Strings

** String Classes: **`String`**, **`StringBuffer`**, **`StringBuilder`**, **`StringTokenizer`**, **`Scanner`** and **`Character`**

** Formatting numerical data

Chapter 3 (sections 3.6+3.7) – "Big Java" book
Chapter 10 + Appendix B – "Head First Java" book
Chapter 9 – "Introduction to Java Programming" book
Chapter 9 – "Java in a Nutshell" book

Strings

- **Strings** are not a basic type in Java – they are objects!
  - But are so common that Java provides some language level support for strings.
  - As in C, string literals are delimited by double quote marks.
    - **Example**: `"Hello"` is an acceptable string literal.
- To use strings, we can (*but don't have to!*) input the **`java.lang`** package, by adding the statement **`import java.lang.String;`**.
  - The **`java.lang`** package is part of the JDK Class Library and provides classes that are fundamental to the Java programming language's design.
  - The **`import`** keyword tells the compiler that the program uses external packages.

The **`java.lang`** package is automatically imported into every Java program.

Strings: Another Property

- Strings are **immutable** (i.e. *can't be changed*): this is for security purposes and to minimise memory usage.
  - **Example**: Ten `String` objects are created (with values `"0"`, `"01"`, …, `"0123456789"`). At the end,
    - variable `s` refers to the `String` object with value `"0123456789"`;
    - there exist 10 `String`s.

```
String s = "0";
for (int x=1; x<10; x++) { s = s + x; }
```

- **String Pool**: Area of memory where `String` literals are put by the JVM when created.
  - JVM doesn't create a duplicate if there's already a `String` in memory with the same value; it refers the reference variable to the existing entry.
  - **Garbage Collector** doesn't cleanup the String Pool!

Strings are immutable (i.e. *can't be changed*) …

- How *general objects* behave (remember the **`Rabbit`** class …):

```
Rabbit r1 = new Rabbit();
r1.setName("Benji");
Rabbit r2 = r1;
r2.setName("Blinky");
System.out.println(r1.getName());
System.out.println(r2.getName());
```

Prints:

```
------
------
```

- How *Strings* behave:

```
String s1 = "Sherlock";
String s2 = s1;
s2 = "Holmes";
System.out.println(s1);
System.out.println(s2);
```

Prints:

```
--------
--------
```

Example: Mutable & Immutable Objects

```
public class Rectangle {
  private int width, height;

  public Rectangle(int h, int w) {
    height = h;
    width = w;
  }
  public int getWidth() {
    return width;
  }
  public int getHeight() {
    return height;
  }
  public void setWidth(int w) {
    width = w;
  }
  public void setHeight(int h){
    height = h;
  }
}
```

```
public class Colouring {
  private int red, green, blue;

  public Colouring(int r, int g, int b) {
    red = Math.max(0, Math.min(255, r));
    green = Math.max(0, Math.min(255, g));
    blue = Math.max(0, Math.min(255, b));
  }
  public int getRed() { return red; }
  public int getGreen() { return green; }
  public void getBlue() { return blue; }
  public Colouring tint(double t) {
    int r = Math.round(red*t);
    int g = Math.round(green*t);
    int b = Math.round(blue*t);
    return new Colouring(r,g,b);
   }
}
```

Examples of Java API classes whose objects are immutable: **String**, **Character**, **Integer**, …

String Classes: `String`

- Java overloads the **+** operator for string concatenations.
  - Unlike C++, this is pretty much the only operator overloading in Java!
- As strings in Java are objects, methods can be invoked on them.
- String classes:
  - for **constant strings** → **`String`**;
  - **indexing of string elements**: starts at **`0`** and ends at **`length()-1`**.
  - **`String`** class has several constructors:

```
String s = "HelloWorld!";                // s = "HelloWorld"
String s = new String("HelloWorld!");    // s = "HelloWorld"
String s = new String();                 // s is an empty string

char[] Cat_Array ={'t','i','g','e','r','s'};  // like an array
String s = new String(Cat_Array);             // s = "tigers"
String s = new String(Cat_Array,1,2);         // s = "ig"
```

Methods: `String` Class (1/3)

- **`int length()`**: returns number of characters in a **`String`** object.

```
String s = "HelloWorld!";
System.out.println(s.length());          // output is 11
int i = "The rain in Spain".length(); // i = 17
```

- **`char charAt(int index)`**: returns the character at **`index`**.

```
char c = "The rain in Spain".charAt(4); // c = 'r'
String s = "HelloWorld!";
System.out.println(s.charAt(5)); // output is W
```

- **`int indexOf(ch)`**: returns **`ch`**'s first occurrence position; if not found returns **`-1`**.

```
int i = "She sells sea-shells on the sea shore".indexOf('s');
// i = 4
```

Method	Return Value
indexOf(char ch)	position of first ch
indexOf(String str)	position of first str (string)
lastIndexOf(char ch)	position of last ch
lastIndexOf(String str)	position of last str (string)

Methods: `String` Class (2/3)

- **`boolean equals(obj)` / `boolean equalsIgnoreCase(str)`**:

```
if ("black".equals("white")) {
  System.out.println("A deer is a horse.");
}
```

- **`int compareTo(str)`**: compares two strings, returns < ,> , =0 if the compared string is smaller, larger, or equal to **`str`**.

```
String str1 = "Joanna";
String str2 = "James";
int result = str1.compareTo(str2);
if (result < 0)
  System.out.println("str1 comes before str2");
else if (result > 0)
  System.out.println("str1 comes after str2");
else
  System.out.println("str1 and str2 are equal");
```

Methods: `String` Class (3/3)

- **`substring(index1,index2)`**: returns the substring between **`index1`** and (excluding) **`index2`**.

```
String s = "HelloWorld!".substring(1,6);
// S = "elloW";
```

- **`concat(s)`**: concatenates two strings.

```
String s = "Hello".concat("World"); // s = "HelloWorld"
```

- **`toUpperCase()`** / **`toLowerCase()`**: convert all characters in string to upper/lower case.

```
String sUpper = "Cat".toUpperCase(); // sUpper = "CAT"
String sLower = "Cat".toLowerCase(); // sLower = "cat"
```

- **`toString()`**: convert input to a string.

```
double d = 12.3;
String dString = Double.toString(d); // dString = "12.3"
```

When you write a new class, you should override the **`toString()`** method.

`String` Class : The `split()` Method

- **`split(String s)`**: splits the string around matches of the given *regular expression* **`s`** and returns an array with those substrings.

```
public class UsingSplit {
  public static void main(String[] args) {
    String str = "bar:foo:bar";
    String[] splitStr = str.split(":");
    for (int i=0; i < splitStr.length; i++)
      System.out.println(splitStr[i]);
  }
}
```

Output is …

```
bar
foo
bar
```

Another example …

```
String[] splitStr = str.split("a");
```

then output is …

```
b
r:foo:b
r
```

Some other methods of the `String` class

- **`void getChars(i,j,A,k)`**: returns characters from **`i`** to **`j`** (excluding), and stores them into *array* **`A`** starting from **`A[k]`**.

```
char[] A = new char[4];
"The rain in Spain".getChars(4,8,A,0);
// A = {'r','a','i','n'}
```

- **`substring(index)`**: returns substring from **`index`** to end.

```
String s = "Monkeys".substring(3);          // s = "keys"
String s = "Monkeys".substring(3,"Monkeys".length()-1);
                                            // s = "key"
```

- **`replace(oldCh,newCh)`**: replace **`oldCh`** by **`newCh`** everywhere in the string.

```
String s = "goose".replace('o','e'); // s = "geese"
```

For other **`String`** class methods, see the Java API: https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/String.html.

StringBuffer & StringBuilder Classes

- Used for variable strings, whereas **String** class is used for constant strings.
  - Usually, you leave it to the system to use the **StringBuffer** class.
  - **Example**: If we write **"Bugs" + " Bunny"**, Java will automatically call a **StringBuffer** class method to generate the string **"Bugs Bunny"**.
- Java uses Unicode (2 bytes per character) → not very efficient. Class **StringBuffer** makes it worse (always has to re-allocate space). To alleviate this, Java always allocates 16 more characters than needed.

```
StringBuffer s = new StringBuffer();
StringBuffer s = new StringBuffer(20);
StringBuffer s = new StringBuffer("cat");  // s = "cat"
```

- Old Java uses **StringBuffer** – *from Java 5.0 onwards*, **StringBuilder** is used (as it's more efficient).

Use when you have lots of **String** manipulation in a program!

`StringBuilder` Class (*from Java 5.0*)

- Defines the **same methods** as **`StringBuffer`**, but doesn't declare them as synchronised.
  - Use **`StringBuilder`** class when a program:
    - uses only one thread: it results in better performance;
    - requires lots of string manipulations.
  - **Synchronised methods**: only one thread at a time can access them!
  - **Thread**: a given "thread" of execution.
    - Java allows for multithreading, i.e. the capability of performing several tasks simultaneously.

Synchronisation and Multithreading: out of scope for this course!

Character Class

- The **Character** class is a Wrapper class for a single character, and belongs to the **java.lang** package.
  - Some of its static methods:

```
isLetter(char c)        //   isDigit(char c)
isUpperCase(char c)     //   isLowerCase(char c)
```

  - You can create a **Character** object from a **char** value:

```
Character myCharacter = new Character('p');
char c = myCharacter.charValue(); // char value wrapped in
                                  // the Character object
```

- **Examples** using the **Character** class:

```
Character myCharacter = new Character('c');
```

```
myCharacter.compareTo(new Character('f')); // returns -3
myCharacter.compareTo(new Character('a')); // returns 2
myCharacter.equals(new Character('e')); // returns false
Character.isLetterOrDigit(new Character('?')); // returns false
```

`StringTokenizer` Class

- The **`StringTokenizer`** class is used to extract tokens and process text from a string, and it belongs to the **`java.util`** package.
  - It breaks strings into several pieces, e.g. a line of text can be broken into substrings each containing a word.

```
1 string      → "I like learning Java"
4 substrings  → I     like     learning     Java
```

  - Individual words are identified by using individual characters as delimiters.
  - The substrings resulting from breaking a string into several pieces are known as tokens.
  - Delimiters are specified in **`StringTokenizer`** constructors; the default ones are *space*, *tab*, *new line* and *carriage return*.

`StringTokenizer` is a legacy class; should **not** be used with new code, but should be understood in relation to existing/old code. Instead, use **`String.split()`** …

`StringTokenizer` Constructors

- Constructors:

```
StringTokenizer(String s)
StringTokenizer(String s, String delimeters)
StringTokenizer(String s, String delimiters, boolean returnDelimiters)
```

`StringTokenizer` for a string with default delimiters; doesn't count delimiters as tokens.

`StringTokenizer` for a string with specified delimiters; doesn't count delimiters as tokens.

`StringTokenizer` for a string with specified delimiters; may count delimiters as tokens.

Good programming practice (usually): to provide a no-arguments constructor for a class. **`StringTokenizer`** doesn't have a no-arguments constructor, because a **`StringTokenizer`** object must be created for a string!

Using StringTokenizer

```
String s = "I am from Portugal.";

// Create a StringTokenizer.
StringTokenizer myTokenizer = new StringTokenizer(s);

System.out.println("Number of tokens is " +
                   myTokenizer.countTokens() + ".");
while (myTokenizer.hasMoreTokens())
  System.out.println(myTokenizer.nextToken());
```

Output is …

```
Number of tokens is 4.
I
am
from
Portugal.
```

- **Other Examples**:

```
StringTokenizer myTokenizer = new StringTokenizer(s, "nu");
StringTokenizer myTokenizer = new StringTokenizer(s, "nu", true);
```

What is the **output**?

How would you rewrite this code using **`String.split()`**?

Practice Exercise 1

- Answer the following questions:
  - Define the two terms: *token*, *delimiter*.

  - What is the output of this code:

```
StringTokenizer stuff = new
                    StringTokenizer("abc,def,ghi");
System.out.println(stuff.nextToken());
```

  - What is the output of this code:

```
StringTokenizer stuff = new
                  StringTokenizer("abc,def,ghi", ",");
System.out.println(stuff.nextToken());
```

`Scanner` Class (*from Java 5.0*)

- Delimiters are single characters in the **`StringTokenizer`** class; however, the **`Scanner`** class (of **`java.util`** package) allows a *word* to be specified as a delimiter.
- Example:

```
String s = "Let your heart guide you.";
Scanner myScanner = new Scanner(s);
myScanner.useDelimiter("you");
while (myScanner.hasNext())
  System.out.println(myScanner.next());
```

Output is …

```
Let
r heart guide
.
```

returns **`true`** if there are tokens left

returns a token as a string

A word can be a single character, so **`Scanner`** can specify a single character delimiter!

`Scanner` Class: Other Uses

- Scanning primitive type values: several methods can be used to obtain a token with a primitive data type value.
  - **Example**:

```
String s = "1 10 100 1000";
Scanner myScanner = new Scanner(s);
int sum = 0;
while (myScanner.hasNext()) { sum += myScanner.nextInt(); }
System.out.println("Sum = " + sum);
```

- Reading console input
  - **Example**:

```
System.out.print("Please enter an int value: ");
Scanner myScanner = new Scanner(System.in);
int i = myScanner.nextInt();
```

To scan a string with multiple single characters as delimiters, use **`String.split()`**. To use a word as the delimiter, use **`Scanner`**.

The Format Specifier

% [argument_number$] [flags] [width] [.precision] type

```
String.format("I have %,6.2f bugs to fix", 12345.6789);
```

which argument it refers to, if there's more than one

minimum number of characters used

special formatting options, e.g. *inserting commas*, putting *negative numbers in parentheses, left/right alignment*

number of decimal places

%d – decimal
%f – floating point
%x – hexadecimal
%c – character

What about when there's more than one argument?

Output is …

```
I have 12,345.68 bugs to fix
```

Example: `printf` *versus* `println`

```
public class TestPrintMethods {
  public void printSomeStrings(String firstName, String lastName,
                               int numPets, String petType) {
    System.out.printf("Using printf: %s %s has %d %s.\n",
                      firstName, lastName, numPets, petType);
    System.out.println("Using println: " + firstName + " " +
                       lastName + " has " + numPets + " " +
                       petType + ".");
  }
  public static void main(String args[]) {
    TestPrintMethods test = new TestPrintMethods();
    test.printSomeStrings("John", "Doe", 7, "chickens");
  }
}
```

Output …

```
Using printf: John Doe has 7 chickens.
Using println: John Doe has 7 chickens.
```

Example: Controlling Width and Precision

```
public void printSomeSalaries() {
  CEO[] softwareCEOs = { new CEO("Jeff Bezos", 567.986323),
                         new CEO("Larry Ellison", 6789.0),
                         new CEO("Bill Gates", 78901234567890.12) };
  System.out.println("SALARIES:");
  for (CEO ceo: softwareCEOs) {
    System.out.printf("%15s: $%,8.2f%n", ceo.getName(), ceo.getSalary());
}
```

```
public class CEO {
  private String name;
  private double salary; // In millions ($).
  public CEO(String name,double salary) {
    this.name = name; this.salary = salary;
  }
  public String getName() { return(this.name); }
  public double getSalary() { return(this.salary); }
}
```

Output …

```
SALARIES:
     Jeff Bezos: $  567.99
  Larry Ellison: $6,789.00
     Bill Gates: $78,901,234,567,890.12
```