# Strings and Regular Expressions

CSC02A2

# Outline

# Outline





# Strings

# The String Class

Strings in **Java** are fully fledged classes. The **String** class is found in the *java.Lang* package. The **String** class is imported by default so there is no need to import it manually.

Some helpful functions are available in the the **String** class, such as ***Length()*** which returns the length of the **String**. **Java Strings** are **zero-indexed**.

Some examples of how **String** instances can be created:

```
1 String newString = new String("String literal");
2 String boxedString = "String literal as well";
3 char[] sequence = {'H','E','L','L','O'};
4 String charString = new String(sequence);
```



# String Immutability

**String** is an immutable class (immutable classes discussed in previous lecture).

The following code does not alter the original **String**:

```
1 String myString = "Hello";
2 myString = "Goodbye"; // The String Hello is still in memory
```

On line 2 of the above code, the **String** is still in memory as only the reference has changes when we assigned the new value to the variable ***myString***. This is exploited by interned **String**s.



# Interned Strings

**Java** attempts to save memory and improve efficiency by using interned strings. Each **unique** character sequence exists in memory only once. Each reference to this unique sequence of characters points to the same location in memory. To override this mechanism a reference to a new **String** will point to a new location in memory.

```
1 String stringRef1 = "Hello World";
2 String stringRef2 = new ("Hello World");
3 String stringRef3 = "Hello World";
4 String stringRef4 = "hello World";
5
6 System.out.println(stringRef1 == stringRef2); // false
7 System.out.println(stringRef1 == stringRef3); // true
8 System.out.println(stringRef1 == stringRef4); // false
```



# String Comparison

**Java**'s == operator only performs reference checks and cannot be overridden.

```
1 String ref1 = new String("abc");
2 String ref2 = new String("abc");
3
4 if(ref1 == ref2)  { System.out.println("Same"); }
5 else              { System.out.println("Not same"); }
```

Each reference is a separate String in memory and therefor will not be the same. However, there is a method available to test character-wise equality: ***equals()*** (this method is from the *java.lang.Object* class and is overridden).

Another method which can be used to compare **String** instances in the ***compareTo()*** method. This method employs lexicographic ordering.

An example of lexicographic ordering:

```
1 "" "!" "0" "A" "Andy" "Z" "Zero" "a" "an" "and" "andy" "candy" "zero"
```



# String methods

The **String** class contains a number of helpul functions pertaining to operating with strings.

- ***chatAt()*** provides a character and a specified index.
- ***substring*** returns a sub-string which is specified from the parameters.
- ***indexOf*** and ***lastIndexOf*** searches of a sequence of characters.
- ***toUpperCase()*** convert **String** to upper-case.
- ***toLowerCase()*** convert **String** to lower-case.
- ***trim()*** removes trailing whitespace.
- ***replace()*** replace character(s) with another set of character(s).

See the **String** **Java**Doc for more methods which are available.



# StringBuilder and StringBuffer

**StringBuilder** and **StringBuffer** provide a means to work with mutable *strings*. Modifying a **String** creates new instances and results in the garbage collector having to clean up old references. Using either a **StringBuilder** or **StringBuffer** avoids this problem.

- **StringBuilder** is fast in operation but is not thread safe.
- **StringBuffer** is slower but is thread safe.

Threads and multi-threaded programming will be discussed in a future lecture.



# StringTokenizer

The **StringTokenizer** class breaks up an input **String** into *tokens*. Tokens are useful pieces of **String** which may have special meaning to the problem at hand.

For example: Given an array **String**s where each element is single combined **String** of username and password in the format "username:password" provide a function to separate the username and password.

```
1 public static String[] separate(String combined)
2 {
3   StringTokenizer userpassTokens = new StringTokenizer(combined, ":");
4   String username = userpassTokens.nextToken();
5   String password = userpassTokens.nextToken();
6   String[] pieces = {username, password};
7   return pieces;
8 }
```



# Regular Expressions

# Regular Expressions

A regular expression is a sequence of characters that is used to find a specific pattern. Regular expressions are defined by a formal language.

Each regular expression consists of special character classes. Each character class has a unique meaning.

Along with character classes, quantifiers can be specified to indicate the amount of a character class that is required.

**Note**

The following slides show the **Java** Regular Expression convention which might not work in other regular expression implementations.



# Character classes

| Class | Meaning | Description |
|---|---|---|
| `[abc]` | a, b or c | simple class |
| `[^abc]` | Anything except a, b or c | negation |
| `[a-zA-Z]` | a through z, or A through Z | range |
| `[a-d[m-p]]` | a through d, or m through p | union |
| `[a-z&&[def]]` | d, e or f | intersection |
| `[a-z&&[^bc]]` | a through z except b and c | subtraction |

| Special | Meaning | Description |
|---|---|---|
| `.` | ... | any single character |
| `\d` | `[0-9]` | a single digit |
| `\D` | `[^0-9]` | a single non-digit |
| `\s` | `[\t\n\x0B\f\r]` | a single whitespace character |
| `\S` | `[^\s]` | a single non-whitespace character |
| `\w` | `[a-zA-Z_0-9]` | a single word character |
| `\W` | `[^\w]` | a single non-word character |



# Quantifiers

| Greedy | Reluctant | Possessive | Meaning |
|---|---|---|---|
| X? | X?? | X?+ | X, once or not at all |
| X* | X*? | X*+ | X, zero or more times |
| X+ | X+? | X++ | X, one or more times |
| X{$n$} | X{$n$}? | X{$n$}+ | X, exactly $n$ times |
| X{$n$,} | X{$n$,}? | X{$n$,}+ | X, at least $n$ times |
| X{$n$,$m$} | X{$n$,$m$}? | X{$n$,$m$}+ | X, at least $n$ but no more than $m$ times |



# Java and Regular Expressions

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class RegexDate
{
  public static void main(String[] args)
  {
    // Dates we want to check if they are valid
    String[] dates =
      {  "2016-02-26" , "2099-12-31", "2099-13-31",
        "2099-12-32" , "2089-01-02", "BACD-02-02",
        "2016/02/26" , "2016-02-26", "2016-00-26",
        "2016-012-31", "2016-22-31", "2016-13-31",
        "2016-10-31" , "2016-12-32", "2016-O1-31" };

    // Regular expression for checking dates
    // \d{4}[/-](0[1-9]|1[012])[/-](0[1-9]|[12][0-9]|3[01])
    Pattern datePattern =
        Pattern.compile("\\d{4}[/-](0[1-9]|1[012])[/-](0[1-9]|[12][0-9]|3[01])");

    // For each date in dates array
    for (String date : dates)
    {
      // Create matcher for specific date and pattern
      Matcher dateMatcher = datePattern.matcher(date);
      // Test if date matches pattern
      if (dateMatcher.matches())
      {
        System.out.println(date + " matches!");
      } else {
        System.err.println(date+ " does not match!");
      }
    }
  }
}
```