# Text

**Characters**

**Strings and StringBuilders**

**Formatting and Parsing**

**Regular Expressions**



# Characters

**Concepts: Characters, Code Points, Encodings…**

**Unicode: A to Z… and more**





## Characters

- Step 1: A **character**
  - Abstract definition: "An atom of information"
  - Concrete examples:
    - A B C D
    - # " ! ( )
    - å ä ö ċ õ ĕ ķ Ħ Ŗ ǽ Ґ ы Ω Ą Δ Ж ï
    - € ℮ ∞ ∫ ℝ 𝓂 ∇ ∰ ∺ ≩ ⊛ ☺ ♠ ◆ ◬



## Character Repertoires

- Step 2: A **character repertoire**
  - Defines a set of available characters
  - ASCII: 128 characters, including 33 control characters
  - ISO 646-SE: Remove [|]{\}, add åäöÅÄÖ
  - ISO Latin-1: Adds 32 control chars, 96 symbols (west Europe)
  - ISO Latin-2: Adds 32 control chars, 96 symbols (east/central)
  - ISO Latin-3: Adds 32 control chars, 96 symbols (south Europe)
  - Unicode 4.0:
    - Graphic characters: 96,248
    - Formatting: 134
    - Control: 65
    - Private Use: 137,468
    - Surrogate: 2,048
    - Non-character: 66
    - Reserved: 878,083



## Coded Character Sets

- Step 3: A **coded character set**
  - Assigns an integer for each character in the repertoire
  - ASCII: values 0 to 127
    - 0 to 31 are control characters: 7 = bell, 9 = tab, 10 = line feed, …
    - 32 = space
    - 65 = A, 66 = B, 67 = C, …, 97 = a, 98 = b, 99 = c, …

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | |



## Coded Character Sets (2)

- **ISO Latin-1**: ASCII, with additional chars at 128-255
- **EBCDIC** (IBM mainframes):
  - 129 = a, 130 = b, …, 137 = i, 145 = j, …

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | ¢ | . | < | ( | + | \| |
| & | | | | | | | | | | ! | $ | * | ) | ; | ¬ |
| - | / | | | | | | | | | ¦ | , | % | _ | > | ? |
| | | | | | | | | | ` | : | # | @ | ' | = | " |
| | a | b | c | d | e | f | g | h | i | | | | | | |
| | j | k | l | m | n | o | p | q | r | | | | | | |
| | ~ | s | t | u | v | w | x | y | z | | | | | | |
| | | | | | | | | | | | | | | | |
| { | A | B | C | D | E | F | G | H | I | | | | | | |
| } | J | K | L | M | N | O | P | Q | R | | | | | | |
| \ | | S | T | U | V | W | X | Y | Z | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | |



## Coded Character Sets (3)

- **Unicode** – some examples:
  - 0000..001F <control-0000>..<control-001F>
  - 0020..007E SPACE..TILDE
  - 007F..009F <control-007F>..<control-009F>
  - 00A0..00AC NO-BREAK SPACE..NOT SIGN
  - 03D0..03D6 GREEK BETA SYMBOL..GREEK PI SYMBOL
  - 05B0..05B9 HEBREW POINT SHEVA..HEBREW POINT HOLAM
  - 0A01 GURMUKHI SIGN ADAK BINDI ਈਉਊਏਓਔਕਖਗਘਙ
  - 0EDC..0EDD LAO HO NO..LAO HO MO ໜໝ
  - 2302..237A HOUSE..APL FUNCTIONAL SYMBOL ALPHA ⌂⌘⌦⌨⍝⍯
  - 2E80..2E99 CJK RADICAL REPEAT..CJK RADICAL RAP
  - 33E0 IDEOGRAPHIC TELEGRAPH SYMBOL FOR DAY ONE
  - 1039F UGARITIC WORD DIVIDER
  - 100000..10FFFD <private-use-100000>..<private-use-10FFFD>



## Character Encodings

- Step 4: **Character encoding**
  - How are characters stored in a computer?
    - Coded character set defines an integer for each character set
    - This integer must be coded into one or more bytes!
  - If there are at most 128 characters: Simple
    - 128 characters will fit in 7 bits
    - Use the lower 7 bits in a byte
    - Everything is assumed to support at least 7 bits



## Character Encodings (2)

- With **at most 256** characters: Also simple – or not?
  - 256 characters will fit into 8 bits = 1 byte
  - Modern software is mostly "8-bit clean" – use one byte/char
  - Legacy software may only be "7-bit clean"
    - Corrupts or misinterprets the 8th bit
    - Requires a different "transport encoding"
  - URL and MIME encodings: 256 characters, using bytes 0..127
    - URL: Encodes characters >127 as %XY (hexadecimal)
    - MIME Quoted-Printable: >127 as =XY (hexadecimal)
  - HTML: Character encoding specified in header
    - <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
    - Also supports entities: "‰" = "‰" (Unicode 0x2030)

## Character Encodings (3)



- **Unicode**: Tens of thousands of characters!
  - Won't even fit into 16 bits…
  - Two **fixed-length encodings**:
    - UCS-2: One 16-bit word per character (does not support all chars!)
    - UCS-4: One 32-bit word per characters
  - Problem: Completely unsupported by most legacy software
    - ASCII: "Hello" coded as 48 65 6c 6c 6f
    - UCS-2: "Hello" coded as 00 48 00 65 00 6c 00 6c 00 6f
    - Contains "00" bytes which C programs interpret as "end of string"
  - Problem: Inefficient for English and English-like languages
    - Twice or four times as many bytes

## Character Encodings (4)



- Three official **variable-length encodings**
  - UTF-8: Each character stored as 1 to 6 bytes
    - U-00000000 - U-0000007F: 0*xxxxxxx* *(same as ASCII!)*
    - U-00000080 - U-000007FF: 110*xxxxx* 10*xxxxxx*
    - U-00000800 - U-0000FFFF: 1110*xxxx* 10*xxxxxx* 10*xxxxxx*
    - U-00010000 - U-001FFFFF: 11110*xxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx*
    - U-00200000 - U-03FFFFFF: 111110*xx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx*
    - U-04000000 - U-7FFFFFFF: 1111110*x* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx* 10*xxxxxx*
  - UTF-16: Each character stored as one or two 16-bit words
  - UTF-32: Each character stored as one 32-bit word
- Which one to choose?
  - UTF-8 most widely supported
  - Efficient for English and similar languages: Chars fit into 1 byte
- Example for UTF-8:
  - R**ä**ksm**ö**rg**å**s: 72 **c3 a4** 6b 73 6d **c3 b6** 72 67 **c3 a5** 73
  - In ISO Latin-1, this byte sequence would mean "rÃ¤ksmÃ¶rgÃ¥s"

# Text in Java

**Characters**

**Strings**

**StringBuilders**



## Characters 1: Unicode



- Java uses **Unicode**!
  - Up to Java 1.3: Unicode 2.1, with 38,807 distinct symbols
  - From Java 1.4: Unicode 3.0, with 49,194 distinct symbols
  - From Java 1.5: Unicode 4.0, with 96,248 distinct symbols
- Internally, Java uses 16 bits
  - Java 1.3 and 1.4: Direct mapping (1 char = 1 code point)
  - Java 1.5: UTF-16 (may require two "chars" for one character)

More information: http://www.unicode.org

## Characters 2: Character Literals



- **Character literals** are written using single quotes
  - char c = 'x', c2 = '7';
- But chars are represented as numbers…
  - char c = 120; // Makes c an 'x', since in Unicode, character #120 is an 'x'.
- **Escape codes** represent unprintable characters
  - '\b': Backspace<br>'\t': Tab<br>'\n': Newline<br>'\f': Form feed<br>'\r': Carriage return<br>'\'': Single quote (quote backslash quote quote)<br>'\\': A backslash<br>'*nnn*': The character with octal value *nnn* ('\012' == '\n')<br>'\u*nnnn*': The character with hex value nnnn ('\u03D0' == greek beta)

## Characters 3: Class java.lang.Character



- **Static utility methods** in java.lang.Character
  - Character classes
    - isLowerCase(), isUpperCase(), isTitleCase(), isDigit(), isLetter(), isLetterOrDigit(), isWhitespace(), …
    - Use these methods to accept greek lower case, kyrillic upper case, non-arabic digits, and so on
    - Character ch; ...; **if** (ch.isLetter()) { ... }
  - Case conversion
    - toLowerCase(), toUpperCase(), toTitleCase()
  - Conversion from and to decimal/hex/…
    - Character.digit('7', 10) = 7<br>Character.digit('b', 16) = 11
    - Character.forDigit(11, 16) = 'b'
    - Character.getNumericValue('7') = 7<br>Character.getNumericValue('\u216c') = 50 (roman numeral)

The letter 'ǉ':
Lower case 'ǉ'.
Upper case 'Ǉ'.
Title case 'ǈ'.

## Strings 1: Strings in Java



- Two **main classes** representing strings
  - String: Constant strings
  - StringBuilder: Buffers used for building / concatenating strings
    - Temporarily – then you make a String from the StringBuilder
    - Used implicitly for the "+" operator on Strings
- Also other related classes
  - StringWriter
    - I/O class; what you write is appended to a buffer; retrieve as a String
  - CharBuffer
    - I/O class in the "new I/O" system
  - StringBuffer
    - Old version of StringBuilder

## Strings 2: String interfaces in Java



- Two interfaces characterize **generic functionality**
  - ***Appendable***: Anything you can append ("write") text to
    - StringBuilder, StringWriter, CharBuffer
    - PrintWriter (System.out), FileWriter, BufferedWriter,
    - Not String – it is constant
  - ***CharSequence***: Anything you can get ("read") text from
    - (Random access required: char charAt(int index))
    - String, StringBuffer, CharBuffer
    - Not StringWriter

## Strings 2: Literals



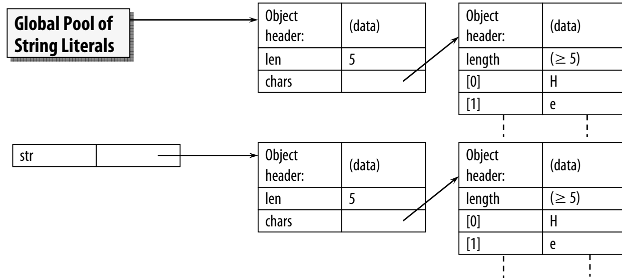
- String literals are written within **double quotes**
  - Escape codes can be used
    - "Hello, World"
    - "This is the first line\nThis is the second line, with a \" quote char"
- String literals are **String objects**
  - They are not char arrays, but can be created from char arrays
    - String str = { 'a', 'b', 'c' }; // WRONG<br>char[] array = "abc"; // WRONG
    - char[] array = { 'a', 'b', 'c' }; // OK<br>String str = **new** String(array); // OK
  - No need to assign them to variables; they **are** objects
    - String str = "Hello";<br>System.out.println(str.length()); // 5
    - System.out.println("Hello".length()); // 5

Call a method in the string literal!

# Strings 3: Literals and Constructors



- No need to use a constructor for a literal
  - String str = **new** String("Hello"); // NOT necessary; inefficient!
    String str = "Hello"; // Better
- The String literal "Hello" was pre-allocated
  - Calling the constructor creates a copy




# Strings 4: Useful methods



- Checking the length of a String
  - "foo".length() == 3
- Retrieving part of a String
  - charAt(int pos) returns a specific character
    - "a**b**cdefgabcdefg".charAt(1) == 'b'
  - substring(startindex, endindex) (endindex is exclusive)
  - trim() returns a new String without initial / trailing whitespace
  - split(String regexp) splits around regexp matches
    - "foo**:**bar**::**gazonk".split(":") == {"foo", "bar", "", "gazonk"}
    - "f**oo**:bar::gaz**o**nk".split("o*") == {"f", ":bar::gaz", "nk"}

# Strings 5: Useful methods



- String conversions
  - toUpperCase(Locale), toLowerCase(Locale)
  - toUpperCase(), toLowerCase() (uses default locale)
- Finding matches
  - Finding a matching substring
    - "ab**cde**fgabcdefg".indexOf("cde") == 2
    - "abcdefgab**cde**fg".indexOf("cde", 5) == 9
    - "abcdefgab**cde**fg".lastIndexOf("cde") == 9
  - startsWith(), endsWith():
    - "**abc**defg".startsWith("abc") == true
  - Does the string match a regular expression? [more later]
    - "aaaabb".matches("a*b*") == true

# Strings 6: String Comparisons



- Comparing Strings
  - equals(String other), equalsIgnoreCase(String other)
    - True if the strings are equal, false otherwise
    - As usual, == tests whether they are stored in the same location!
  - compareTo(String other), compareToIgnoreCase(String other)
    - -1 if this before other, 0 if equal, 1 if this after other
- Different countries have different sorting rules
  - Comparing á to a, sorting ß as ss,
    sorting ä as a-with-umlaut (German) or after å (Swedish) …
  - Use Collator, RuleBasedCollator for searching and sorting
    - Collator collator = Collator.getInstance(); // Locale-dependent
      Collator collator2 = Collator.getInstance(Locale.CANADA);
      collator.compare("abc", "ABC") // -1, 0, 1

# Formatting Text

**Converting primitive types to text**

**Converting single objects to text**

**String concatenation**

**Advanced formatting**



# Formatting 1: Primitive Datatypes to Strings



- Convert **primitive datatypes to text:**
  - Use the static String.valueOf() methods
    - **static** String valueOf(char c) { … }
      **static** String valueOf(int i) { … }
      **static** String valueOf(boolean b) { … }
      …
    - String str = String.valueOf(127); // "127"

# Formatting 2: Objects to Strings



- Every object has a **string representation**
  - java.lang.Object has a toString() method
    - This method returns a String representing the object
    - System.out.println(Object o) prints o.toString()
  - Example:

```
public class Circle {
   private double x, y, r;
   private Circle(double x, double y, double r) {
      this.x = x; this.y = y; this.r = r;
   }
   public static void main(String[] args) {
      System.out.println(new Circle(2.0, 4.0, 5.0));
   }
}
```

```
>> java Circle
Circle@a5e17120
```

# Formatting 3: Objects to Strings



- Why "Circle@a5e17120"?
  - Object.toString() returns the class name + "@" + a unique ID
  - The method should be overridden!
- Example:

```
public class Circle {
   …
   public String toString() {
      return "Circle[" + x + ", " + y + ", " + r + ']';
   }
   public static void main(String[] args) {
      System.out.println(new Circle(2.0, 4.0, 5.0));
   }
}
```

```
java Circle
Circle[2.0, 4.0, 5.0]
```

# Formatting 4: String Concatenation



- In the previous slide: "Circle[" + x + ", " + y + ", " + r + ']'
  - Java overloads the + operator for string concatenation.
  - Internally, concatenation uses the StringBuilder class
  - The translation is done by Java compiler (javac, jikes, …)

```
final StringBuilder buf = new StringBuilder("");
buf.append("Circle[");
buf.append(x);      → String.valueOf(double)
buf.append(", ");
buf.append(y);      → String.valueOf(double)
buf.append(", ");
buf.append(r);      → String.valueOf(double)
buf.append(']');    → String.valueOf(char)
return buf.toString();
```

The valueOf methods know how to convert primitive datatypes to Strings!

## Formatting 5: String Concatenation



- What about constant strings?
  - String str = "hi " + "there"
  - This is evaluated at compile time!
    - Exactly the same bytecode as String str = "hi there"
- What about objects?
  - ```
    Circle c = new Circle(2.0, 4.0, 5.0);
    String str = "The circle is " + c;     // Concatenate String + Object
    ```
  - This is translated, and becomes…
  - ```
    final StringBuilder buf = new StringBuilder("");
    buf.append("The circle is ");
    buf.append(c); // ➔ String.valueOf(c) ➔ c.toString()
    String str = buf.toString();
    ```

## Formatting 6: StringBuilder Example



- Using **StringBuilder** in toString():
  - In loops, better to use StringBuilder explicitly
    - ```
      public class ArrayList {
          private Object[] elements;
          private int size;
          …
          public String toString() {
            final StringBuilder buf = new StringBuilder();
            buf.append("ArrayList<");
            for (int i = 0; i < size; i++) {
              if (i != 0) buf.append(", ");
              buf.append(elements[i].toString());
            }
            buf.append(">");
            return buf.toString();
          }
      }
      ```

Using String and += would be **extremely** inefficient: The string is copied twice each concatenation!

## Advanced Text Formatting



- Use a **Formatter** for advanced text formatting
  - Step 1: Create a formatter (possibly specifying a locale)
    - ```
      // Send all output to the Appendable object sb
      StringBuilder sb = new StringBuilder();
      Formatter formatter = new Formatter(sb, Locale.US);
      ```
  - Step 2: Format text using the formatter (similar to printf in C)
    - Each call specifies a format string and any number of arguments
    - Each call appends to the given appendable
    - ```
      formatter.format("Hello, %s!\n", person.name);
      formatter.format("You are %d years old.\n", person.age);
      ```
    - ```
      // Minimum 30, maximum 40 characters (padded with spaces)
      formatter.format("Hello, %30.40s!\n", person.name);
      ```
    - ```
      // Here we specify an explicit locale…
      formatter2.format(Locale.FRANCE, "e = %+10.4f", Math.E);
      // -> "e =    +2,7183"
      ```

## Advanced Text Formatting 2



- Often no need to explicitly create a Formatter
  - Use utility methods instead
  - In PrintStream: printf()
    - System.out.printf("Hello, %s!\n", person.name);
  - In String: format()
    - ```
      Calendar c = new GregorianCalendar(1995, MAY, 23);
      String s = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);
      // -> s == "Duke's Birthday: May 23, 1995"
      ```

# Parsing Text



## Parsing 1: Strings to Primitive Datatypes



- Convert **text to data** using a number of methods
  - Depending on data type
  - Every data type (int, float, …) has a corresponding class
    - In class Integer: **static** int parseInt(String str);
    - In class Float: **static** float parseFloat(String str);
    - In class Double: **static** double parseDouble(String str);
    - …
    - float val = Float.parseFloat("127");     // 127.0f

## Parsing 2: Tokenizing Formal Languages



- Tokenizer: **java.util.StringTokenizer**
  - Breaks strings into words given break characters
    - ```
      final StringTokenizer st;
      st = new StringTokenizer("simple test\nLine 2", " \t\r\n");
      while (st.hasMoreTokens()) {
         println(st.nextToken());
      }
      ```
      Space, tab, CR, newline
    - simple<br>test<br>Line<br>2
  - Like String.split(), but more efficient and more restricted
    - String[] tokens = "simple test\nLine 2".split("( |\t|\r|\n)");
  - See also:
    - StreamTokenizer – to find tokens in Java-like languages
    - BreakIterator – for natural language text

# Specifying Character Encodings

**Different software, different encodings…**

**How do we specify which encoding is used?**



## Encodings 1: HTML



- HTML: **Character Repertoires**
  - HTML 1.0 to 3.2: Only supports ISO Latin-1
  - HTML 4.0 and later: UCS (same chars as Unicode 2.0)
- HTML 4.0: Specifying **Character Encodings**
  - Encoding header in HTTP
    - Content-Type: text/html; charset=ISO-8859-1
    - Content-Type: text/html; charset=utf-8
    - Encoding header comes before the HTML file
    - Requires server configuration
  - Encoding meta tag in HTML file
    - <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
    - Up to this header, all characters interpreted as ASCII
  - Regardless of encoding: &amp;#12345; is Unicode char 12345 (dec)
      ```

## Encodings 2: Files



- **File I/O**: Two parallel subsystems in Java
  - Binary I/O: InputStream, OutputStream (pure 8-bit bytes)
    - FileInputStream, DataInputStream, BufferedInputStream, …
  - Text I/O: Reader, Writer (converts bytes to/from chars
    - FileReader, StringReader, BufferedReader, …
    - By default: Uses platform-specific local character set
    - **final** OutputStream os = **new** FileOutputStream("file.txt");
      **final** Writer wr = **new** OutputStreamWriter(os, "US-ASCII");
    - **final** OutputStream os = **new** FileOutputStream("file.txt");
      **final** Writer wr = **new** OutputStreamWriter(os, "ISO-8859-1");
    - **final** Reader re = **new** InputStreamReader(is, "UTF-16");
  - This is true for standard output too!
    - System.out converts chars to bytes

## Encodings 3: Java GUI



- **GUI character support** depends on platform/font
  - Regardless of method used
    - myTextField.setString("å ä ö ċ õ ĕ ķ Ħ Ŗ ǽ Ґ ы Ω Ą Δ Ж ΐ");
    - void paintComponent(Graphics g) {
      g.drawString(10, 100, "å ä ö ċ õ ĕ ķ Ħ Ŗ ǽ Ґ ы Ω Ą Δ Ж ΐ");
      }
  - Any Java environment can display ISO Latin-1
  - For more characters: May have to modify font.properties
    - Add new fonts that handle parts of the Unicode character set
    - For example:
      sansserif.3=Lucida Sans Unicode,UNICODE_CHARSET

## Encodings 4: Databases



- **JDBC**: Depends on database and drivers
  - MySQL: Many character encodings ("sets") and collations
    - SHOW CHARACTER SET; SHOW COLLATION;
  - Default character encoding for DB, table, connection
    - (CREATE|ALTER) DATABASE *db_name*
      [[DEFAULT] CHARACTER SET *charset_name*]
      [[DEFAULT] COLLATE *collation_name*]
    - CREATE TABLE *tbl_name* (*column_list*)
      [DEFAULT CHARACTER SET *charset_name* [COLLATE *collation_name*]]
    - ALTER TABLE *tbl_name* [DEFAULT CHARACTER SET *charset_name*]
      [COLLATE *collation_name*]
    - *col_name* {CHAR | VARCHAR | TEXT} (*col_length*)
      [CHARACTER SET *charset_name* [COLLATE *collation_name*]]
  - SELECT k FROM t1
    ORDER BY k COLLATE latin1_german2_ci;

# Patterns and Regular Expressions

**See also: Java Tutorial chapter on regular expressions**

http://java.sun.com/docs/books/tutorial/extra/regex/



## Glob Patterns



- Filename pattern matching (**globbing**):
  - "?" matches any single character
  - "*" matches any sequence of characters
    - *.java
  - Maybe a few more patterns – but not very powerful
- **Regular Expressions**: Much more powerful patterns
  - Directly supported in Java
  - Syntax as in Perl 5 (mostly)

## Regexps 1: Strings



- **Simplest regular expressions**: Ordinary strings
  - hydro
    - matches "hydro", "hydrogen", "hydrodynamics"
  - top ten
    - matches "top ten", "stop tension"
    - does not match "stop tension" (additional spaces)

## Regexps 2: Period



- **Metacharacters** have special meaning
  - Period (".") matches any single character
  - 2,.-dimethylbutane
    - Matches "2,2-dimethylbutane" or "2,3-dimethylbutane"
    - Does not match "2,12-dimethylbutane"
  - 3.14
    - Matches "3.14" or "3x14" or "3714" or "3+14" or "192753514127"
  - 3\.14
    - Matches "3.14" or "73.14159" but not "3x14" or "3714"
    - Metacharacter is escaped using a backslash (also metacharacter)
    - Note: Must write test.callMethod("3\\.14") --
      two backslashes required in a string literal,
      so that one remains in the actual string!

## Regexps 3: Character classes



- […] matches a **character class**
  - Examples
    - [abc] either a or b or c
    - [a-f] a single character between a and f
    - [a-fy-z] a, b, c, d, e, f, y, or z
    - [^abc] anything except a, b, or c
    - [-abc] hyphen or a (hyphen must be first – otherwise marks a range
    - [fs]inger "finger" or "singer"
  - Some predefined character classes
    - \d digit [0-9]
    - \s whitespace [ \t\n\x0B\f\r]
    - \w word character [a-zA-Z_0-9]
    - \p{Punct} punctuation One of !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
    - \p{XDigit} hex digit [0-9a-fA-F]
    - Pattern.compile("\\s\\w"); // white space followed by word char

## Regexps 4: Alternatives



- x|y marks a set of **alternatives**
  - finger|singer
  - html|head|body|title|h[1-6]|p|hr|font|ul|li
    - Some HTML tag names
- May need to place within **parentheses**
  - <html|head|body|title|h[1-6]|p|hr|font|ul|li>
    - matches "<html", "head", …, "li>"
  - <(html|head|body|title|h[1-6]|p|hr|font|ul|li)>
    - matches "<html>", "<head>", …, "<li>"

## Regexps 5: Question mark



- **Question mark**: Expression should occur 0 or 1 time
  - m?ethane
    - matches "methane" or "ethane" or "asdfethanekasd"
  - comm?a
    - matches "coma" or "comma"
  - [fs]?inger
    - matches "finger", "singer", or "inger"
  - (html|body|head)?
  - NOTE: Different from "?" in filename matching
    - In filename matching, "?" matches one char – same as "." here

## Regexps 6: Star, Plus, Number of times



- **Star**: Expression should occur zero or more times
  - ab*c
    - matches "ac", "abc", "abbc"
- **Plus**: Expression should occur one or more times
  - ab+c
    - matches "abc", "abbc", "acabbc82", … (but not "ac")
- **{n,m}**: Expression should occur n to m times
  - ab{2,4}c
    - matches "abbc", "abbbc", "abbbbc"
    - does not match "abc"
    - does match "abcabbbcababacab"

## Regexps 7: Start and end of text



- Regexps can **occur anywhere** in a string
  - Use special metacharacters to match start/end
    - ^ matches start (of string or line, depending on settings)
    - $ matches end (of string or line, depending on settings)
  - m?ethane
    - matches "methane" or "ethane" or "asdfethanekasd"
  - ^m?ethane
    - matches "methane" or "ethane" or "ethanekasd"
    - does not match "asdfethanekasd"
  - ^m?ethane$
    - matches "methane" or "ethane"
    - does not match anything else

## Regexps 8: Combining metacharacters



- **Combining metacharacters**
  - Metacharacters can be combined
  - Common combination: Period and star .*
    - matches any number of occurrences of any character
    - same as a single "*" in filename globbing
  - cyclo.*ane
    - cycloane
    - cyclodecane, cyclohexane
    - cyclones drive me insane, did you know that?
  - [^>]*
    - matches any sequence of chars not containing ">"
  - <h1[^>]*>
    - matches "<h1", followed by anything except ">", followed by ">"
    - ^<h1.*>$ would have matched "<h1>Hello</h1><p>Some text</p>"

## Regular Expressions in Java (1)



- In Java: First, **create and compile** a pattern:
  - Compiling takes time – but the result can be reused
    - Pattern pat = Pattern.compile("^Hello.*world$");
    - For constant patterns: Store as a static constant!
  - Compiled to bytecode!
    - Creating classes on the fly
    - Very efficient pattern matching
- Then, use it to **match** one or more strings:
  - Matcher mat = pat.matcher("foo");
    **if** (mat.matches()) { … }
  - Matcher mat2 = pat.matcher("Hello, wonderful world");
    **if** (mat2.matches()) { … }

## Regular Expressions in Java (2)



- **Example**: Matching lines in a price list
  - Something: 100.00
    Something else with a long name: 199.50
  - Pattern:
    - One or more non-colon characters
    - Colon
    - Space
    - Any combination of digits/periods
  - Some sub-patterns:
    - One non-colon char — [^:]
      At least one non-colon char — [^:]+
      Any number of digits/periods — [0-9.]+
  - Complete pattern: [^:]+: [0-9.]+

## Regular Expressions in Java (3)



- Parentheses generate **capturing groups**
  - What you match inside parentheses can be extracted
  - Use this to parse the price list
    - Something: 100.00
      Something else with a long name: 199.50
  - Old pattern: [^:]+: [0-9.]+
  - Add parentheses for subgroups: ([^:]+): ([0-9.]+)
  - Code:

```
final Pattern pat = Pattern.compile("([^:]+): ([0-9.]+)");
final Matcher mat = pat.matcher(myEntireTextTable);
while (mat.matches()) {
    System.out.println(mat.group(1)); // What was within the first ()
    System.out.println(mat.group(2)); // What was within the second ()
}
```

## Regular Expressions in Java (4)



- **Another example**
  - Extract all references from an HTML page:
    - <a href="http://www.ida.liu.se">The IDA Home Page</a>
    - Match <a …whatever… href="…" …whatever…>
  - Need to match:

| | |
|---|---|
| <a | Literally: left-bracket and "a" |
| [^>]* | Any character except end-of-tag ">", as many times as you need |
| href=" | The href value… |
| ([^"]*) | A string of characters, not including quote |
| " | Closing quote |
| [^>]*> | A sequence of non-">" followed by a ">" |

```
final Pattern pat = Pattern.compile("<a[^>]*href=\"([^\"]*)\"[^>]*>");
final Matcher mat = pat.matcher(myEntireHTMLDocument);
while (mat.matches()) System.out.println(mat.group(1));
```

## Regular Expressions in Java (5)



- **Replacing subsequences**
  - $1 refers to first capturing group, $2 to second, etc
  - Swap two arguments of a macro:
    - \foo 100,200 → \foo 200,100
    - Pattern to match: \\foo [0-9]+,[0-9]+
    - With capturing groups: \\foo ([0-9]+),([0-9])+
    - Replace with: \\foo $2,$1

```
final Pattern pat = Pattern.compile("\\\\foo ([0-9]+),([0-9])+");
final Matcher mat = pat.matcher("…");
final String replaced = mat.replaceAll("\\\\foo $2,$1");
```