pattern matching, via the following methods:

- `matches()`
- `split()`
- `replaceFirst()`
- `replaceAll()`

The example in the slide shows how to use these methods. For full information, see the JavaDoc documentation for the `String` class.

Any Questions?



28