

# Java Characters and Strings

## Object Oriented Programming



**SoftEng**  
<http://softeng.polito.it>

Version 1.1.1  
© Marco Torchiano, 2020



1



This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license, visit  
<http://creativecommons.org/licenses/by-nc-nd/4.0/>.

You are free: to copy, distribute, display, and perform the work

Under the following conditions:



**Attribution.** You must attribute the work in the manner specified by the author or licensor.



**Non-commercial.** You may not use this work for commercial purposes.



**No Derivative Works.** You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

2

## Primitive types

Type	Size	Encoding
<code>boolean</code>	1 bit	–
<code>char</code>	16 bits	Unicode UTF16
<code>byte</code>	8 bits	Signed integer 2C
<code>short</code>	16 bits	Signed integer 2C
<code>int</code>	32 bits	Signed integer 2C
<code>long</code>	64 bits	Signed integer 2C
<code>float</code>	32 bits	IEEE 754 sp
<code>double</code>	64 bits	IEEE 754 dp
<code>void</code>	–	

3

3

## Literals

- Literal values for chars and strings follow the C syntax
  - ♦ Non printable characters are introduced by a `'\'` backslash
  - ♦ Chars
    - `'a'` `'%'` `'\n'`
  - ♦ Strings
    - `"prova"` `"prova\n"`

4

4

# Characters and Strings

---

- Characters
    - ♦ Primitive type **char**
    - ♦ Wrapper class **Character**
  - String
    - ♦ No primitive representation!
    - ♦ Class **String**
    - ♦ Class **StringBuffer**
      - and class **StringBuilder**
- 

5

---

## CHARACTERS

---

6

## Wrapper Character

---

- Encapsulates a single character
  - ♦ Immutable (like all wrapper classes)
- Utility methods for the char category
  - ♦ `isLetter()`
  - ♦ `isDigit()`
  - ♦ `isSpaceChar()`
- Utility methods for conversions
  - ♦ `toUpperCase()`
  - ♦ `toLowerCase()`

---

7

7

## Unicode

---

- Standard that assigns a unique code to every character in any language
  - ♦ **Core specification** gives the general principles
  - ♦ **Code charts** show representative glyphs for all the Unicode characters.
  - ♦ **Annexes** supply detailed normative information
  - ♦ **Character Database** normative and informative data for implementers

---

<http://www.unicode.org/versions/latest/>

8

## Characters and Glyphs

---

- **Character**: the abstract concept
    - ♦ e.g. `LATIN SMALL LETTER I`
  - **Glyph**: the graphical representation of a character
    - ♦ e.g. `ı / ĩ ï`
  - **Font**: a collection of glyphs
- 

9

## Unicode Codepoint

---

- **Codepoint**: the numeric representation of a character
    - ♦ Included in the range 0 to  $10FFFF_{16}$  (23 bits)
    - ♦ Represented with `U+` followed by the hexadecimal code
    - ♦ e.g. `U+0069` for 'i'
- 

10

## Unicode Encoding

---

Mapping between byte sequence and code point.

- **UTF-32** fixed width, 32 bits per char
    - ♦ Almost 23 used: memory occupation
  - **UTF-16** variable width, represents
    - ♦ codepoints from `U+0` to `U+d7ff` on 16 bits (2 bytes)
    - ♦ codepoints from `U+10000` to `U+10ffff` on 32 bits (4 bytes)
- 

11

## Unicode Encoding

---

- **UTF-8** variable width,
    - ♦ codepoints `U+00` to `U+7f` are mapped directly to bytes,
      - i.e. ASCII transparent
    - ♦ High bit (0x80) marks multi byte code
    - ♦ Most non-ideographic codepoints are represented on 1 or 2 bytes
      - e.g. `U+00C8` representing character 'è' is mapped to two bytes: `0xC3` `0xA8`.
- 

12

## Character set

---

- Class **Charset** allows handling different charsets
  - A few static methods
    - ♦ **defaultCharset()**
    - ♦ **forName(..)**
      - Returns the corresponding charset
    - ♦ **availableCharsets()**
      - Returns a map of all charsets by name
- 

13

## Predefined charsets

---

- **US-ASCII**
    - ♦ 7-bit ASCII, a.k.a. *ISO646-US*
  - **ISO-8859-1**
    - ♦ 8-bit single byte ISO Latin No. 1, a.k.a. *ISO-LATIN-1*
  - **UTF-8**
    - ♦ 8-bit multi byte UCS Transformation Format
  - **UTF-16BE**
    - ♦ 16-bit UCS Transformation Fmt., big-endian
  - **UTF-16LE**
    - ♦ 16-bit UCS Transformation Fmt., little-endian
  - **UTF-16**
    - ♦ 16-bit UCS Transformation Fmt., w/byte-order mark
- 

14

# Encoding and Decoding


---

- Convenience methods
    - ♦ `CharBuffer decode(ByteBuffer)`
    - ♦ `ByteBuffer decode(CharBuffer)`
  - Generation of codecs
    - ♦ `getDecoder()`
    - ♦ `getEncoder()`
    - Warning: decoder and encoder have an internal state
      - e.g. awaiting next byte of a multi-byte representation
- 

15

# Encoding mismatch

---

- Using an encoding scheme to decode a string encoded with a different scheme
  - E.g.
    - ♦ Character 'è' has Unicode codepoint `U+00C8` which is mapped in UTF-8 to two bytes: `0xC3` `0xA8`, while ISO-8859-1 decoding interprets the above sequence as the two characters 'Ã'
    - ♦ Viceversa, 'è' in ISO-8859-1 is represented as `0xE8` which is an invalid character in UTF-8 (usually represented as )
- 

16



---

# STRINGS

---

17

## String and StringBuffer

---

- Class **String**
  - ♦ Not modifiable / Immutable
- Class **StringBuffer** / **StringBuilder**
  - ♦ Modifiable / Mutable

```
String s = new String("literal");  
StringBuffer sb = new StringBuffer("lit");
```

---

18

18

## Operator +

- It is used to **concatenate** 2 strings

```
"This is " + "a concatenation"
```

- ♦ Remember: strings are immutable, therefore + creates a new string object with the concatenation

- Works also with other types

- ♦ Everything is automatically converted to a string representation and concatenated

```
System.out.println("pi = " + 3.14);  
System.out.println("x = " + x);
```

19

19

## String methods

- **int length()**
  - ♦ returns string length
- **boolean equals(String s)**
  - ♦ compares the contents of two strings

```
String h = "Hello";  
String w = "World";  
String hw = "Hello World";  
String h_w = h + " " + w;  
hw.equals(h_w)    // -> true  
hw == h_w         // -> false
```

20

20

## String methods

---

- **String toUpperCase()**
  - ♦ Converts string to upper case
- **String toLowerCase()**
  - ♦ Converts string to lower case
- **String concat(String str)**
  - ♦ Creates a concatenation with the given string
- **int compareTo(String str)**
  - ♦ Compare to another string returning
    - < 0 : if this string precedes the other
    - == 0 : if this string equals the other
    - > 0 : if this string follows the other

21

21

## Method subString

---

- **String subString(int startIndex)**  
`"Human".subString(2) → "man"`
- **String subString(int start, int end)**
  - ♦ Char *start* included, *end* excluded  
`"Greatest".subString(0, 5) → "Great"`
- **int indexOf(String str)**
  - ♦ Returns the index of the first occurrence of *str*
- **int lastIndexOf(String str)**
  - ♦ The same as before but search starts from the end

22

22

## String (*static* methods)

- **String** **valueOf**(...)
  - ♦ Converts any primitive type into a **String**
  - ♦ Overloads defined for all primitive types
- **String** **format**(**String** fmt, ...)
  - ♦ Builds a string using the format string
  - ♦ Similar format as C printf()

23

23

## Format essentials

Start at 1

Min width

Max width or decimal digits for floats

**%**[arg\_index\$][flags][width][.prec]conversion

F	Result
-	left justified
+	include sign
0	0 padding
(	Neg in parenthesis

C	Conversion
b	boolean
s	string
d	integer
f	decimal
e	scientific

24

24

## StringBuffer

---

- Represents a string of characters
  - It is **mutable** and allows operation that modify the content
  - Can be converted to the corresponding **String** using the method **toString()**
- 

25

## StringBuffer

---

- **append(String str)**
  - ♦ Inserts **str** at the end of string
- **insert(int offset, String str)**
  - ♦ Inserts **str** starting from **offset** position
- **delete(int start, int end)**
  - ♦ Deletes character from **start** to **end** (excluded)
- **reverse()**
  - ♦ Reverses the sequence of characters

They all return a **StringBuffer** enabling chaining

---

26

26

## Performance issues

```
String s="";  
  
for(i=0; i<N; ++i){  
    s += i;  
}
```

12 sec  
N = 100k

```
StringBuffer sb=  
    new StringBuffer();  
for(i=0; i<N; ++i){  
    sb.append(i);  
}
```

2 ms  
N = 100k

Three order of magnitudes difference!

27

## Class **StringBuilder**

- Method-level compatible with **StringBuffer**
- Non thread safe and non reentrant
- More efficient: ~25% faster

28

## String pooling

---

- Class String maintains a private static pool of distinct strings
  - Method **intern()**
    - ♦ Checks if any string in the pool *equals()*
    - ♦ If not, adds the string to the pool
    - ♦ Returns the string in the pool
  - For each string literal the compiler generates code using `intern()` to keep a single copy of the string
- 

29

## String internalization

---

```
public static final void main() {  
    char chars[] = {'H', 'i'};  
    String s1 = new String(chars);  
    String s2 = new String(chars);  
    String i1 = s1.intern();  
    String i2 = s2.intern();  
}
```

---

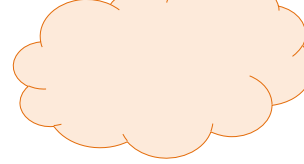
30

30

# String internalization

```
char chars[] = {'H', 'i'};  
String s1 = new String(chars);  
String s2 = new String(chars);  
String i1 = s1.intern();  
String i2 = s2.intern();
```

String pool



s1



: String  
"Hi"

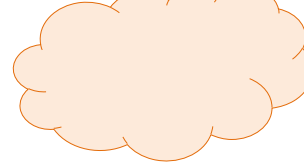
31

31

# String internalization

```
char chars[] = {'H', 'i'};  
String s1 = new String(chars);  
String s2 = new String(chars);  
String i1 = s1.intern();  
String i2 = s2.intern();
```

String pool



s1



: String  
"Hi"

s2



: String  
"Hi"

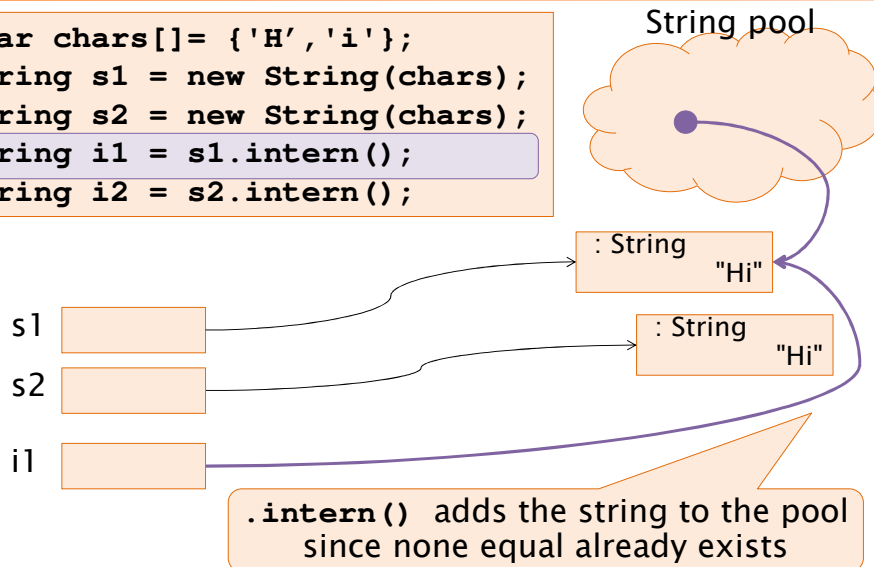
32

32



# String internalization

```
char chars[] = {'H', 'i'};  
String s1 = new String(chars);  
String s2 = new String(chars);  
String i1 = s1.intern();  
String i2 = s2.intern();
```

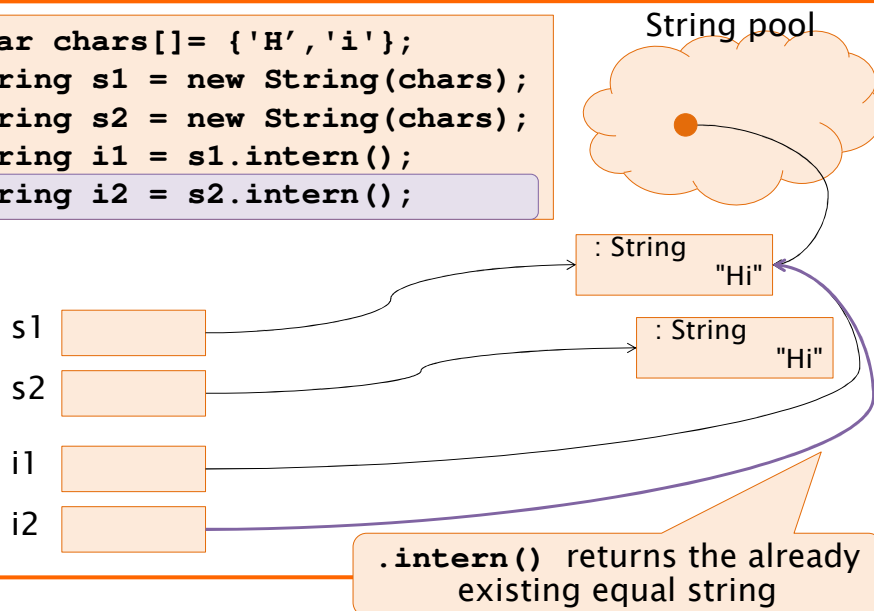


33

33

# String internalization

```
char chars[] = {'H', 'i'};  
String s1 = new String(chars);  
String s2 = new String(chars);  
String i1 = s1.intern();  
String i2 = s2.intern();
```



34

# Internalizing literals

---

```
String ssl = "Hi";
```

- ◆ Generates the same code as:

```
String ssl = (new String(  
    new char[]{'H', 'i'})  
    ).intern();
```

- ◆ On the first occurrence of a literal
    - compiler creates the string and
    - adds it to the pool
  - ◆ Upon later occurrences of a literal
    - compiler creates a string and
    - through intern returns reference to the one in the pool
- 

35

# Wrap-up

---

- Java characters are stored a 16 bits unicode
  - Conversion to/from streams of bytes is managed by **Charset** objects
  - **String** is immutable representation of strings
  - **StringBuffer** are mutable
    - ◆ Significantly more efficient for string manipulation
- 

36

36

## References

---

- Unicode specification
    - ♦ <http://www.unicode.org/versions/latest/>
  - Standard ECMA-94 “8-Bit Single Byte Coded Graphic Character Sets – Latin Alphabets No. 1 to No. 4”
    - ♦ <https://www.ecma-international.org/publications/standards/Ecma-094.htm>
-