Regular Expressions

Before I end this discussion on regular expressions, I want to swing back to the first two I showed you at the start of this section.

I had promised you'd understand these, or a lot of the parts, by the end of this section.

Patter for:	Regular Expression	Examples
U.S. Phone Number	`\\([0-9]{3}\\) [0-9]{3}-[0-9]{4}`	(800) 123-4567
HTML Tag	`<(\\w+)[^>]*>([^\\v</>]*)(</\\1>)*`	`<h1>Title</h1>` `<br/>` `<h2 class="red">Hello World</h2>`

Understanding the Phone Number Regular Expression

```
Pattern phonePattern =
        Pattern.compile("\\([0-9]{3}\\) [0-9]{3}-[0-9]{4}");
```

This pattern starts with two backslashes before an opening parentheses.

You'll remember parentheses are used in regular expressions as a meta character, identifying groups in most cases.

But in this case, I want a literal opening parentheses because I want to match a parentheses character in my string, so I need to escape it this way.

Understanding the Phone Number Regular Expression

```
Pattern phonePattern =
        Pattern.compile("\\([0-9]{3}\\) [0-9]{3}-[0-9]{4}");
```

Next, I'm using a character class that defines a range, in this case the digits, zero through 9, because I only want numbers.

The three in curly braces is a quantifier that says I want three digits here, after the opening parentheses.

Understanding the Phone Number Regular Expression

```
Pattern phonePattern =
        Pattern.compile("\\([0-9]{3}\\) [0-9]{3}-[0-9]{4}");
```

This is followed by a closing parentheses, and again, I need to escape it, for it to be used literally.

I've included a space character after that, which is a literal space, so it won't include all white space.

In other words it won't match on a tab for instance.

Understanding the Phone Number Regular Expression

```
Pattern phonePattern =
        Pattern.compile("\\([0-9]{3}\\) [0-9]{3}-[0-9]{4}");
```

I again have a character class in square brackets for the full range of digits, and that also has a quantifier meaning this has to be exactly 3 digits here.

This is followed by a dash, which doesn't need to be escaped in Java, and then again, digits, but ending in four digits this time.

Understanding the Phone Number Regular Expression

```
Pattern phonePattern =
        Pattern.compile("\\(*[0-9]{3}[)\\s]*[0-9]{3}-[0-9]{4}");
```

This character class, with an asterisk quantifier, will match either a closing parentheses or a space, zero or many times.

Alternate Character classes

	Range	Predefined character class	POSIX character classes (US-ASCII only)	java.lang.Character classes
Digits	`[0-9]`	`\d`	`\p{Digit}`	
Lower case letters	`[a-z]`		`\p{Lower}`	`\p{javaLowerCase}`
Upper case letters	`[A-Z]`		`\p{Upper}`	`\p{javaUpperCase}`
Both Upper and Lower Case	`[a-zA-Z]`		`\p{Alpha}`	`\p{javaUpperCase}`
All letters, digits and an underscore, called a word.	`[a-zA-Z_0-9]`	`\w`	`\p{Alnum}` ** Does not include underscore	
Spaces	`[ \t\n\x0B\f\r]`	`\s`	`\p{Space}`	`\p{javaWhitespace}`

This slide shows some common ranges, and some alternatives that mean the same or similar things.

The zero through 9 range can be replaced with either a backslash d, or the backslash p, and Digit spelled out in curly braces.

Alternate Character classes

	Range	Predefined character class	POSIX character classes (US-ASCII only)	java.lang.Character classes
Digits	`[0-9]`	`\d`	`\p{Digit}`	
Lower case letters	`[a-z]`		`\p{Lower}`	`\p{javaLowerCase}`
Upper case letters	`[A-Z]`		`\p{Upper}`	`\p{javaUpperCase}`
Both Upper and Lower Case	`[a-zA-Z]`		`\p{Alpha}`	`\p{javaUpperCase}`
All letters, digits and an underscore, called a word.	`[a-zA-Z_0-9]`	`\w`	`\p{Alnum}` ** Does not include underscore	
Spaces	`[ \t\n\x0B\f\r]`	`\s`	`\p{Space}`	`\p{javaWhitespace}`

The range with lowercase letters a to z, can be substituted with backslash p, and Lower spelled in curly braces.

If you need to support unicode characters, you can use the java.lang.Character classes.

There are similar predefined classes for upper case letters, and there is a combined class for US-ASCII alphabetical letters.

Alternate Character classes

	Range	Predefined character class	POSIX character classes (US-ASCII only)	java.lang.Character classes
Digits	`[0-9]`	`\d`	`\p{Digit}`	
Lower case letters	`[a-z]`		`\p{Lower}`	`\p{javaLowerCase}`
Upper case letters	`[A-Z]`		`\p{Upper}`	`\p{javaUpperCase}`
Both Upper and Lower Case	`[a-zA-Z]`		`\p{Alpha}`	`\p{javaUpperCase}`
All letters, digits and an underscore, called a word.	`[a-zA-Z_0-9]`	`\w`	`\p{Alnum}` ** Does not include underscore	
Spaces	`[ \t\n\x0B\f\r]`	`\s`	`\p{Space}`	`\p{javaWhitespace}`

Words are described by letters, digits and the underscore, as shown by the range.

Whitespace, as defined by backslash s, will match a literal space, a tab, a new line, and carriage returns, etc.

Understanding the HTML Tag Regular Expression

```
Pattern htmlPattern =
        Pattern.compile("<(\\w+)[^>]*>([^\\v</>]*)(</\\1>)*");
```

This expression starts with an angle bracket.

An angle bracket is a metacharacter when used if you're naming your group, but the regular expression processor is smart enough to check context, so I don't have to escape it here.

This means this regular expression will start matching when it finds the opening angle bracket of a tag.

Understanding the HTML Tag Regular Expression

```
Pattern htmlPattern =
        Pattern.compile("<(\\w+)[^>]*>([^\\v</>]*)(</\\1>)*");
```

Next, there's a group defined, which is automatically indexed as group 1.

Remember group 0 is the entire matching subsequence.

In this case, there's backslash backslash w plus in the parentheses.

The backslash w meta character means its a word character.

It's followed by the plus quantifier, which means it will match at least one word character but could match more.

Understanding the HTML Tag Regular Expression

```
Pattern htmlPattern =
        Pattern.compile("<(\\w+)[^>]*>([^\\v</>]*)(</\\1>)*");
```

After this I have a character class in brackets, that starts with a carat.

A carat in square brackets means to ignore the characters that follow, so in this case, a closing angle bracket will stop the matching.

This is followed by an asterisk, so any character that's not a closing angle bracket is going to match.

Understanding the HTML Tag Regular Expression

```
Pattern htmlPattern =
        Pattern.compile("<(\\w+)[^>]*>([^\\v</>]*)(</\\1>)*");
```

This is followed by the closing bracket for the tag.

This code will match tags that have no body, as well as tags that have attributes.

Understanding the HTML Tag Regular Expression

```
Pattern htmlPattern =
        Pattern.compile("<(\\w+)[^>]*>([^\\v</>]*)(</\\1>)*");
```

Another group follows.

This group's purpose is to capture the text that's contained between the opening and closing html tags.

You might expect this to be dot asterisk in parentheses, but there are some characters that we don't want to match on, so again I'll use a custom character class with a carat, so that any character that follows the carat will end the matching for this subsequence.

These characters are new lines or carriage returns, the opening and closing angle brackets and the backslash character.

Understanding the HTML Tag Regular Expression

```
Pattern htmlPattern =
        Pattern.compile("<(\\w+)[^>]*>([^\\v</>]*)(</\\1>)*");
```

Line breaks or carriage returns are called vertical white space.

After this I have the opening angle bracket, the backslash and the closing bracket.

This just means the code will match any character that's not one of these, and because I'm using the asterisk quantifier, this can be empty.

Understanding the HTML Tag Regular Expression

```
Pattern htmlPattern =
        Pattern.compile("<(\\w+)[^>]*>([^\\v</>]*)(</\\1>)*");
```

The final group is really only a group because I want to use a quantifier for the entire set.

Here, I have the opening angle bracket and a backslash.

This is how most html tags end, and then they're followed by the tag label, p, or h1 or whatever.

I'm using a back reference to the first group.

This means the ending label needs to match the opening label.

Understanding the HTML Tag Regular Expression

```
Pattern htmlPattern =
        Pattern.compile("<(\\w+)[^>]*>([^\\v</>]*)(</\\1>)*");
```

I'm also using a quantifier on this last group, meaning this could be omitted, as it will be in one of my examples, that uses the xhtml formatted br tag, that includes a backslash before the closing bracket.