Text and strings

Char
String
StringBuilder
String functions

Char

Char

Char

type	Size (bits)	Description
char	16	Represents a Unicode character

Char represents a single Unicode character.

A char is defined as

```
char c = 'A';
```

Always used with single quotes ' '

Data Types – char and string

Escape characters are used to express special or non-printable characters.

```
// new line
char c = '\n';

// ☺
char d = '\u263a';

// "
char e = '\"';
```

Escape sequence	Represents
\b	Backspace
\t	Horizontal tab
\n	Line feed (LF)
\f	Form feed
\r	Carriage return (CR)
\"	Double quote
\'	Single quote
\\	Backslash
\uhhhh	Unicode character in hex notation.

Read more about the ☺ character (U+263A) here:
http://www.fileformat.info/info/unicode/char/263a/index.htm

Unicode

Data Types: char and String

Unicode allows us to represent most of the worlds characters in Java, including:

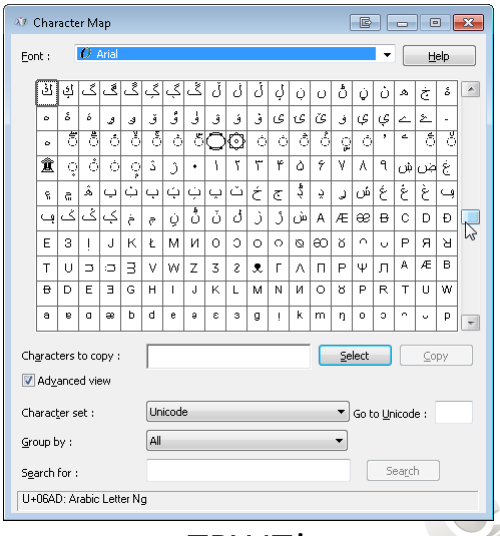
你好吗？ 很好, 谢谢。

안녕하세요

הרחפת שלי מלאה בצלופחים

Data Types: char and String

Using the Windows Character map tool you can explore the various character sets.



TRY IT!

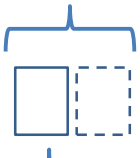
Data Types: String

A Unicode string **Hello**, as it is written to the screen here, is internally broken down as follow:

```
Hello
```

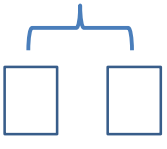
The character that we see on screen

One character on screen usually consists of one Java **char** value, but might need two.



char

Each Java char consumes **two bytes** of memory



byte

String

String

Type	Size (bits)	Description
String	Variable	Represents a immutable sequence of unicode chars.

A string is defined in Java as

```
String name = "Edument";
```

Always used with double quotes " "

Data Types – String

All escape sequences that are valid for **char** also works with strings.

```
String name = "Edument\r\nAB";
String path = "c:\\temp\\file.txt";
```

Outputs:

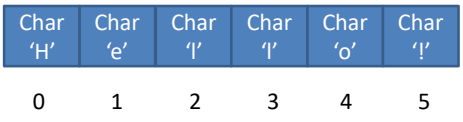
```
Edument
AB
```

```
C:\temp\file.txt
```

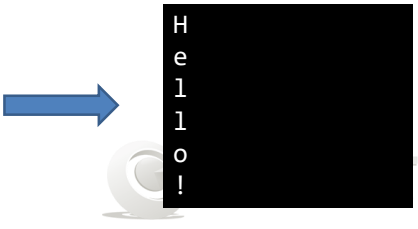
Working with strings

Working with strings

We can use the **charAt method** to access individual characters inside a string.




```
String str1 = "Hello!";
System.out.println(str1.charAt(0));
System.out.println(str1.charAt(1));
System.out.println(str1.charAt(2));
System.out.println(str1.charAt(3));
System.out.println(str1.charAt(4));
System.out.println(str1.charAt(5));
```



String

A string is actually an array of characters

Char array to string

```
char[] name = new char[7];
name[0]='E';
name[1]='d';
name[2]='u';
name[3]='m';
name[4]='e';
name[5]='n';
name[6]='t';

System.out.println(name);
String namestr = name.toString();
```

```
Edument
```

String to char array

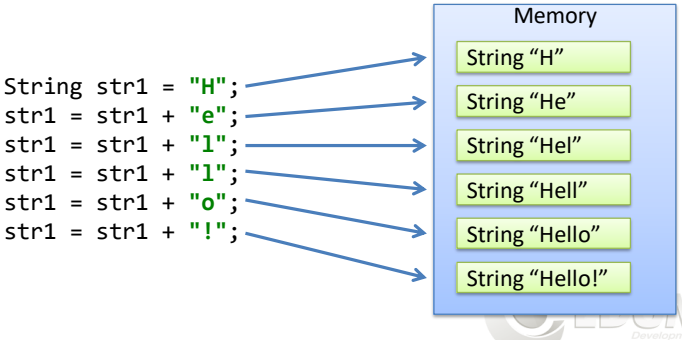
```
String name = "Edument";

System.out.print(name.charAt(0));
System.out.print(name.charAt(1));
System.out.print(name.charAt(2));
System.out.print(name.charAt(3));
System.out.print(name.charAt(4));
System.out.print(name.charAt(5));
System.out.print(name.charAt(6));

char[] namearray = name.toCharArray();
```

```
Edument
```

Working with strings

However strings are **immutable** (read-only)

This means that each time we modify a string, a new string is created.



Working with strings

Each time we want to change a string, a **new string** will automatically be created.

For example, this code will create 100000 strings in memory.

```
String str1 = "";
for (int i = 0; i < 100000; i++) {
    str1 = str1 + "!";
}
```

When a new string is created, the content of the previous string is **copied** into the new string.

Working with strings

This idea that strings are **immutable** is actually a good thing.

For example if we have this string:

```
String str1 = "Hello World";
```

Then I can always be sure that no-one else in the system can change it!

Working with strings

An unassigned string has the default value **null**.

```
String str;
String str2 = null;
```

However null is **not the same** as an empty string.

We can assign an **empty string** like this:

```
String emptyStr = "";
```

Working with strings

We can test if a string is **null** using:

```
if (str1 == null) {
    System.out.println("The string is null.");
}
```

We can test if a string is **null** or **empty** using:

```
if (str1 == null || str1.equals("")) {
    System.out.println("The string is null or empty.");
}
```

Comparing strings

When comparing strings, we <u>should</u> use the **equals** method.

```
String str1 = "Hello";
String str2 = "Hello";

if(str1.equals(str2))
{
    //str1 contains the same value as str2
}
```

Comparing using **==** may seem to work, but will introduce various problems:

```
String str1 = new String("test");
String str2 = new String("test");

//Will print Not same
if(str1 == str2)
    System.out.println("Same");
else
    System.out.println("Not same");
```

```
String str1 = "Hello";
String str2 = "Hello";

//Will print same
if(str1 == str2)
    System.out.println("Same");
else
    System.out.println("Not same");
```

Same due to compiler trick (string interning)

StringBuilder

StringBuilder (1)

As we know, strings in Java are **immutable**.

This means that when modifying a string, you actually create a **new** string in memory.

If you do this many times (such as in a loop), this can quickly become **expensive**.

StringBuilder (2)

If we know in advance we're going to do lots of work with strings, we should consider using the **StringBuilder** class.

Unlike normal strings, StringBuilder is **mutable**, and you can change the contents as much as you want.

StringBuilder (3)

When creating a StringBuilder, you can pass an optional initial string to the constructor.

```
StringBuilder sb = new StringBuilder("Hello ");
sb.append("Tore.");
```

When you are done, you can convert the StringBuilder back to a string via the **toString** method:

```
String myString = sb.toString();
```

StringBuilder (4)

StringBuilder contains the methods you'd expect from string, such as **Remove**, **Replace** and **Insert**.

```
StringBuilder sb = new StringBuilder("Hello ");
sb.append("Tore..");

sb.insert(5, ",");          // Insert a comma
sb.replace(7, 11, "class"); // Replace "Tore" with "class"
sb.delete(12, 13);          // Remove the additional .

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

```
Hello, class.
```

StringBuilder (5)

As we discussed earlier, the following code will create 100000 strings in memory.

```
String str1 = "";

for (int i = 0; i < 100000; i++) {
    str1 = str1 + "!";
}
```

If we use a StringBuilder instead, we create far fewer objects in memory.

This will use much less memory, and be much quicker.

StringBuilder (6)

Let's compare the speed difference:

```
ZonedDateTime now = ZonedDateTime.now();

String str1 = "";
for (int i = 0; i < 100000; i++) {
    str1 = str1 + "!";
}

long seconds = now.until(ZonedDateTime.now(), ChronoUnit.MILLIS);
System.out.println(seconds + " ms");

ZonedDateTime now2 = ZonedDateTime.now();

StringBuilder builder = new StringBuilder();

for (int i = 0; i < 100000; i++) {
    builder.append("!");
}

seconds = now2.until(ZonedDateTime.now(), ChronoUnit.MILLIS);
System.out.println(seconds + " ms");
```

```
10529 ms
1 ms
```

StringBuilder (7)

As this shows, StringBuilder provides substantial improvements when you are performing many changes, such as in loops.

Using StringBuilder is not recommended when you are only working with a small number of changes (e.g. combining 2-3 strings).

This is because the performance improvement is negligible.

Exercise 8

Lets do exercise 8

Reference
- String functions

Reference – String functions

String functions reference material

Working with strings

To determine the **length** of a string , we can use:

```
String str1 = "Hello World!";

System.out.println(str1.length());
```

```
12
```

To convert a string to all upper or lower case:

```
String str1 = "Hello World!";

System.out.println(str1.toUpperCase());
System.out.println(str1.toLowerCase());
```

```
HELLO WORLD!
hello world!
```


Finding characters

We can use the following methods to test if a substring exists in a string.

Method	Description
contains	True if the substring exists anywhere in the string
endsWith	True if the string ends with the substring
startsWith	True if the string starts with the substring.

All these methods returns **true** or **false**.

```
String str1 = "Hello World!";

System.out.println(str1.contains("World")); // True
System.out.println(str1.startsWith("Hello")); // True
System.out.println(str1.endsWith("!")); // True
```

Finding characters

We can use the following methods to determine the position of a substring in a string

Method	Description
indexOf(char)	Returns the index of the first occurrence of the char
indexOf(string)	Returns the index of the first occurrence of the string
lastIndexOf(char)	Returns the index of the last occurrence of the char
lastIndexOf(string)	Returns the index of the last occurrence of the string

All these methods returns -1 if not found.

```
String str1 = "Hello World!";

System.out.println(str1.indexOf('o')); // 4
System.out.println(str1.indexOf("Hello")); // 0
System.out.println(str1.lastIndexOf('o')); // 7
System.out.println(str1.lastIndexOf("World")); // 6
```


Trimming characters

To remove whitespace characters from the start and end of a string, we can use the Trim method.

Method	Description
trim()	Removes all whitespace characters from the start and end

```
String str1 = "  Hello World!  ";

System.out.println("'" + str1.trim() + "'");
```

```
'Hello World!'
```

Replace

We can use the **Replace** to replace a string or char with another string or char.

Method	Description
replace(findStr, replaceStr)	Replaces the string findStr with replaceStr.
replace(findChar, replaceChar)	Replaces of the char findChar with replaceChar.

```
// Must use an empty string because there is no empty char in Java
String str1 = " A B C D E F ".replace(" ", "");
System.out.println(str1);

String str2 = "Hello Hello Hello Hello Hello!!".replace("Hello", "Hi");
System.out.println(str2);

String str3 = "1:2:3:4:5:6:7".replace(':', '-');
System.out.println(str3);
```

```
'ABCDEF'
'Hi Hi Hi Hi Hi!!'
'1-2-3-4-5-6-7'
```


Splitting strings

Split is a useful method to split a string into its parts that are separated by a given character.

```
String input = "0:7:5:3:1:34:78:43:100:10:0";

String[] numbers = input.split(":");

for (String num : numbers) {
    System.out.println(num);
}
```

```
0
7
5
3
1
34
78
43
100
10
0
```

Joining strings

Join is the opposite of split.

It will concatenate a collection of strings using the provided separator between each item.

```
String input = "0:7:5:3:1:34:78:43:100:10:0";

String[] numbers = input.split(":");

String result = String.join(", ", numbers);

System.out.println(result);
```

```
0, 7, 5, 3, 1, 34, 78, 43, 100, 10, 0
```


Formatting Strings

Formatting Strings

We often want to combine strings, particularly when outputting them. We can make this easier through Java's **formatting**.

This is done by passing at least two strings: one **format string**, and then **one or more** normal strings.

The **format string** is just like a normal string, but contains special **format items** inside **{}**. These items are replaced automatically when the string is created.

Formatting Strings

This can be used to combine string variables with a welcome message:

```
Scanner in = new Scanner(System.in);

String firstname = in.nextLine();
String surname = in.nextLine();

String greeting = String.format("Hello, %s %s", firstname, surname);

System.out.println(greeting);
```

```
Hello, Tore Nestenius.
```

Formatting Strings

We can also use additional data in our **format items** to format the string. We can pad strings with spaces or choose how numbers are output.

```
String str = "\"word\"";

System.out.println(String.format("This string: %15s is left padded to 15 chars", str));
System.out.println(String.format("This string: %-15s is right padded to 15 chars", str));

double cost = 24.99;
System.out.println(String.format("This item costs £%.2f.", cost));
```

```
This string:          "word" is left padded to 15 chars
This string: "word"          is right padded to 15 chars
This item costs £24.99.
```


Summary

The methods we mentioned here are the most common one, but do explore the alternatives.

There are many more string methods and overloads available.

Many methods also have options to deal with different culture (language) settings.

To see all methods visit:

http://docs.oracle.com/javase/8/docs/api/java/lang/String.html