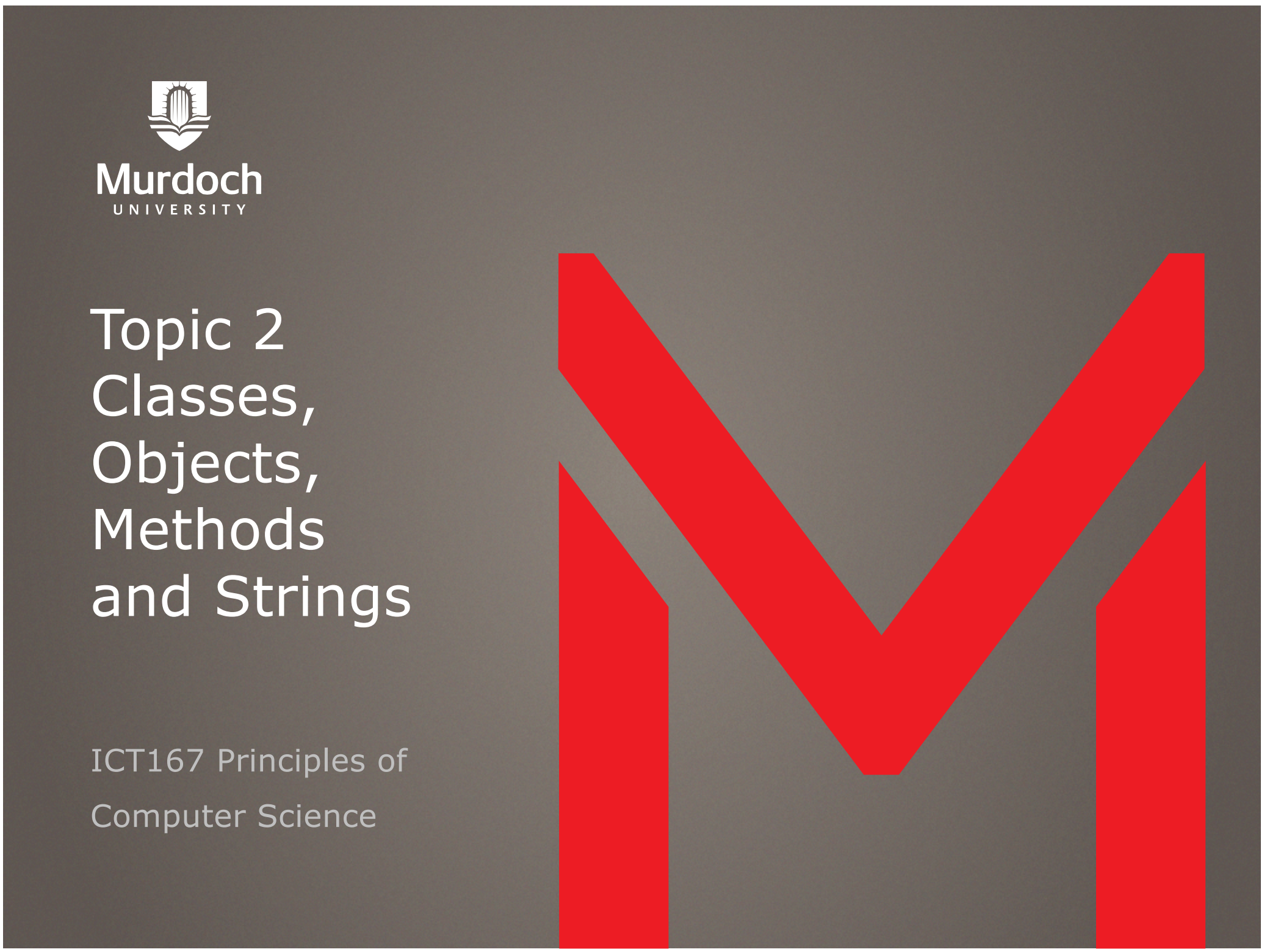


Topic 2 Classes, Objects, Methods and Strings

ICT167 Principles of Computer Science

Murdoch UNIVERSITY

Objectives

- Briefly explain how **O-O design** supports better design
- Describe the difference between an **object** and a **class**
- Be able to declare **String variables** and create new **String objects**
- Know how to use basic String methods such as **length()**, **trim()**, **charAt()**, **substring()**, **equals()**, **equalsIgnoreCase**, **compareTo()** and **indexOf()**

Objectives

- Know how to find and extract information from the Java **on-line library** class documentation
- Know what the two main types of **variables** are in Java
- Be able to set up a separate **client class**
- Describe the main components of a **class definition**
- Describe the UML notation for a class
- Know what an **instance variable** is and what is it for

Objectives

- Explain what is done by the following line of code: `MyClass c = new MyClass();`
- Explain what is meant by "**invoking a method**"
- Be able to classify **variables** as **local** or **instance**
- Be able to define and use a method with a **parameter**
- Explain in what way Java uses **call-by-value** for primitive types
- The use of array in Java (introduction)

Reading - Savitch: Chapters 2.2, 5.1, 7,1

Recommended self test questions:

Chapters 5.1

Murdoch UNIVERSITY

Object-Oriented Design Methodology

- A systematic way to develop better software designs
- May be supported by a programming language
- OOP = Object-Oriented Programming; that is, programming in an O-O way
- Began in 1960s with the language Simula for simulation programs
  - Became popular in early 1990s with C++ for general purpose programming

Object-Oriented Design Methodology

- Now it is the main way of constructing complex software
- C++ is widely used, Java is being used at a hugely increasing rate
- The main characteristic of OOP is the use of Objects and Classes

Object-Oriented Design Methodology

- The purpose is to support better design by:
  - Supporting ***abstraction***, in particular data abstraction, i.e. being able to organize the data in a program, by lumping related data together and having a simple name for a big lump. Associated procedures can get lumped in together too (also called **Abstract Data Type - ADT**)
    - Eg: we can represent all the details concerning a student (name, number, units enrolled in, etc.) together with the operations (initialise, update, print, etc.) in one class

Object-Oriented Design Methodology

- The purpose is to support better design by:
  - Supporting ***re-use***, allowing programmers to reuse their own and other people's code (in more sophisticated ways than cut, paste and edit)
  - Encouraging the design of code appropriate to a particular area (or ***problem domain***) rather than a particular task

What is an Object?

- An object (in real world as well as in software) represents an identity that can be distinctly identified
- Eg:
  - a person
  - a bank account
  - a house
  - a species
  - a list
  - a button etc.

What is an Object?

- An object has:
  - ***identity*** - it acts as a single whole
  - ***state*** - it has various properties (a set of data fields with their current values) that might change
  - ***behaviour*** - it can "do things" and can have things done to it
    - A software object does something when one of its methods is called

What is a Class?

- When a Java application is running, its objects are created and their methods are invoked
- To create an object, there needs to be a description of it
- A ***class*** is a description of a kind of object (it is a construct that defines objects of the same type)
  - Programmers may define their own classes or may use predefined classes that come in class libraries, or do both

What is a Class?

- A class is merely a plan for a possible object (or objects)
- A class does not itself create an object
  - An object is created when the operator ***new*** is used with the name of the class
- Creating an object is called ***instantiation***

Figure 5.1 A class as a blueprint

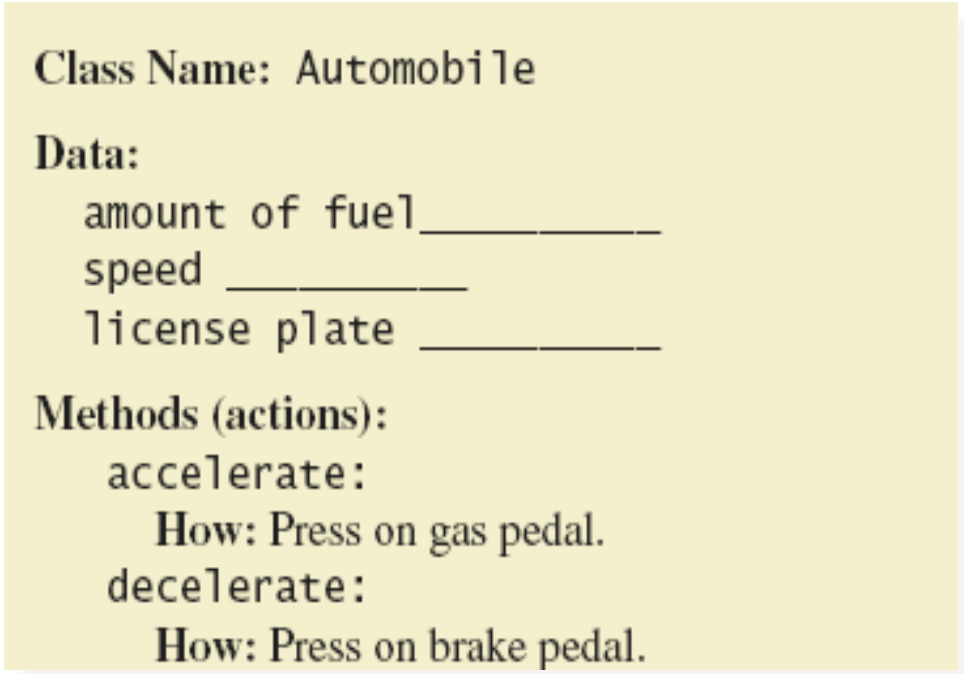
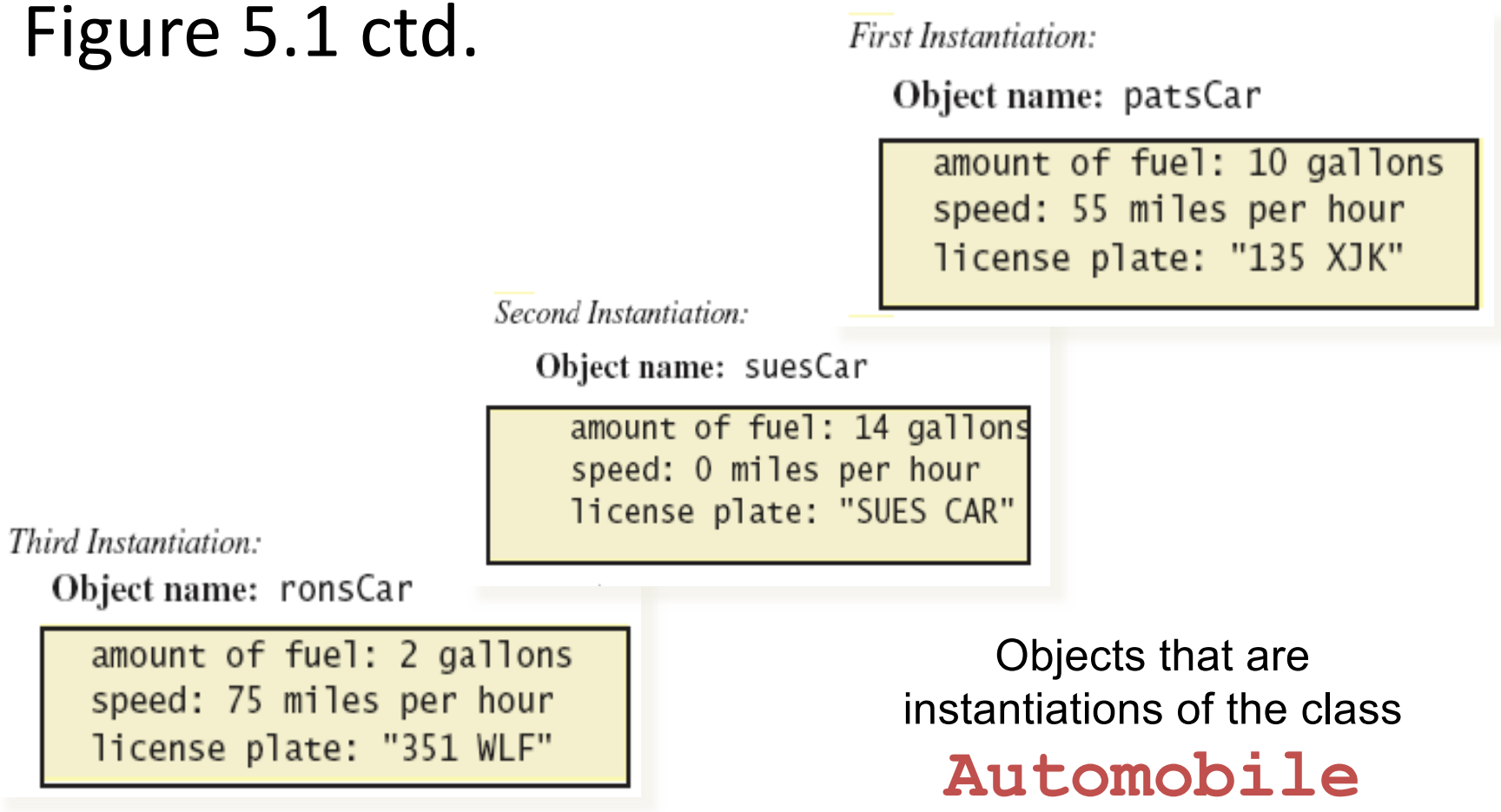


Figure 5.1 ctd.



Objects that are instantiations of the class **`Automobile`**

Murdoch UNIVERSITY

Strings in Java

- We have already used constants of type **String**, eg: "Enter two numbers separated by a space character"
- A value of type **String** is a sequence of characters treated as a single item
- The class **String** is found in the `java.lang` library and is automatically available to all Java programs

Strings in Java

- You can declare a String variable by:

  `String s;` OR

- Declare and initialize a String variable by:

  `String s = "Hello";`

- The above statement is an abbreviation for:

  `String s = new String("Hello");`

- These declare the variable `s`, make a new String object with current contents "Hello" and get `s` to refer to that object

Strings in Java

- You can concatenate two or more strings using the '+' operator (called the **concatenation operator**)

```
s = s + " World";
System.out.println("Result: " + s);
int count = 193;
s = s + " of " +count+ " Countries!";
System.out.println(s); // what is the output?
```

Strings in Java

- String is a special class because:
  - It is built-in
  - The compiler automatically recognizes String constants (i.e. quoted text)
  - You don't have to use **new** to get a new String
  - There is a concatenation operator (+) which can be applied to strings
  - And no methods allow you to change the value of a **String** object
- However, String is also a class like any other class

String Class Methods

- There are many useful methods in the String class. For example:

  `int` **`length()`** returns the length of the String

  Eg:

```
String s = "Hello";
int a = s.length();
```

- The value returned by length() method will be 5
- You can use a call to method **length()** anywhere an **int** can be used. Eg:

```
System.out.println("Length is " + s.length());
```

String Class Methods

- **Note that strings in Java start at position 0 and end at position length()-1**
- `String` **`trim()`** returns a String which is this String with leading and trailing white spaces removed

  Eg:

```
String numInPlus= " 2.5    ";
String numIn= numInPlus.trim();
```

String Class Methods

- `char` **`charAt`**`(int pos)` returns the character at position indicated by its argument `pos`. Note that the first character is at position 0
- Eg:

```
String myStr = "Computer Science";
char ch = myStr.charAt(5);
```

returns the character `'t'` in variable `ch`

String Class Methods

- `String` **`substring`**`(int start, int end)` returns the substring starting at position `start` and ending one character before position `end`
- Eg:

  `String sub1 = myStr.substring(2, 4);` returns "mp" in String variable sub1 (assuming that `myStr` has the value “Computer Science”)

String Class Methods

- `String` **`substring`**`(int start)` returns the substring starting at position `start` of this string through to the end of the string
- Eg:

```
String sub2 = myStr.substring(9);
```

returns "Science" in String variable sub2 (assuming that `myStr` has the value “Computer Science”)

String Class Methods

- `boolean` **`equals`**`(String other)` returns whether or not this String has the same value as the `other` String
- Eg:

```
s.equals("Hello")
```

- `boolean` **`equalsIgnoreCase`**`(String other)` behaves like `equals` but regards upper and lower case versions of a letter to be the same

String Class Methods

- `int` **`compareTo`**`(String other)` compares `this` String to the `other` String and returns:
  - 0 if they have the same value
  - a negative number if `this` String comes **before** the `other` String in the lexicographic (dictionary) ordering
  - a positive number otherwise

String Class Methods

- Eg:

```
System.out.println("abc".compareTo("abc"));
```

- // will output the value 0

```
System.out.println("abc".compareTo("bac"));
```

- // will output a negative number

```
System.out.println("xyz".compareTo("def"));
```

- // will output a positive number
- **`compareToIgnoreCase(`**`other`**`)`** is also available, which compares two strings lexicographically, ignoring case differences

String Class Methods

- `int` **`indexOf`**`(String other)` returns the index of first occurrence of substring `other` within this String. Returns -1 if substring `other` is not found
- `String` **`replace`**`(char oldChar, char newChar)` returns a new string having the same characters as this string, but with each occurrence of oldChar replaced by newChar
- And many more (eg: `toLowerCase()`, `toUpperCase()`, `lastIndexOf(String other)`, `...`)

String Class Methods

- If you want to find out what methods are available (and exactly what they do, and how to call them, etc.) for the String class or any other Java library class then you can look up the Java on-line documentation
- Java documentation is provided on the Web by Oracle at:

  http://docs.oracle.com/javase/8/docs/

String Class Methods

- When you find the right Class in the right library you will get
  - an overview of the Class
  - a summary of the Methods and
  - a list of the details of the methods
- **It is strongly recommended that you become familiar with using the documentation**

Example

```
// File: TestString.java
class TestString {
  public static void main( String[] args ){
  // str1 and str2 are variables referring to an object,
  // but the objects do not exist yet.
     String str1;
     String str2;
  // len1 + len2 are two primitive variables of type int
     int len1, len2;
  // create an object of type String
     str1 = new String("Computer Science");
  // create another object of type String
     str2 = new String("Games Technology");
```

Example

```
// invoke the objects length() method
   len1 = str1.length();
   len2 = str2.length();
   System.out.println("The string \"" + str1 +
        "\" is " + len1 + " characters long");
   System.out.println("The string \"" + str2 +
        "\" is " + len2 + " characters long");
```

Example

```
// compare strings with equals() method
  if (str1.equals(str2))
     System.out.println("\nThe two strings
                        are equal (same).\n");
  else
     System.out.println("\nThe two strings
          are not equal (not same).\n");
```

Example

```
    // compare strings with compare() method
    if (str1.compareToIgnoreCase(str2)< 0)
       System.out.println("\nThe two strings
       \"" + str1 + "\" and \"" + str2 + "\"
            are in alphabetical order.\n");
    else
       System.out.println("\nThe two strings
       \"" + str1 + "\" and \""+ str2 + "\"
       are not in alphabetical order.\n");
  }// end of main
}// end of class TestString
```

Output

```
/* OUTPUT
The string "Computer Science" is 16 characters
    long
The string "Games Technology" is 16 characters
    long
The two strings are not equal (not same).
The two strings "Computer Science" and "Games
Technology" are not in alphabetical order.
*/
```

Output

- Note that == is not appropriate for determining if two String objects have the same value
- Eg:

  `if (str1 == str2)` ... determines only if `str1` and `str2` refer to a common memory location
- If `str1` and `str2` refer to strings with identical sequences of characters, but are stored in different memory locations then `(str1 == str2)` will yield false

Objects, Methods and Classes

- An O-O design has Objects in it
  - The Objects get created, have their properties changed, change the properties of other Objects etc., as the program runs
- The designer / programmer chooses what sorts of objects and how many of each sort are used in the program

Objects, Methods and Classes

- For example, an Object may be:
  - a person, or all the data about a particular person, in a program that manages some aspect of an organization
  - a visible component on a GUI
  - a chemical formula
  - or something really complex like a list of classes of school children

Objects, Methods and Classes

- Objects belong to Classes
- A class:
  - Specifies the kinds of *data* an object of the class can have
  - Provides *methods* specifying the actions an object of the class can take

Objects, Methods and Classes

- There may be several Classes of Objects involved in a particular program
  - In a particular run of that program no., one, several or many Objects belonging to a particular Class might be used
- An Object may own a more or a less complicated bunch of data
  - The values may change as the program runs

Objects, Methods and Classes

- However, the sorts (types) of data are fixed for that Object and are the same for all Objects of the same Class
- An Object can do certain things
  - There is a fixed set of Methods (like procedures) available to it
  - Every Object in the same Class has the same Methods

Primitive Type Variables vs Class Type Variables

- Each variable in a Java program has to be declared to be of a particular type
- The variable may be of a *primitive type* (like int, boolean, double, char etc) or of a *Class type*
- The Class type variables must be declared to be of a particular Class type. Eg: String, Button or an Array of something or some programmer defined class like Sheep

Primitive Type Variables vs Class Type Variables

- The variable will then be able to refer to a particular Object belonging to that Class
- It may sometimes refer to no Object (a null reference) and it may sometimes change which Object it refers to (eg: by assignment) but it is only allowed to refer to Objects belonging to that Class. (*Later we see that this is not quite true)

Class Files, Clients and Separate Compilation

- Many Classes may be necessary to solve a particular problem
- We may want to write several of our own which use each other, use library Classes, use other people's Classes or let other people use our Classes
- A **programmer** or another Class which uses one of our Classes may be called a **client**

Class Files, Clients and Separate Compilation

- The most basic set up is to have only one Class per file
- Remember that a file called `MyClass.java` should contain source code for a class called `MyClass`
- The compiled bytecode will be kept in a file called `MyClass.class`

Class Files, Clients and Separate Compilation

- In ICT167, you will most often be acting as your own client
- If you use `MyClass` in a Class called `MyClientClass` then it is simplest to put `MyClientClass.java` in the same directory as `MyClass.java`.
- When you compile `MyClientClass.java` then the compiler will find the compiled version of `MyClass` and there should be no problem

Class Files, Clients and Separate Compilation

- Note that your (`myClass`) does not have to have a main method
- If you try running such a Class

  ```
  java MyClass
  ```

- you'll get
- Exception ... no such method: main
- `MyClass` may be designed only to be used by clients

Example Class


```
import java.util.*;
public class SpeciesFirstTry {
   public String name;
   public int population;
   public double growthRate;

   public void readInput( ) {
      Scanner keyboard = new Scanner(System.in);
      System.out.println("What species' name?");
      name = keyboard.nextLine( );
      System.out.println("What is the population
                                 of the species?");
      population = keyboard.nextInt( );
```

Example Class

```
   while (population < 0) {
      System.out.println("Population must not
                          be negative.");
      System.out.println("Re-enter
                             population:");
         population = keyboard.nextInt( );
   }// end while
   System.out.println("Enter growth rate
               (percent increase per year):");
   growthRate = keyboard.nextDouble( );
} // end readInput
```

Example Class

```
public void writeOutput( ) {
   System.out.println("Name = " + name);
   System.out.println("Population="+population);
   System.out.println("Growth rate = " +
                                 growthRate + "%");
}
```

Example Class


```
public int getPopulationIn10( ) {
   int result = 0;
   double populationAmount = population;
   int count = 10;
   while ((count >0)&&(populationAmount >0)){
      populationAmount = (populationAmount +
         (growthRate/100) * populationAmount);
      count--;
   }// end while
```

Example Class

```
        if (populationAmount > 0)
           result = (int)populationAmount;
        return result;
    }// end getPopulationIn10
}// end class SpeciesFirstTry
```

Example Client


```
public class SpeciesFirstTryDemo {
  public static void main(String[] args) {
    SpeciesFirstTry speciesOfTheMonth = new
                                SpeciesFirstTry();
    System.out.println("Enter Species data:");
    speciesOfTheMonth.readInput();
    speciesOfTheMonth.writeOutput();
    int futurePopulation =
       speciesOfTheMonth.getPopulationIn10();
    System.out.println("In ten years the
      population will be "+futurePopulation);
```

Example Client

```
//change the species to show how to change
//the values of instance variables
  speciesOfTheMonth.name = "Klingon ox";
  speciesOfTheMonth.population = 10;
  speciesOfTheMonth.growthRate = 15;
  System.out.println("The new Species of the
                                    Month:");
  speciesOfTheMonth.writeOutput();
  System.out.println("In ten years the
    population will be " +
    speciesOfTheMonth.getPopulationIn10());
  }// end main
}// end class SpeciesFirstTryDemo
```

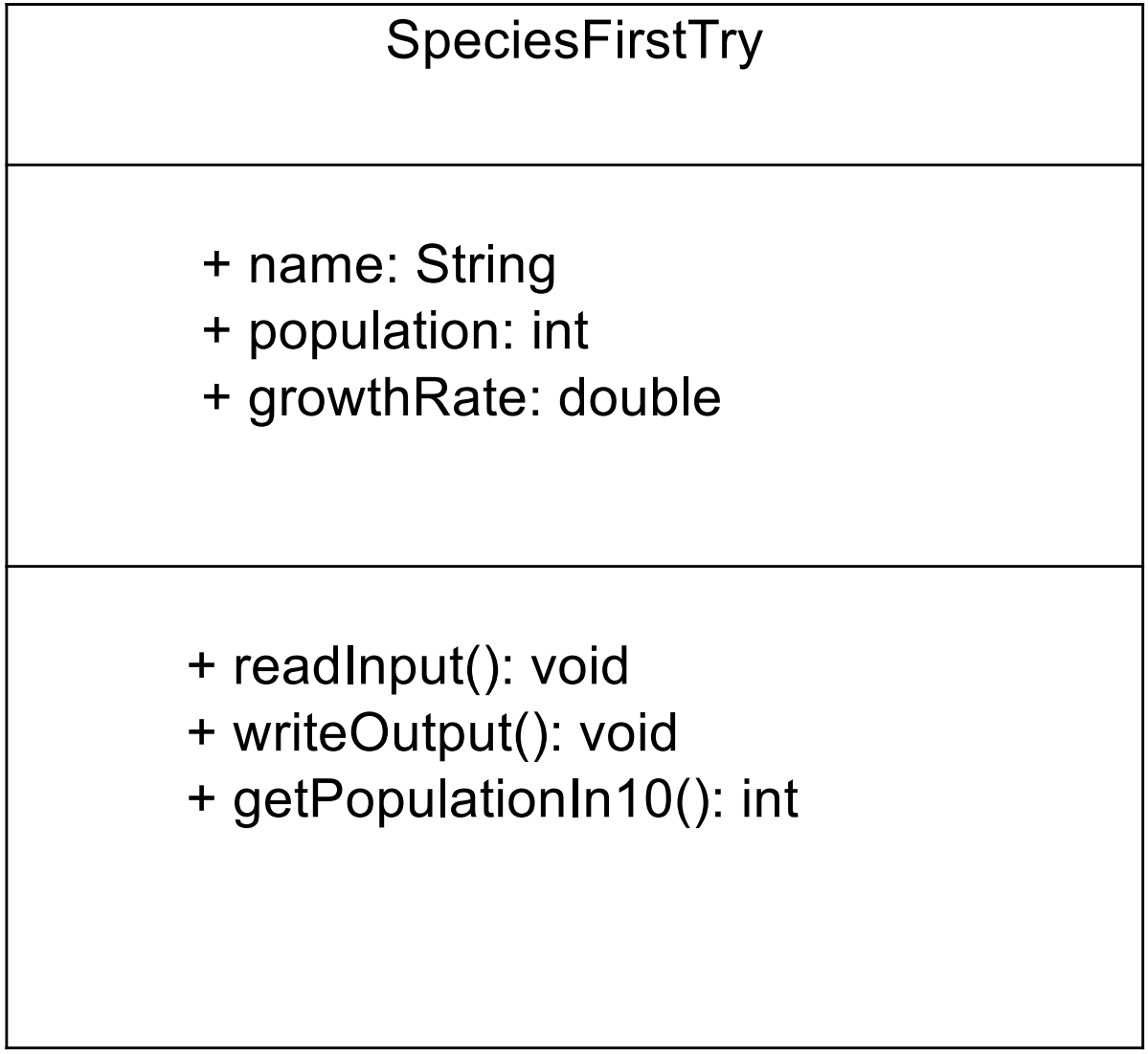
Class Definitions

- Look at the definition of the class `SpeciesFirstTry`
- This is supposed to supply all the code belonging to any Object of that class
- We see:
  - The **class name** (and an access modifier saying that it is a publically usable class)
  - Three **instance variables** (what data each Object has) and
  - Three **method definitions** (what can be done by Objects of the Class)

Class Definitions

- The instance variables and methods are sometimes (confusingly) called members of the Object
- **Note:**
  - Each class definition should be in a separate file
  - Use the same name for the class as the file, except add ".java" to the file name (Java is case sensitive!)
  - As a good programming practice, start the class (and file name) with a capital letter and capitalise the first letter of inner words. Eg: `SpeciesFirstTry.java` for the class `SpeciesFirstTry`

A UML Class Diagram



← Class name

← Data (instance variables)

← Methods (actions)

A UML Class Diagram

- Universal Modelling Language (UML) outlines the definition of a class diagrammatically
- UML diagrams are mostly self-explanatory
- A plus sign (+) indicates a **public** instance variable or method
- A minus sign (-) indicates a **private** instance variable or method
- Typically, the class diagram is created before the class is defined

Instance Variables

- `SpeciesFirstTry` class has three instance variables: name, population, and growthRate
- The accessibility, types, and names of these instance variables are declared:

```
public String name;
public int population;
public double growthRate;
```

Instance Variables

- **public** means that there are no restrictions on how these instance variables are used
  - They can be looked at and/or changed by a client
  - The client just needs to specify which instance variable of which Object is being accessed
  - Eg: `speciesOfTheMonth.population = 10;`
- Later we will see that these should be declared as private instead of public

Instance Variables

- The above declaration means that each `SpeciesFirstTry` Object owns some data, a `String` called `name`, an `int` called `population` and a `double` called `growthRate`
- The actual values for a particular `SpeciesFirstTry` Object may change as the program runs
- And a different `SpeciesFirstTry` Object may have different values
  - But the types of the values are fixed

Instance Variables

- In the main method in the client class, `speciesOfTheMonth` is a variable which refers to a `SpeciesFirstTry` Object
- So this object (like any other in that class) has the three instance variables with particular values
- So, it has an `int` called `population`
  - The above statement changes the value of that `int` to 10

Instance Variables

- If an Object referred to as **`X`** has an instance variable **`var`** of any type then **`X.var`** can be used wherever any other variable of that type could be used

What's New?

- Notice a couple of uses of **new** in the client program. Eg:

```
SpeciesFirstTry speciesOfTheMonth
                          = new SpeciesFirstTry();
```

- We will learn more about `new` later. Roughly ...
- The above statement does two main things. It is like the usual declaration and initialization statements for a new primitive variable. Eg:

```
int b = 2;
```

- We declare the variable type and give it a value

What's New?

- Here three actions are carried out:
  - We declare `speciesOfTheMonth` to be a variable of (Class) type `SpeciesFirstTry`
  - We create a new Object of that type using **`new`**
  - The variable now refers to that new Object
- So `new SpeciesFirstTry()` creates a new `SpeciesFirstTry` Object with its own three instance variable values
- We can go on to use that object later in the program because we also have a reference to it

Using Methods

- A method is an action that an object can take
  - Which methods are available to a particular Object depends on its class
  - Eg: `SpeciesFirstTry` objects have three methods
- A client may want, at a particular moment, to get a particular Object to do a particular action
- This is called **invoking the method**, or **calling the method** or **passing a message to the object**

Using Methods

- The client needs to specify (via a reference) the Object being called and the method name (with a dot between)
- The method, like a procedure, may have parameters and may have a return value
- If the method has no parameters then you still need to put parentheses() after the method name
  - The method may return no value, i.e. if it is a void method

Using Methods

- If the method returns a value then the client may want to use that
- Here are some example invocations:

```
speciesOfTheMonth.readInput();
int futurePopulation =
  speciesOfTheMonth.getPopulationIn10();
System.out.println("In ten years the
  population will be " +
  speciesOfTheMonth.getPopulationIn10());
```

Instance vs Local Variables

- In the client code there are two sorts of variables:
  - **instance variables** such as `speciesOfTheMonth.population`

    (declared in a class definition outside any method)
  - and **local variables** such as `futurePopulation` (declared in a method)
- It is easy to see the difference in usage (note the dot)

Instance vs Local Variables

- But now look at the body of a method. See

```
double populationAmount = population;
int count = 10;
```

- Notice two local variables. But also notice that the instance variable has lost its dot
- Whose population is being used here? Remember, every `SpeciesFirstTry` Object has its own population

Instance vs Local Variables

- The answer is that the method body will only be executed when the method is called by a client on a particular object, and it is that object's population which is used
  - The calling object is assumed to own these
- Eg: call `speciesOfTheMonth.getPopulationIn10()` and it will be `speciesOfTheMonth.population` which is used here

Instance vs Local Variables

- Note that you can write `this.population` in a method if you want to (or sometimes need to)
- **`this`** refers to the calling object

Example

- Replace `getPopulationIn10()` in `SpeciesFirstTry` by the following method:

```
public int predictPopulation(int years) {
   int result = 0;
   double populationAmount = population;
   int count = years;
   while ((count > 0)&&(populationAmount > 0)){
      populationAmount = (populationAmount +
            (growthRate/100) * populationAmount);
      count--;
   }
```

Example

```
  if (populationAmount > 0)
     result = (int)populationAmount;
  return result;
}// end predictPopulation method
```

Example


```
SpeciesSecondTryDemo

/** Demonstrates the use of a parameter with the

method predictPopulation */
public class SpeciesSecondTryDemo {
   public static void main(String[] args) {
      SpeciesSecondTry speciesOfTheMonth
                              = new SpeciesSecondTry();
      System.out.println("Enter data on the
                               Species of the Month:");
      speciesOfTheMonth.readInput();
      speciesOfTheMonth.writeOutput();
      int futurePopulation =
        speciesOfTheMonth.predictPopulation(10);
```

Example

```
    System.out.println("In ten years the
  population will be " + futurePopulation);
  //change the species to show how to change
  //the values of instance variables
    speciesOfTheMonth.name = "Klingon ox";
    speciesOfTheMonth.population = 10;
    speciesOfTheMonth.growthRate = 15;
    System.out.println("New Species of Month:");
    speciesOfTheMonth.writeOutput();
    System.out.println("In ten years the
          population will be " +
     speciesOfTheMonth.predictPopulation(10));
   }// end main
}// end class SpeciesSecondTryDemo
```

Example With Parameter

- `SpeciesSecondTry` is much more useful
  - We can project population any number of years ahead (and not just 10)
- To do so, we use a parameter
- So the method definition has a formal parameter, here an `int` called `years`, and the method must be invoked by supplying an `int` argument
- Eg:

```
speciesOfTheMonth.predictPopulation(10);
```

Example With Parameter

- The argument may be an `int` constant (10) or an `int` variable or any expression of `int` type
- When the method is invoked the current argument value is given as the initial value of the formal parameter which acts like a local variable in the method body
- Note that in general, there may be many parameters for a method and they may be of various types, including primitive types, and class types (and arrays)

Example With Parameter

- The types, number and order of the arguments must match exactly
- This allows current values to be transferred across to matching formal parameters when the method is invoked

Call-by-value on Primitives

- Suppose that we have a method with some primitive parameters. Eg:

```
public int multAndInc(int x, int y){
  int ans = x*y;
  x = x+1;
  return ans;
}
```

- We can call it with a variable argument. Eg:

```
int a = 2;
int b = 3;
int c = X.multAndInc(a,b);
```

Call-by-value on Primitives

- Suppose that we change the value of a parameter in the body of the method
  - Then the value of the variable argument is NOT changed
- In this example, the value of `a` is not changed

Call-by-value on Primitives

- The situation may be similar or different in other programming languages
- We summarize the situation by saying that for primitive types, Java uses ***call-by-value***
- Only the current *value* of the argument is passed over to the formal parameter
- There is no more lasting association

Arrays in Programming Languages

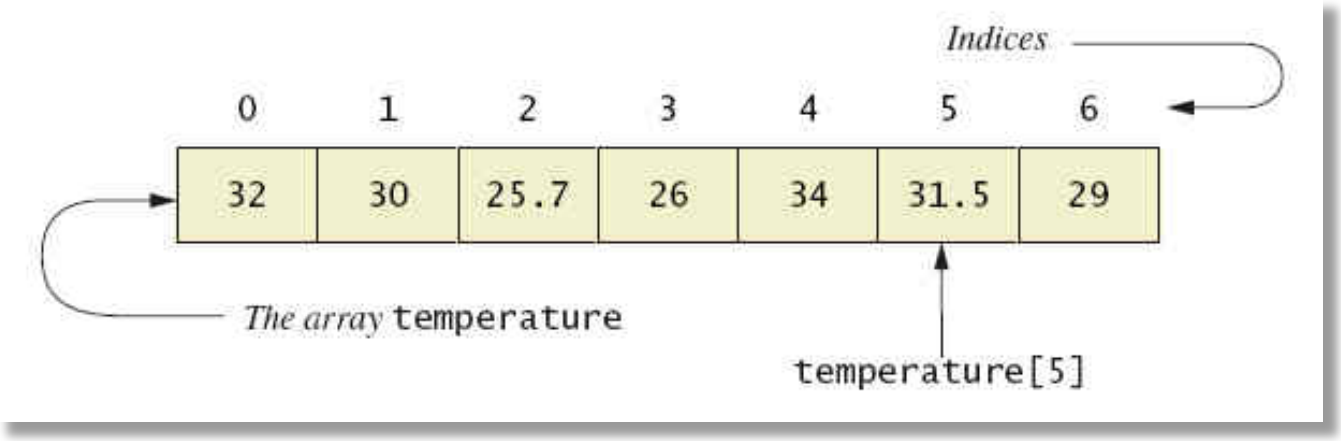
- An array consists of a systematically organised and named sequence of similar variables - called the **elements** of the array
  - That is, it is a single name for a collection of data values, all of the same type
- The elements are numbered: 0, 1, 2, … and so on, called the **index** (or subscript)
- An array is used in place of a lot of separate variables (which are of the same type)
- You have seen String earlier, and it can be considered as an array of characters

Arrays in Programming Languages

- An array can be small with only 2 or 3 elements (or even zero), or it can be very large with thousands of elements
- An array is an *ordered collection* of data items
- Each item has a position (or index)
- Each item (except first item) has a unique predecessor
- Each item (except last item) has a unique successor

Visualize Array

- Figure 7.1 A common way to visualize an array



- Note sample program, listing 7.1
  `class ArrayOfTemperatures`

Murdoch UNIVERSITY

Creating Arrays in Java

- General syntax for declaring an array:

```
BaseType[] ArrayName= new BaseType[Length];
```

- Examples:

```
// 80-element array with base type char
char[] symbols = new char[80];
// 100-element array of doubles:
double[] readings = new double[100];
//100-element array of Species:
Species[] specimen = new Species[100];
```

Creating Arrays in Java

- Length of an array is specified by the number in brackets when it is created with ***new***
  - it determines the amount of memory allocated for the array elements (values)
  - it determines the *maximum* number of elements the array can hold
  - storage is allocated whether or not the elements are assigned values

Creating Arrays in Java

- The array length is established when the array is created
  - It is automatically stored in the (read-only) instance variable length, and cannot be changed
- An array is a special kind of object in Java
- Eg: declare an array of ints:

```
int[] mark;

// mark is now an “array of int” type variables, with
// null reference
```

Creating Arrays in Java

- Create an array of int “objects” of a certain length:

```
mark = new int[7];
```

// the variable mark now refers to an array of seven ints
// each one initialised to the default int value of zero

- OR, declare and create:

```
int[] mark = new int[7];
```

- Data can now be stored in the array as:

```
mark[0] = 85;
```

Creating Arrays in Java

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
mark:	85	70	50	62	39	92	54

- You should have covered the concept of Arrays in your previous unit, more details of Arrays in Java will be covered in Topic 6.

End of Topic 2

Murdoch UNIVERSITY