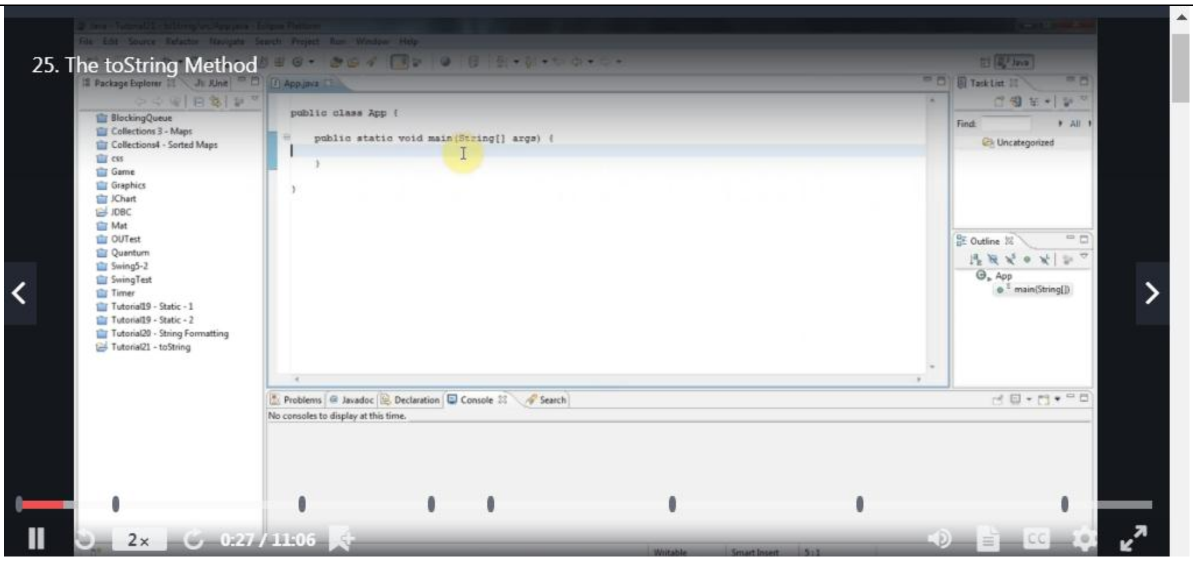
JAVA REPORT

Date:	**11/06/2020**	**Name:**	**SAFIYA BANU**
Course:	**JAVA**	**USN:**	**4AL16EC061**
Topic:	1. The toString Method 2. Inheritance 3. Packages 4. Interfaces 5. Public, Private, Protected 6. Polymorphism 7. Encapsulation and the API Docs 8. Casting Numerical Values 9. Upcasting and Downcasting 10. Using Generics	**Semester & Section:**	**8TH B**
Github Repository:	**Safiya-Courses**		

AFTER NOON SESSION DETAILS



```
public class App {

    public static void main(String[] args) {

    }

}
```

1. The toString Method

The method is used to get a String object representing the value of the Number Object.

If the method takes a primitive data type as an argument, then the String object representing the primitive data type value is returned.

If the method takes two arguments, then a String representation of the first argument in the radix specified by the second argument will be returned.

Syntax

Following are all the variants of this method −

String toString()
static String toString(int i)

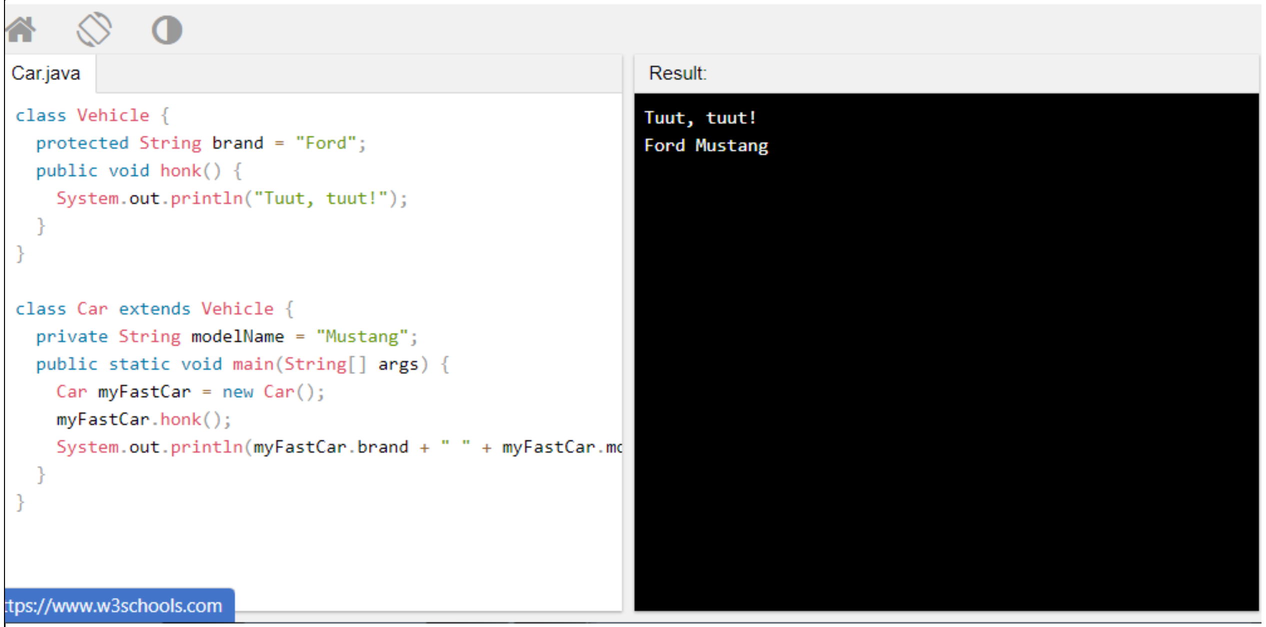
2.Inheritance

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- subclass **(child) - the class that inherits from another class**
- superclass **(parent) - the class being inherited from**

To inherit from a class, use the extends keyword.



Car.java

```
class Vehicle {
  protected String brand = "Ford";
  public void honk() {
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";
  public static void main(String[] args) {
    Car myFastCar = new Car();
    myFastCar.honk();
    System.out.println(myFastCar.brand + " " + myFastCar.mo
  }
}
```

Result:

```
Tuut, tuut!
Ford Mustang
```

tps://www.w3schools.com

2. Packages

A package in Java is used to group related classes. Think of it as **a folder in a file directory**. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the import keyword:

Syntax

```
import package.name.Class;   // Import a single class

import package.name.*;   // Import the whole packages
```

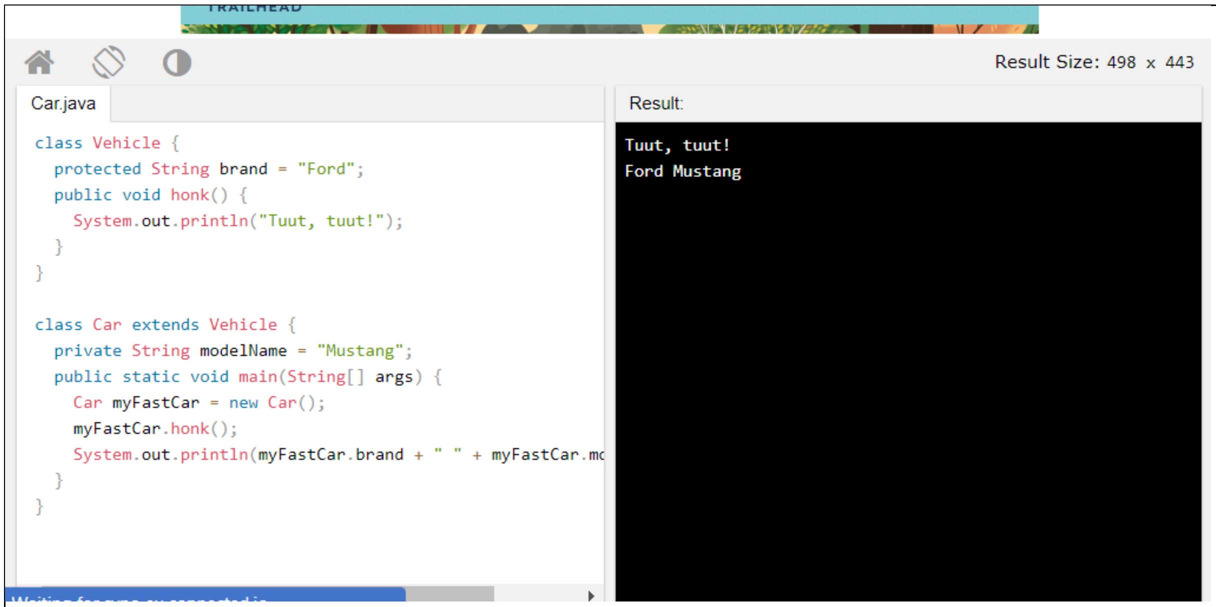
4.Interfaces

Java Inheritance (Subclass and Superclass)

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.



```
class Vehicle {
  protected String brand = "Ford";
  public void honk() {
    System.out.println("Tuut, tuut!");
  }
}

class Car extends Vehicle {
  private String modelName = "Mustang";
  public static void main(String[] args) {
    Car myFastCar = new Car();
    myFastCar.honk();
    System.out.println(myFastCar.brand + " " + myFastCar.mo
  }
}
```

```
Tuut, tuut!
Ford Mustang
```

5. Public, Private, Protected

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors. The four access levels are −

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Default Access Modifier - No Keyword

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example

Variables and methods can be declared without any modifiers, as in the following examples −

```
String version = "1.5.1";

boolean processOrder() {
   return true;
}
```

Private Access Modifier - Private

Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.

Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

Example

The following class uses private access control −

```
public class Logger {
   private String format;

   public String getFormat() {
      return this.format;
   }

   public void setFormat(String format) {
      this.format = format;
   }
}
```

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So, to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and *setFormat(String)*, which sets its value.

Public Access Modifier - Public

A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example

The following function uses public access control −

```
public static void main(String[] arguments) {
   // ...
}
```

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Protected Access Modifier - Protected

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example

The following parent class uses protected access control, to allow its child class override *openSpeaker()* method −

```
class AudioPlayer {
  protected boolean openSpeaker(Speaker sp) {
    // implementation details
  }
}

class StreamingAudioPlayer extends AudioPlayer {
  boolean openSpeaker(Speaker sp) {
    // implementation details
  }
}
```

6 POLYMORPHISM

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

For example, think of a superclass called Animal that has a method called animalSound(). Subclasses of Animals could be Pigs, Cats, Dogs, Birds - And they also have their own implementation of an animal sound (the pig oinks, and the cat meows, etc.):

7.ENCAPSULATION

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare class variables/attributes as private
- provide public **get** and **set** methods to access and update the value of a private variable

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the `import` keyword:

Syntax

```
import package.name.Class;   // Import a single class

import package.name.*;   // Import the whole package
```

8.CASTING NUMERICAL VALUES

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

- Widening Casting **(automatically) - converting a smaller type to a larger type size**
  byte -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**

- Narrowing Casting **(manually) - converting a larger type to a smaller size type**
  double -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**

9.UPCASTING AND DOWNCASTING

Generally, upcasting is not necessary. However, we need upcasting when we want to write general code that deals with only the supertype. Consider the following class:

```
public class AnimalTrainer {
   public void teach(Animal anim) {
      anim.move();
      anim.eat();
   }
}
```

Downcasting is casting to a subtype, downward to the inheritance tree. Let’s see an example:

```
Animal anim = new Cat();
Cat cat = (Cat) anim;
```

10.USING GENERICS

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods −

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).