JAVA PROGRAMMING BASICS

Module 2: Java Object-oriented Programming

Training program

1. Classes and Instances
2. The Methods
3. The Constructors
4. Static Elements
5. Initialization sections
6. Package
7. Inheritance and Polymorphism
8. Abstract classes and Interfaces
9. **String processing**
10. Wrapper classes for primitive types
11. Exceptions and Assertions
12. Nested classes
13. Enums
14. Generics
15. Collections
16. Method overload resolution
17. Multithreads
18. Core Java classes
19. Object Oriented Design
20. Functional Programming

Module contents

- String processing
  - The String class
  - Operations with Strings
  - Immutable String in Java
  - The StringBuilder and StringBuffer
  - String pool in Java
  - The StringTokenizer class
  - The Regular Expressions in Java
  - The Scanner class

Module contents

- String processing
  - The String class
  - Operations with Strings
  - Immutable String in Java
  - The StringBuilder and StringBuffer
  - String pool in Java
  - The StringTokenizer class
  - The Regular Expressions in Java
  - The Scanner class

The String class 1/4

- Strings, which are widely used in Java programming, are a sequence of characters.
- In the Java programming language, strings are objects.
- The Java platform provides the String class to create and manipulate strings.

- String greeting = **"Hello!"**;

String s5 = "\u041f\u0440\u0438\u0432\u0435\u0442\u01fe"; //ПриветǾ

Text Blocks

- Text block starts and ends with a “”” (three double-quote marks) followed by optional whitespaces and a newline.
- Inside the text block, we can freely use newlines and quotes **without the need for escaping line breaks**.

```
String page = """
        <html>
        <head>
        <meta charset="UTF-8">
        <title>Simple Servlet</title>
        </head>
        <body>
        <h2>Simple Servlet at %s</h2>
        <br/>%s
        </body>
        </html>
        """
```

New line required

Quote escape does not required

Since Java 15

The String class 2/4

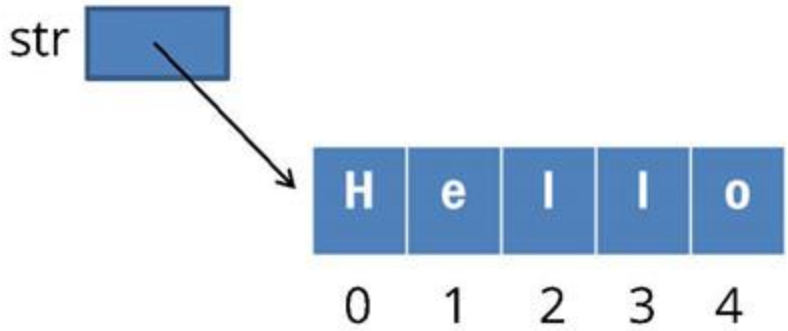
- String constructors:

1. String()
2. String(String value)
3. String(char[] value)

- String s1 = **new** String();
- String s2 = **new** String(**"Hello"**);

The String class 3/4

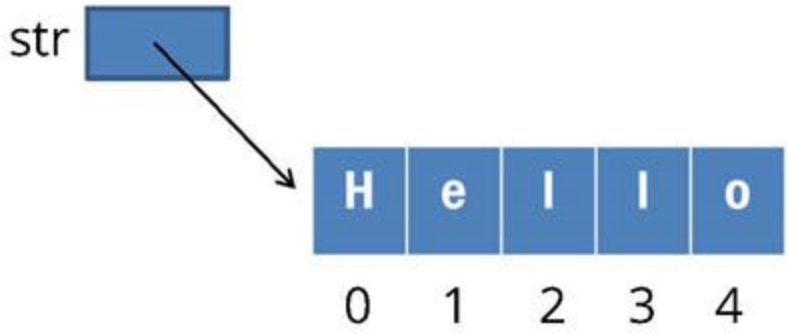
```
1. public static void main(String[] arg) {
2.     char[] helloArray = {'H', 'e', 'l', 'l', '\u0048'};
3.     String str = new String(helloArray);
4.     System.out.println(str);
5. }
```



String(**char[]** value, **int** offset, **int** count);

The String class 3/4

```
1. public static void main(String[] arg) {
                        throws UnsupportedEncodingException {
2.     byte [] helloArray = {0x48, 0x65, 0x6C, 0x6C, 0x6F};
3.     String str = new String(helloArray, "UTF-8" );
4.     System.out.println(str);
5. }
```



The String class 4/4

1. The length() method returns the number of characters contained in the string object

```
2. public static void main(String[] arg) {
3.     String str = "Hello World!!!";
4.     int len = str.length();
5.     System.out.println("String Length is : " + len);
6. }
```

Console output:

String Length is : 14

The String class

Before Java 9

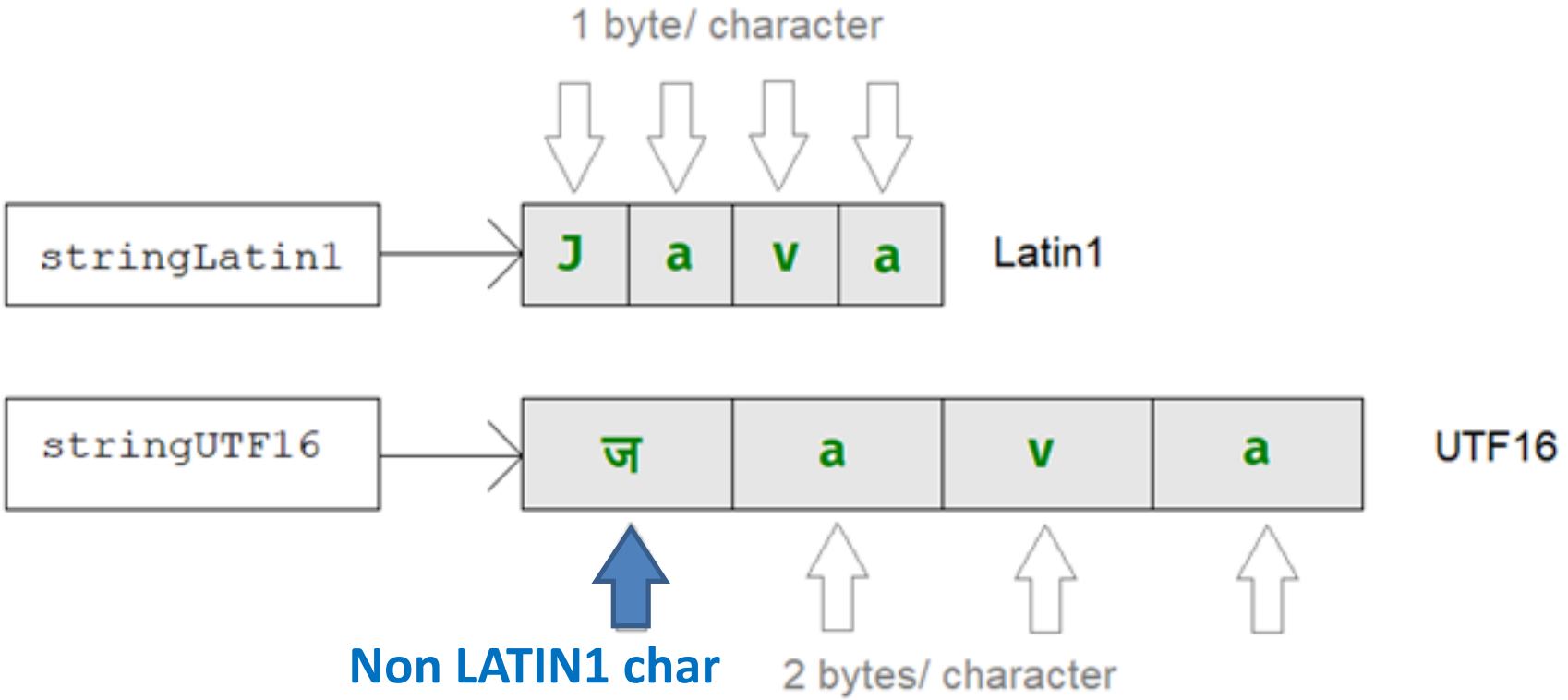
```
public final class String implements java.io.Serializable,
                                     Comparable<String>, CharSequence {

private final char value[];        //2 bytes per character
...
```

Since Java 9 - **Compact Strings**

```
public final class String implements java.io.Serializable,
Comparable<String>, CharSequence, Constable, ConstantDesc {
@Stable
private final byte[] value;        //1-2 bytes per character
/*The identifier of the encoding used to encode the bytes in value.
The supported values in this implementation are LATIN1 and UTF16*/
private final byte coder;          //LATIN1 = 0, UTF16 = 1
...
```

String are a major component of heap usage (approx. 20%) and *most* String objects contain only ISO-8859-1 or Latin-1 characters.

The String class



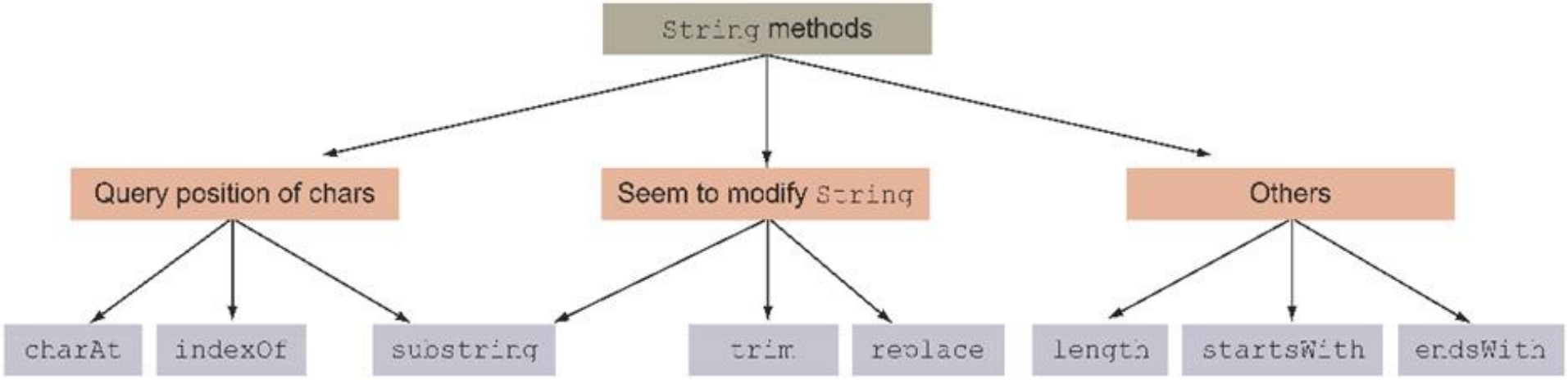
With Compact Strings, only the implementation details have changed for class String starting Java 9. **There are no API changes for class String.**

Module contents

- String processing
  - The String class
  - Operations with Strings
  - Immutable String in Java
  - The StringBuilder and StringBuffer
  - String pool in Java
  - The StringTokenizer class
  - The Regular Expressions in Java
  - The Scanner class

Operations with Strings 1/6

- The String class has a number of methods for examining the contents of strings, finding characters or substrings within a string, changing case, and other tasks.



String class basic methods
- Query position of chars

```
char charAt(int index)
int indexOf(String str)
int indexOf(String str, int fromIndex)
int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
int length()
```

String class basic methods
- Seem to modify String

String concat(String str)
String replace(**char** oldChar, **char** newChar)
String replace(CharSequence target,
CharSequence replacement)
String replaceAll(String regex, String replacement)
String substring(**int** beginIndex)
String substring(**int** beginIndex, **int** endIndex)
String repeat(**int** count)
String[] split(String regex)
String intern()
...

String class basic methods
- Seem to modify String

...

String toLowerCase()

String toString()

String toUpperCase()

String trim()

String class basic methods - Other

boolean isBlank()
boolean isEmpty()
boolean matches(String regex)
boolean endsWith(String suffix)
boolean equals(Object anObject)
boolean equalsIgnoreCase(String anotherString)
boolean startsWith(String prefix)
boolean startsWith(String prefix, **int** toffset)
...

String class basic methods - Other

...

```
int compareTo(String anotherString)
int compareToIgnoreCase(String str)
char[] toCharArray()
void getChars(int srcBegin, int srcEnd,
                    char[] dst, int dstBegin)
byte[] getBytes(String charsetName)
static String valueOf(primitive data type x)
```

Operations with Strings 2/6

- The String class includes a method for concatenating two strings or you can use “+” operator and += operator

```
1. String str1 = "Hello ";
2. String str2 = "World!!!";
3. String str3 = str1.concat(str2);
4. String str4 = str1 + str2;
5. System.out.println(str3);
6. System.out.println(str4);
   str4 +="!!!";
   System.out.println(str4);
```

The method returns a NEW string

Console output

```
Hello World!!!
Hello World!!!
```

Hello World!!!!!!

Operations with Strings 3/6

- Getting Characters by Index

```
1. String str = "Hello World!";
2. char aChar0 = str.charAt(0);
3. char aChar1 = str.charAt(1);
4. char aChar11 = str.charAt(11);
5. System.out.println(aChar0);
6. System.out.println(aChar1);
7. System.out.println(aChar11);
```

Console output:

```
H
e
!
```

Operations with Strings 4/6

- Getting Substrings

```
1. String str = "Hello World!";
2. String substr1 = str.substring(6);
3. String substr2 = str.substring(6,11);
4. String substr3 = str.substring(7,9);
5. System.out.println(substr1);
6. System.out.println(substr2);
7. System.out.println(substr3);
```

Console output:

```
World!
World
or
```

The method returns a NEW string

Operations with Strings 5/6

- Searching for Characters in a String

```
1. String str = "Hello World!";
2. int i1 = str.indexOf('o');
3. int i2 = str.lastIndexOf('o');
4. int i3 = str.indexOf('x');
5. System.out.println(i1);
6. System.out.println(i2);
7. System.out.println(i3);
```

Console output:

```
4
7
-1
```

Operations with Strings 6/6

- Searching for Substrings in a String

```
1. String str = "Hello World!";
2. int i1 = str.indexOf("Hello");
3. int i2 = str.lastIndexOf("World");
4. int i3 = str.indexOf("world");
5. System.out.println(i1);
6. System.out.println(i2);
7. System.out.println(i3);
```

Console output:

```
0
6
-1
```

String methods chaining


```
String result = "Sunday  ".replace(' ', 'Z').trim().concat("M n");

System.out.println(result);                    //SundayZZM n
```

The methods are evaluated from left to right.
The first method to execute in this example is replace, not concat.

Module contents

- String processing
  - The String class
  - Operations with Strings
  - Immutable String in Java
  - The StringBuilder and StringBuffer
  - String pool in Java
  - The StringTokenizer class
  - The Regular Expressions in Java
  - The Scanner class

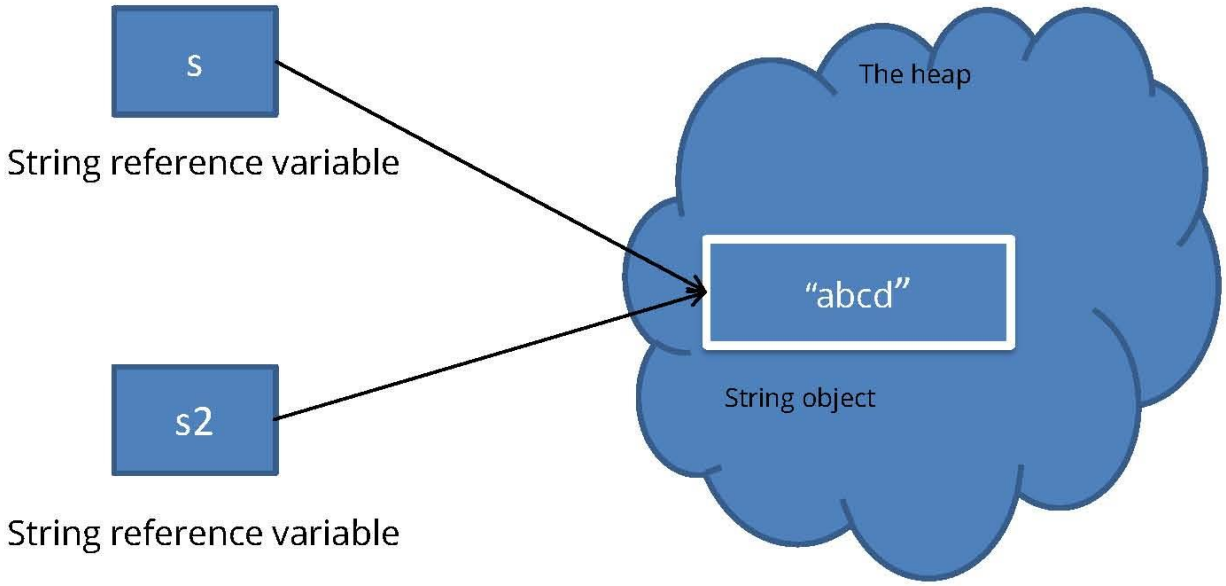
Immutable String in Java 1/3

- The String class is immutable, so that once it is created a String object cannot be changed.

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    /**
     * The value is used for character storage.
     *
     * @implNote This field is trusted by the VM, and is a subject to
     * constant folding if String instance is constant. Overwriting this
     * field after construction will cause problems.
     *
     * Additionally, it is marked with {@link Stable} to trust the contents
     * of the array. No other facility in JDK provides this functionality (yet).
     * {@link Stable} is safe here, because value is never null.
     */
    @Stable
    private final byte[] value;
```

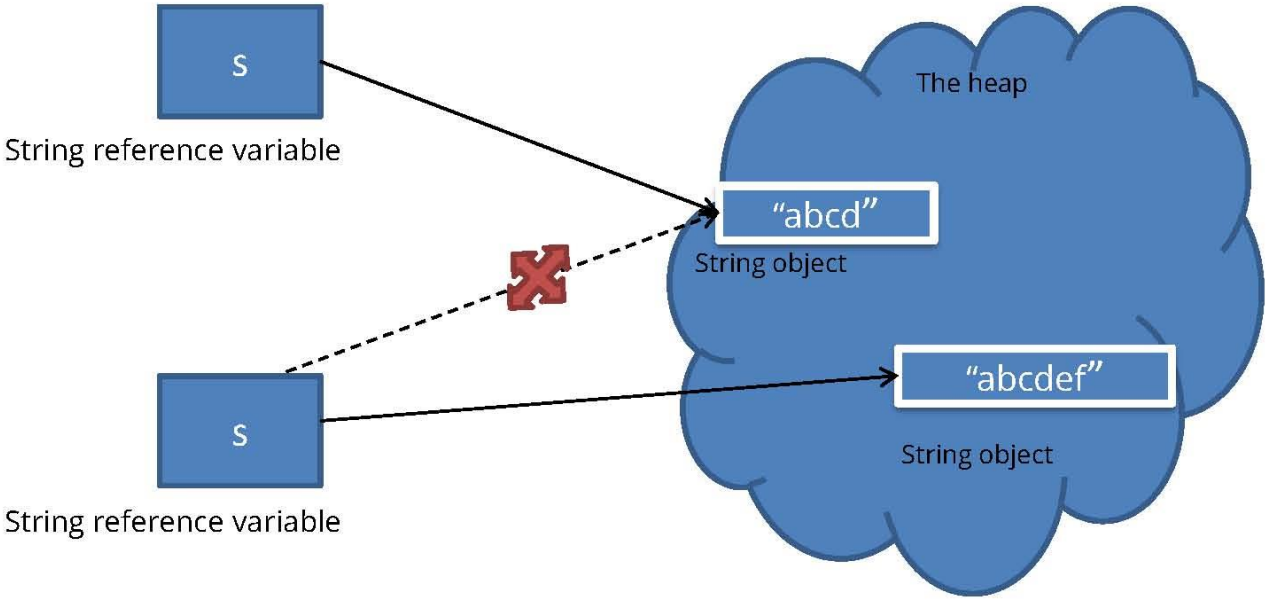
Immutable String in Java 2/3

1. String s = **"abcd"**;
2. String s2 = s;



Immutable String in Java 3/3

1. String s = **"abcd"**;
2. s = s+**"ef"**;



Module contents

- String processing
  - The String class
  - Operations with Strings
  - Immutable String in Java
  - The StringBuilder and StringBuffer
  - String pool in Java
  - The StringTokenizer class
  - The Regular Expressions in Java
  - The Scanner class

The StringBuilder and StringBuffer 1/6

- String str = **"A0"**;
  for (**int** i = 1; i < 7; i++){
  str+=**"P"**+i;
  }
  System.***out***.println(str);

- Each time creating new Object in heap
- Bad for performance

- **Objects in heap:**
- A0
- A0P1
- A0P1P2
- A0P1P2P3
- A0P1P2P3P4
- A0P1P2P3P4P5
- A0P1P2P3P4P5P6

You can print str.hashcode() in the loop to see that instances are different

StringBuilder/StringBuffer

```
public final class StringBuilder
    extends AbstractStringBuilder
    implements java.io.Serializable, Comparable<StringBuilder>, CharSequence
{
```

```
abstract class AbstractStringBuilder implements Appendable, CharSequence {
    /**
     * The value is used for character storage.
     */
    byte[] value;
```

no **final** modificator  mutable String

StringBuffer - since Java 1.0

StringBuilder - since Java 5

The StringBuilder and StringBuffer 2/6

- **StringBuffer** - thread-safe, mutable sequence of characters

- **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
- **StringBuffer(String str):** creates a string buffer with the specified string and capacity string.length() + 16.
- **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

The StringBuilder and StringBuffer 3/6

- **StringBuilder** – non-thread-safe, mutable sequence of characters
- The Java **StringBuilder** class is same as StringBuffer class except that it is non-synchronized.

StringBuilder(): creates an empty string Builder with the initial capacity of 16.
StringBuilder(String str): creates a string Builder with the specified string. and capacity string.length() + 16.
StringBuilder(int length): creates an empty string Builder with the specified capacity as length.

StringBuilder/StringBuffer basic methods

- Method modified StringBuilder/StringBuffer (synchronized for StringBuffer)

StringBuilder **append**(String s)
StringBuilder **insert**(**int** offset, String s)
StringBuilder **delete**(**int** startIndex, **int** endIndex)
StringBuilder **deleteCharAt**(**int** index)
StringBuilder **replace**(**int** startIndex, int endIndex,
String str):
StringBuilder **reverse**()

StringBuilder/StringBuffer basic methods

- Other methods

```
int capacity()
ensureCapacity(int minimumCapacity)
charAt(int index)
int indexOf(String str)
int indexOf(String str, int fromIndex)
int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
int length()
String substring(int beginIndex)
String substring(int beginIndex, int endIndex)
String toString()
int compareTo(StringBuilder another)
```

StringBuilder/StringBuffer basic methods

StringBuilder append(boolean b)
StringBuilder append(char c)
StringBuilder append(char[] str)
StringBuilder append(char[] str, int offset, int len)
StringBuilder append(double d)
StringBuilder append(float f)
StringBuilder append(int i)
StringBuilder append(long lng)
StringBuilder append(CharSequence s)
StringBuilder append(CharSequence s, int start, int end)
StringBuilder append(Object obj)
StringBuilder append(String str)
StringBuilder append(StringBuffer sb)

StringBuffer insert(**int** offset,
boolean b):
similarly!

The StringBuilder and StringBuffer 4/6

- **StringBuilder.append(String str)** method appends the specified string to this character sequence.

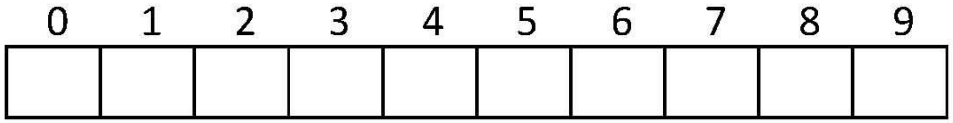
```
1. StringBuilder sb = new StringBuilder("A0");
2. for (int i = 1; i < 7; i++){
3.     sb.append("P");
4.     sb.append(i);
5. }
6. System.out.println(sb.toString());
```

The StringBuilder and StringBuffer 5/6

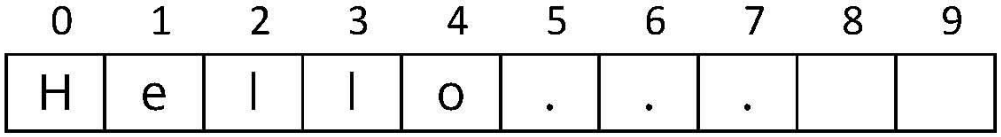
```
1. StringBuilder sb = new StringBuilder(10);
2. sb.append("Hello...");
3. char c = '!';
4. sb.append(c);// append a character
5. sb.insert(8, " Java");// Insert a string at index 5
6. sb.delete(5, 8);// Delete substring at index 5 to 8
7. System.out.println(sb);
```

The StringBuilder and StringBuffer 6/6

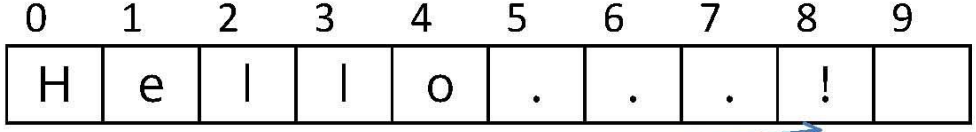
StringBuilder sb = new StringBuilder(10);



sb.append("Hello...");



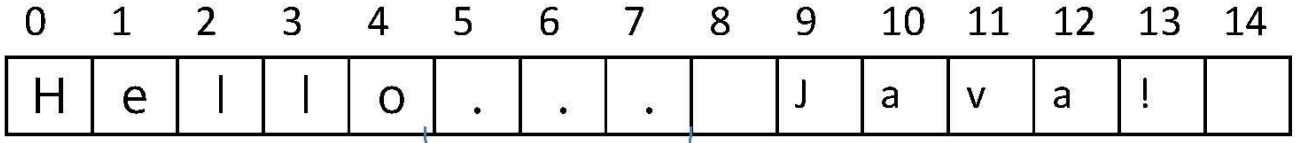
sb.append('!');



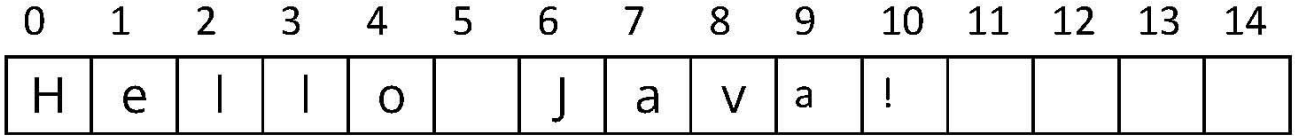
sb.insert(8," Java");

" Java"

whitespace



sb.delete(5,8);



Module contents

- String processing
  - The String class
  - Operations with Strings
  - Immutable String in Java
  - The StringBuilder and StringBuffer
  - String pool in Java
  - The StringTokenizer class
  - The Regular Expressions in Java
  - The Scanner class

String pool in Java 1/4

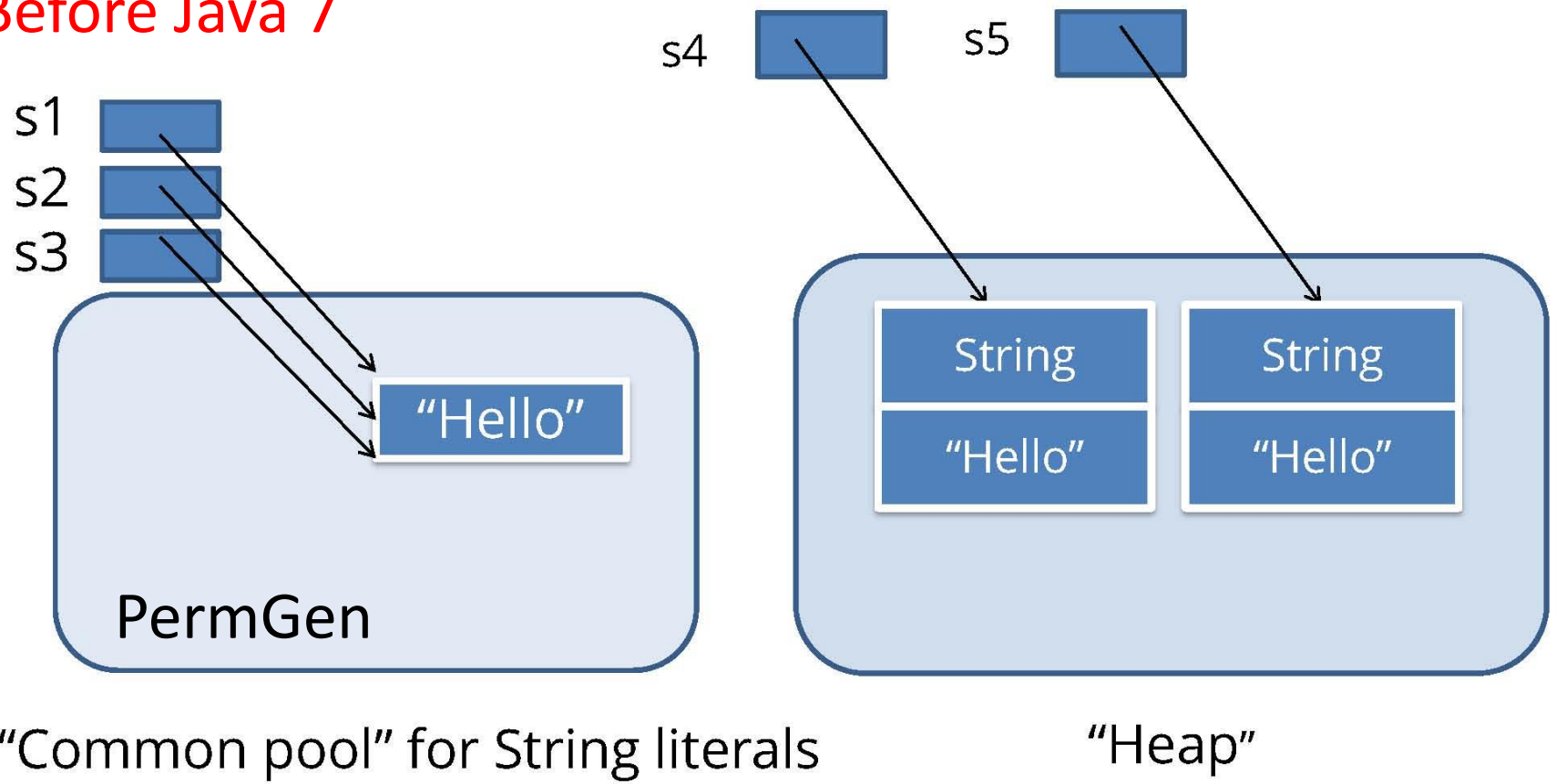
- Java has provided a special mechanism for keeping the String literals - in a so-called string common pool.
- If two string literals have the same contents, they will share the same storage inside the common pool.
- The JVM can make this optimization only because String is immutable.

String pool in Java 2/4

```
1. String s1 = "Hello";              // String literal
2. String s2 = "Hello";              // String literal
3. String s3 = s1;                   // same reference
4. String s4 = new String("Hello");  // new String object
5. String s5 = new String("Hello");  // new String object
```

String pool in Java 3/4

Before Java 7

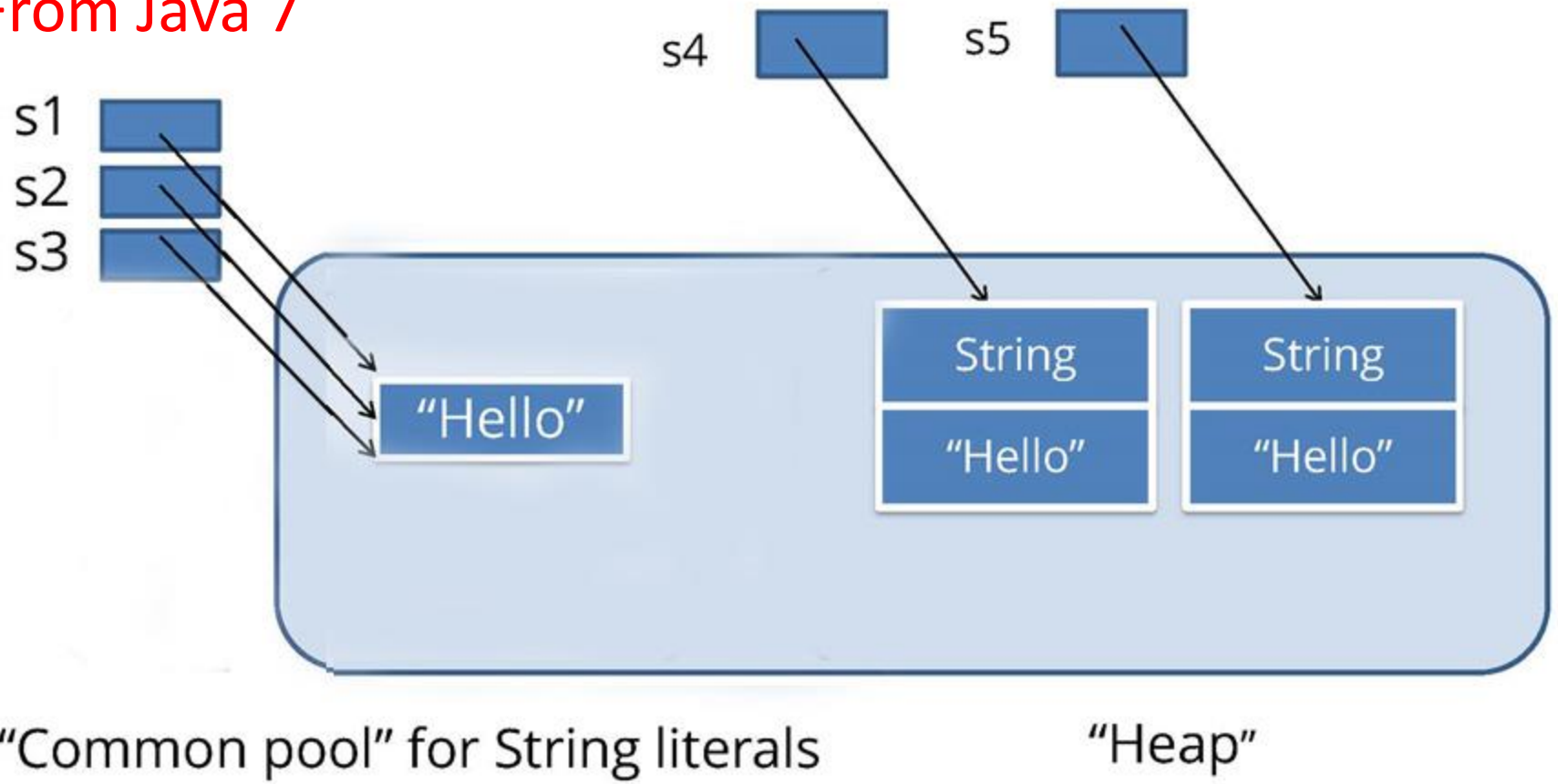


"Common pool" for String literals

"Heap"

String pool in Java 3/4

From Java 7



String pool in Java 4/4

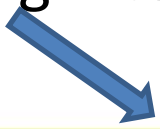
1. System.***out***.println(s1 == s1); *// true, same pointer*
2. System.***out***.println(s1 == s2); *// true, s1 and s1 share storage in common pool*
3. System.***out***.println(s1 == s3); *// true, s3 is assigned same pointer as s1*
4. System.***out***.println(s1.equals(s3)); *// true, same contents*
5. System.***out***.println(s1 == s4); *// false, different pointers*
6. System.***out***.println(s1.equals(s4)); *// true, same contents*
7. System.***out***.println(s4 == s5); *// false, different pointers in heap*
8. System.***out***.println(s4.equals(s5)); *// true, same contents*

Method intern()

Because the == operator checks for identity, all it has to do is compare two pointers, and obviously **this will be much faster than equals()**. So if you're going to compare the same strings multiple times, you can get a significant performance advantage by object identity checking instead of character comparison.

The basic algorithm:

1) Create a hash set of strings
2) Check that the string (as a sequence of characters) with which you are dealing is already in the set
3) If yes, then use a string from the set
4) Otherwise, add this string into a set and then use it



public native String intern();

Method intern()

- When the intern method is invoked, if the pool already contains a string **equal** to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object **is added** to the pool and a reference to this String object is returned.

```
String s1 = "abcd";
String s2 = "abcd";
String s3 = new String("abcd");
System.out.println(s1 == s2);                    //true
System.out.println(s1 == s3);                    //false
/*Add s3 to string pool*/
String s3Interned = s3.intern();
System.out.println(s1 == s3Interned);            //true
```

String s = new String("Some string"); adds "Some string" literal to pool

String concatenation

```
public static void main(String[] args) {
    String s1 = "Hello ";
    String s2 = "World!";

    /*Runtime concatenation - s3 is not treated as literal*/
    String s3 = s1 + s2;
    String s4 = "Hello World!";

    /*The strings computed by concatenation at runtime
      are newly created and distinct*/
    System.out.println(s3.equals(s4));  //true
    System.out.println(s3 == s4);       //false
...
```

String concatenation

```
...
   final String s5 = "Hello ";
   final String s6 = "World!";

    /*Compile time concatenation - s7 is treated as literal and added
        to string pool*/
   String s7 = s5 + s6;

   /*String values created from concatenation of constant expressions
     are computed at compile time and are treated as if they
     were literals*/
   System.out.println(s7.equals(s4));  //true
   System.out.println(s7 == s4);       //true
...
```

String concatenation

```
...
    String s8 = stringConcatenation(s5, s6);
    System.out.println(s8.equals(s4));          //true
    System.out.println(s8 == s4);               //false
}
/*The returned value wouldn't be known at compile time
    so it is not treated as literal*/
static String stringConcatenation(String firstVal, String secondVal) {
    return firstVal + secondVal;
}
```

Module contents

- String processing
  - The String class
  - Operations with Strings
  - Immutable String in Java
  - The StringBuilder and StringBuffer
  - String pool in Java
  - The StringTokenizer class
  - The Regular Expressions in Java
  - The Scanner class

The StringTokenizer class 1/5

- The java.util.**StringTokenizer** class allows an application to break a string into tokens.

tokens is a **parts of the initial string**

- This class is a legacy class that is retained for compatibility reasons although its use is discouraged in new code.

Instead use split() method of String:

String[] split(String regex);

The StringTokenizer class 2/5

StringTokenizer constructors:

- **StringTokenizer(String str)** creates StringTokenizer with specified string.
- **StringTokenizer(String str, String delim)creates** StringTokenizer with specified string and delimeter.
- **StringTokenizer(String str, String delim, boolean returnValue)** creates StringTokenizer with specified string, delimeter and returnValue.

- **delim** - is a string with one or set of characters that separate tokens. First constructor uses default delimeter - " \t\n\r\f" - the space, tab, newline, carriage-return and form-feed chars.
- If **returnValue** argument is **true**, delimiter characters are considered as separate tokens. If it is **false**, delimiter characters serve to separate tokens and don't print to output.

The StringTokenizer class 3/5

public class StringTokenizer implements Enumeration<Object>

Main methods:

public boolean hasMoreTokens()

public String nextToken()

public String nextToken(String delim)

public boolean hasMoreElements()

public Object nextElement()

public int countTokens()

Implementation of Enumeration abstract methods

The StringTokenizer class 4/5

- Using StringTokenizer

```
1. String str = "Its methods do not distinguish among identifiers, numbers, and quoted strings";
2. StringTokenizer s1 = new StringTokenizer(str);
3. while(s1.hasMoreElements()){
4.     System.out.println(s1.nextElement());
5. }
```

Console output:

```
Its
methods
do
not
distinguish
among
identifiers,
numbers,
and
quoted
strings
```

The StringTokenizer class 5/5

- Using StringTokenizer

```
1. String str = "Its methods do not distinguish among identifiers, numbers, and quoted strings";
2. StringTokenizer s2 = new StringTokenizer(str,",");
3. while(s2.hasMoreElements()){
4.     System.out.println(s2.nextElement());
5. }
```

Console output:

```
Its methods do not distinguish among identifiers
 numbers
 and quoted strings
```

Module contents

- String processing
  - The String class
  - Operations with Strings
  - Immutable String in Java
  - The StringBuilder and StringBuffer
  - String pool in Java
  - The StringTokenizer class
  - The Regular Expressions in Java
  - The Scanner class

The Regular Expressions in Java 1/7

- **Regular expression** (abbreviated regex or regexp) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations.
- The java.util.regex package primarily consists of three classes: **Pattern, Matcher, and PatternSyntaxException**.
- Regular Expression can be used to search, edit or manipulate text.

java.util.regex

The Regular Expressions in Java 2/7

- The Java **Pattern** class (java.util.regex.Pattern), is the main access point of the Java regular expression API.
- A **Pattern** object is a compiled representation of a regular expression.
- Pattern pattern = Pattern.*compile*(**".*http://.*"**);
- Pattern pattern = Pattern.*compile*(**".*http://.*"**,

  for multi-line searched text → Pattern.*DOTALL*);

"." - any character ".*" - any number of any characters

public static boolean matches(String regex, CharSequence input)

The Regular Expressions in Java 3/7

- The Java Matcher class (java.util.regex.Matcher) is used to search through a text for multiple occurrences of a regular expression.
- You can also use a Matcher to search for the same regular expression in different texts.

1. Matcher matcher = pattern.matcher("**searched in string http://mycompany.com**");
2. **boolean** matches = matcher.matches();

or

```
System.out.println(Pattern.matches(".*http://.*",
        "searched in string http://mycompany.com"));
                                                //true
```

Matcher methods

```
boolean matches();
boolean find();
boolean find(int start);
int start();
int start(int group)
int end();
int end(int group)
String group();
String group(int group)
```

true - if **entire** input sequence matches with pattern

true - if **part or entire** input sequence matches with pattern

For a matcher **m** with input sequence **s**, the expressions **m.group()** and **s.substring(m.start(), m. end())** are equivalent.

group

```
String patternString = "(This is the text)";
```

Matcher methods

```
String text = "This is the text to be searched. ";
String patternString = "(This is the text)";
Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(text);
int count = 0;
while (matcher.find()) {
        count++;
        System.out.println("\nfound: " + count + " : "
                + matcher.start() + " - " + matcher.end());
          System.out.println("found: " + matcher.group());
}
```

Output: found: 1 : 0 - 16

found: This is the text

The Regular Expressions in Java 4/7

- **Character Classes**
- . Dot, any character
- \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]
- \b - word boundary, e.g. "\\b(\\w{2})\\b"

In Java, you will need to “double escape” these backslashes.

String pattern = "\\b\\d \\D \\W \\w \\S \\s\\b";

The Regular Expressions in Java 5/7

(greedy - matches as much as possible from the input string)

- **Quantifiers**
- * Match 0 or more times
- + Match 1 or more times
- ? Match 1 or 0 times
- {n} Match exactly n times
- {n,} Match at least n times
- {n,m} Match at least n but not more than m times

add ? at the end of quantifier
make it reluctant (non-greedy)

See GreedyReluctantQuantifiers

The Regular Expressions in Java 6/7

- **Meta-characters**
- \ Escape the next meta-character (it becomes a normal/literal character)
- ^ Match the beginning of the line
- $ Match the end of the line (or before newline at the end)
- | Alternation (‘or’ statement)
- () Grouping
- [] Custom character class, [^] is inversion

The RegExp for Ukrainian alphabet is: "[А-Яа-яёЁЇїІіЄєҐґ']"

The dot character in RegExp have to be escaped: "\\."

The Regular Expressions in Java - Capturing group

Capturing group allow us to query the Matcher to find out what the part of the string was that matched against a particular parts of the regular expression pattern. This parts are embraces in ( ) and autonumbered. Index 0 represents the whole Pattern.

```
Pattern datePattern = Pattern.compile("([0-9]{4})-([0-9]{2})-([0-9]{2})");
String now = "Today is " + LocalDate.now().toString();
System.out.println(now);   //Today is 2022-11-25
Matcher m = datePattern.matcher(now);
if(m.find()){
    int year = Integer.parseInt(m.group(1));
    int month = Integer.parseInt(m.group(2));
    int day = Integer.parseInt(m.group(3));
    System.out.println(day + "/" + month + "/" + year);       //25/11/2022
}
```

The Regular Expressions in Java - Capturing group - greedy vs reluctant

```
String regex = "(.*)(\\d+)";  // greedy quantifier
Pattern pattern = Pattern.compile(regex);
String text = "The order number is 8983";
Matcher matcher = pattern.matcher(text);
while (matcher.find()) {
System.out.println(matcher.group(1));      //The order number is 898
System.out.println(matcher.group(2));      //3

regex = "(.*?)(\\d+)";            // reluctant (non-greedy) quantifier
pattern = Pattern.compile(regex);
matcher = pattern.matcher(text);
while (matcher.find()) {
   System.out.println(matcher.group(1));      //The order number is
   System.out.println(matcher.group(2));      //8983
}
```

The Regular Expressions in Java - String methods

- public boolean matches(String regex)
- public String replaceFirst(String regex, String replacement)
- String replaceAll(String regex, String replacement)
- public String[] split(String regex, int limit)
- public String[] split(String regex)

The Regular Expressions in Java - String methods

```
String sentence = "This is the sentence.";
String[] words = sentence.split(" ");
for (String word : words) {
    System.out.println(word);
}
```

Output:
This
is
the
sentence.

```
String sentenceWithExtraSpaces = "This  is    the     sentence.";
String spaceDeleted = sentenceWithExtraSpaces
                                    .replaceAll(" +", " ");
System.out.println(spaceDeleted);
```

Output:
This is the sentence.

one and more space

one space

The Regular Expressions in Java 7/7
Simple email validation

```
String email = "test@ukr.net";
String simplePattern = "^(.+)@(\\S+)$";
if (email.matches(simplePattern)) {
    System.out.println("This is email address");
} else {
    System.out.println("This is not email address");
}
```

The Regular Expressions in Java 7/7
Strict email validation

```
String emailPattern = "^(?=.{1,64}@)[A-Za-z0-9_-]+(\\.[A-Za-z0-9_-]+)*
@[^-][A-Za-z0-9-]+(\\.[A-Za-z0-9-]+)*(\\.[A-Za-z]{2,})$";
```

local-part

- It allows numeric values from 0 to 9.
- Both uppercase and lowercase letters from a to z are allowed.
- Allowed are underscore “_”, hyphen “-“, and dot “.”
- Dot isn't allowed at the start and end of the local part.
- Consecutive dots aren't allowed.
- For the local part, a maximum of 64 characters are allowed.

domain-part

- It allows numeric values from 0 to 9.
- We allow both uppercase and lowercase letters from a to z.
- Hyphen “-” and dot “.” aren't allowed at the start and end of the domain part.
- No consecutive dots.

Module contents

- String processing
  - The String class
  - Operations with Strings
  - Immutable String in Java
  - The StringBuilder and StringBuffer
  - String pool in Java
  - The StringTokenizer class
  - The Regular Expressions in Java
  - The Scanner class

The Scanner class

- The java.util.Scanner class is a simple text scanner, which can parse strings and primitive types using regular expressions.
- Scanner can get text data from the String instance, console input, text file, et al.
- Scanner breaks its input into tokens using a delimiter pattern, which is whitespace by default.
- A scanning operation may block waiting for input.
- A Scanner is not safe while multithreaded use without external synchronization.

The Scanner instance creation

```
String s1 = "Java is a general-purpose programming"
            + " language";
Scanner scanner = new Scanner(s1);
System.out.println(scanner.nextLine());

scanner = new Scanner(System.in);
System.out.println("Enter a line:");
String s2 = scanner.nextLine();
System.out.println(s2);

scanner = new Scanner(new File("src/mypackage/textfile.txt"));
while (scanner.hasNextLine()) {
    System.out.println(scanner.nextLine());
}

scanner.close();
```

The Scanner basic methods

```
public Scanner useDelimiter(String pattern)
public boolean hasNext()
public boolean hasNext(String pattern)
public String next()
public boolean hasNextLine()
public boolean hasNextLine(String pattern)
public String nextLine()
public boolean hasNextType()
public type nextType()
public void close()
```

type = boolean, byte, short, int, long, float, double, BigInteger, BigDecimal

The Scanner basic methods

```
String s = "Java is a general-purpose programming language "
            + "with current JDK 19";
Scanner scanner = new Scanner(s);
while(scanner.hasNext()){
    System.out.println(scanner.next());
}
```

Output:
Java
is
a
general-purpose
...

```
scanner = new Scanner(s).useDelimiter("-");
while(scanner.hasNext()){
    System.out.println(scanner.next());
}
```

Output:
Java is a general
purpose programming ...

```
scanner = new Scanner(s);
while (scanner.hasNext()) {
    if (scanner.hasNextInt()) {
        System.out.println(scanner.nextInt());
    } else {
        scanner.next();      }  }
```

...

Output:
19

The Scanner basic methods

```
...
scanner = new Scanner(s1);
/*Initialize the String pattern which signifies
  that the String token contains '-' character*/
String pattern = "(\\w*)-(\\w*)";
while (scanner.hasNext()) {
    // check if the token consists of declared pattern
    if (scanner.hasNext(pattern)) {
        System.out.println(scanner.next());
    } else
        scanner.next();
}

scanner.close();
```

Output:
general-purpose

The Scanner class 4/4

- The scanner can also use delimiters other than whitespace

```
1. String input = "1 fish 2 fish red fish blue fish";
2. Scanner s = new
   Scanner(input).useDelimiter("\\s*fish\\s*");
3. System.out.println(s.nextInt());
4. System.out.println(s.nextInt());
5. System.out.println(s.next());
6. System.out.println(s.next());
7. s.close();
```

Console output

```
1
2
red
blue
```