NIIT

Java Object-Oriented Programming

Objectives

- By the end of this session, you should be able to have a better understanding of:
  - OOP concepts
  - Class
  - Interface
  - Overloading, overriding, and hiding
  - Final variable, method, and class
  - Polymorphism in action

Object-Oriented Programming

- OOP was designed for better control over concurrent modifications of the shared data
- The idea behind OOP was to restrict the direct access to data and allow it only through a dedicated layer of code
- Since the data needs to be passed around and modified in the process, the concept of **Object** was thought of
- Therefore, OOP is a way of programming solutions in terms of objects, rather than mere data

OOP concepts

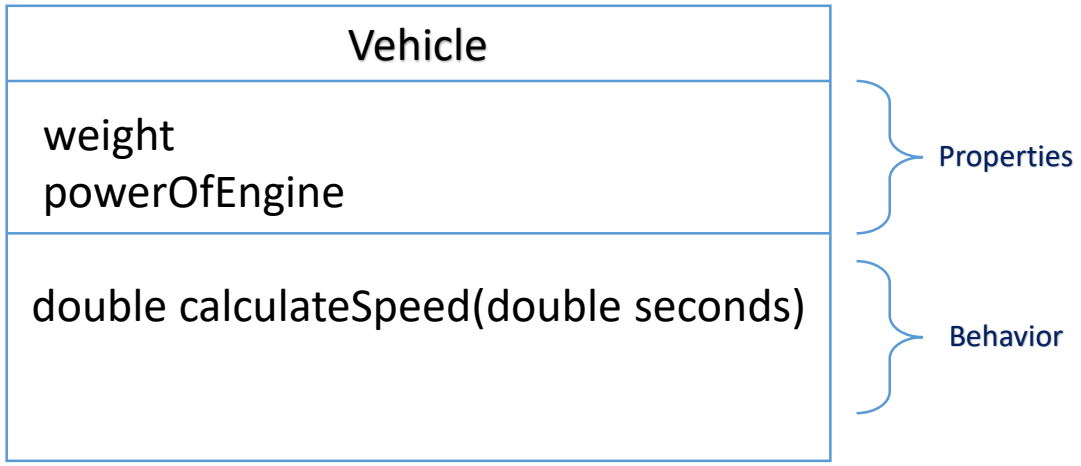
- Object/ Class
- Inheritance
- Abstraction/ interface
- Encapsulation
- Polymorphism

Object/ Class

- In the most general terms, ***an object is a set of data*** that can be passed around and can be accessed only through a *set of methods* passed along with it
- This ***data*** defines an ***object's state***, while the ***methods*** constitute the ***object's behavior***
- Each object is fabricated based on a template called a ***class***
- A class defines:
  - *Object state* in the form of *properties or fields*, and
  - *Object behavior* in the form of *methods*
- A ***method*** is a group of statements that performs some action and may or may not return a result

Object/ Class: Example

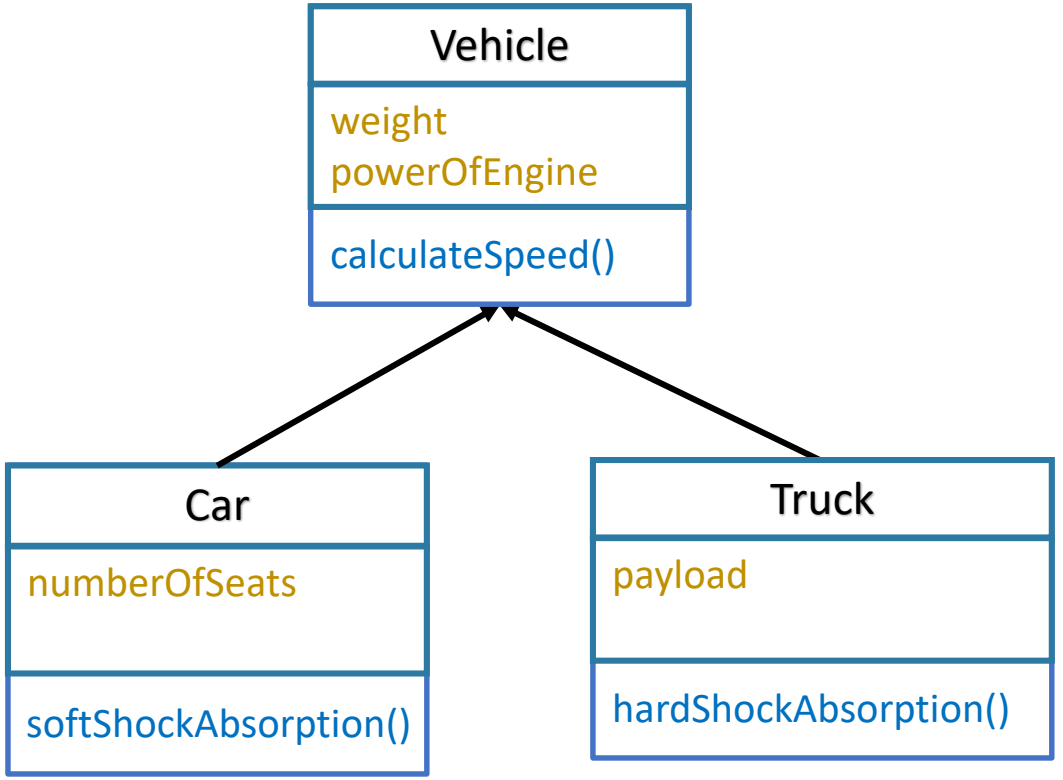
- Let's say, a class **Vehicle,** defines the properties & behavior of a vehicle in principle
- For simplicity, let's assume that a vehicle has only two properties (as mentioned in the diagram)
- It also has a behavior: it can attain a certain speed in a certain amount of time (depending on its weight and the power of it's engine)
- The behavior is expressed as a method that:
  - Takes the amount of time as input
  - Calculates the speed that can be attained by the vehicle, and
  - Returns the calculated speed as a result



Note: *Every object of the Vehicle class will now have this specified state and behavior*

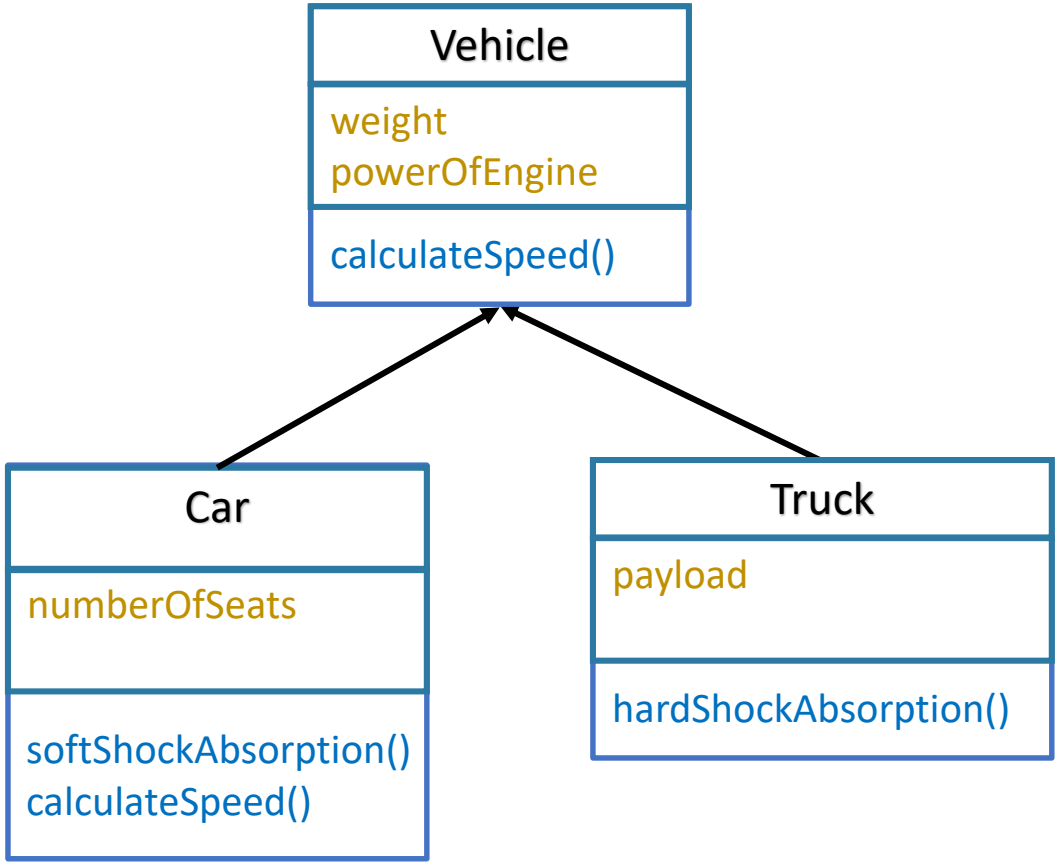
Inheritance

- Objects can establish a ***parent-child*** *relationship*
- This way, the properties, and behavior can be propagated and shared by the connected classes
- In addition, the child class, can also have its own specific properties and behavior
- Each time a child object is created, a new parent object is created first, therefore, the child objects ***can have the same behavior*** *but* ***they exist in different states***



Inheritance (ctd)

- It is possible to make a child behave differently than the inherited behavior would do
- The method that captures the behavior can be re-implemented in the child class
- This means, **the child can *override* the inherited behavior**
- So, the **Car** class can implement its own way of calculating the speed
- Although the ***properties can be inherited*** from the parent class ***they cannot be overridden***



Inheritance (ctd)

- The parent-child relationship in Java is expressed using the ***extends*** keyword
- There is no limit on how long the chain of inheritance can be
- The parent class is called the ***'super class'***, whereas the child class is called the ***'sub class'***
- In the example code, classes **A, B, C,** and **D** have the following relationships:
  - Class D inherits from classes **C, B,** and **A**
  - Class C inherits from classes **B** and **A**
  - Class B inherits from class **A**

```
1  class A {
2      //properties
3      //behavior
4  }
5  class B extends A {
6      //properties
7      //behavior and/or overridden behavior
8  }
9  class C extends B {
10     //properties
11     //behavior and/or overridden behavior
12 }
13 class D extends C {
14     //properties
15     //behavior and/or overridden behavior
16 }
```

Abstraction/ Interface

- Each object has a certain interface, a formal definition of the way other objects can interact with it
- It describes how the object data and behavior can be accessed
- It isolates (abstracts) an object's appearance from its implementations (behavior)
- The ***method signature*** along with a *return type* is presented as an ***interface***
- It does not say anything about the code that does the calculations – only about the method name, parameters' types, their sequence, and the result type
- A class can implement many different interfaces
- Two different classes (and their objects) can behave differently even when they implement the same interface

Abstraction/ Interface (ctd)

- Similar to classes, interfaces can also have a parent-child relationship using the ***extends*** keyword
- Abstraction/ interface also reduces dependency between different sections of the code
- Each class can be changed without the need to coordinate it with its clients, as long as the interface remains the same

```
1 interface A {
2     //behavior
3 }
4 interface B extends A {
5     //behavior and/or overridden behavior
6 }
7 interface C extends B {
8     //behavior and/or overridden behavior
9 }
10 interface D extends C {
11     //behavior and/or overridden behavior
12 }
```

Encapsulation

- Encapsulation simply means, bundling the publicly accessible methods and privately accessible data together
- It controls access to the object properties
- The object state (values of properties) is the data that is encapsulated
- So, encapsulation encourages better management of concurrent access to the shared data
- In the example code, the value of ***'property'***, is not accessible directly because it is defined as ***'private'***
- Its value can only be read or modified using the ***'public'*** methods getProperty*()* and *setProperty(String value)*

```
1 class Example {
2   private String property = "initial value";
3
4   public void setProperty(String value) {
5     property = value;
6   }
7
8   public String getProperty() {
9     return property;
10  }
11 }
```

Polymorphism

- It allows an object to assume an appearance of implemented interfaces and behave as any of its ancestor classes
- Polymorphism won't be possible without inheritance, interface, and encapsulation, because:
  - **Inheritance** allows an object to acquire and/or override the behaviors of all its ancestors,
  - **An Interface** hides the name of the class that implemented it, and
  - **Encapsulation** prevents exposing the object state

Class

- Class is the fundamental program unit in Java
- It comprises methods, which in turn, contain executable statements
- One or more classes are stored in **‘.java’** files
- They are compiled by the Java compiler **‘javac’** and stored in **‘.class’** files
- Each **‘.class’** file contains only one compiled class and can be executed by JVM
- The JVM, when activated, loads the **‘main class’** into the memory, finds the **main()** method, and starts executing it
- The **main() method** has a particular declaration to be adhered to:

public static void main(String[] args)

- **‘args’** here is an array of String type, that is used to store **command line arguments** if any

Example of a main class

- Here is an example of a **'main class'**
- A class that contains the **'main()'** method
- It represents a very simple application that
  - Receives any number of parameters as command line arguments, and
  - Passes them one by one to the **'display()'** method of **'AnotherClass'**

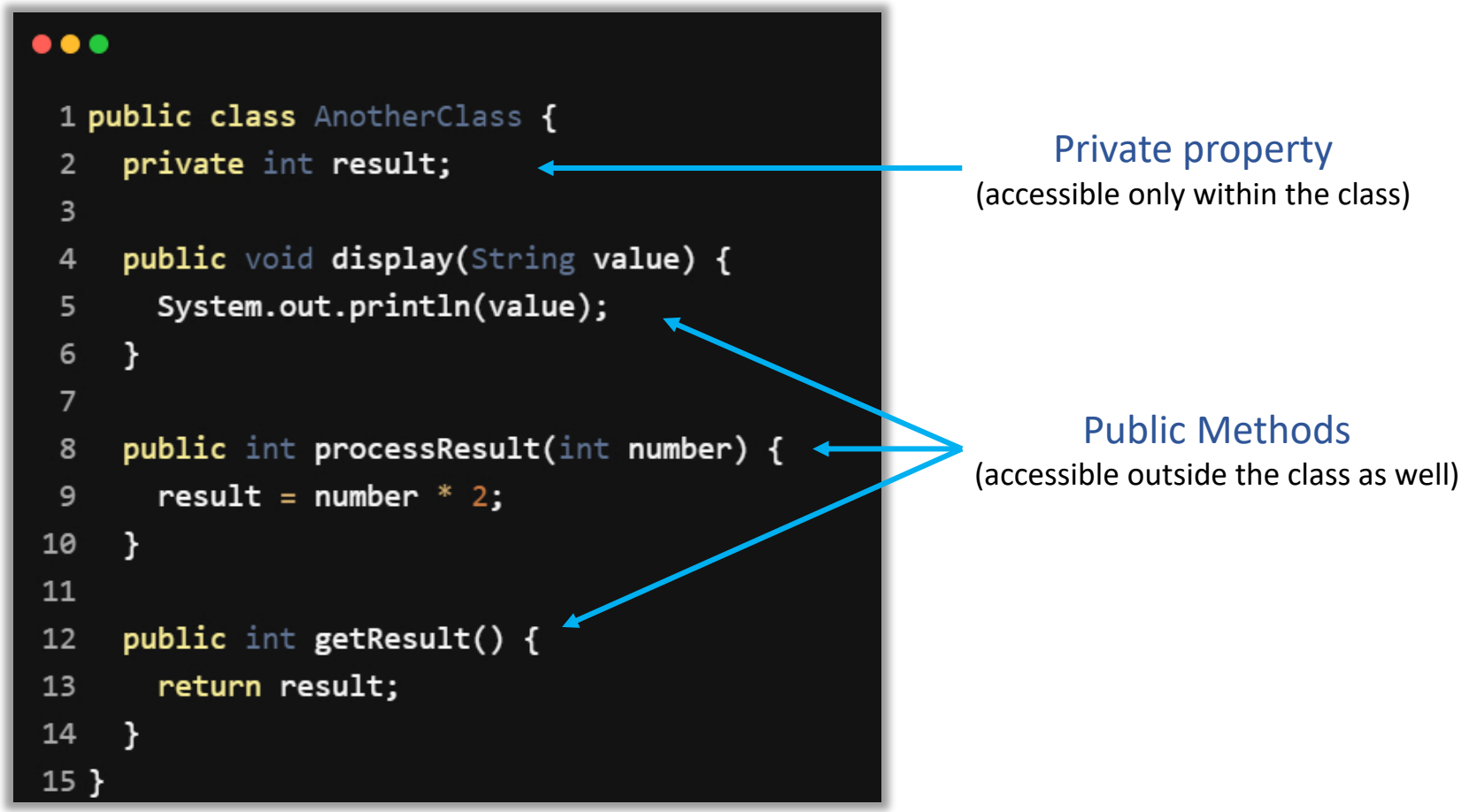
```
1 public class MyApplication {
2   public static void main(String[] args) {
3     AnotherClass anotherObject = new AnotherClass();
4     for(String arg : args) {
5       anotherObject.display(arg);
6     }
7   }
8 }
```

*This is how we create an object of **AnotherClass***

Example of Another class

- **‘AnotherClass’** is defined as shown in the code snippet

```
1  public class AnotherClass {
2    private int result;
3
4    public void display(String value) {
5      System.out.println(value);
6    }
7
8    public int processResult(int number) {
9      result = number * 2;
10   }
11
12   public int getResult() {
13     return result;
14   }
15 }
```



Method

- Java statements are organized as methods
- The general syntax of a method is as follows:

```
<return_type> <method_name>(<list_of_parameter_types>) {
  //method body, that is a sequence of statements
}
```

- The method name along with the parameters list is called the ***method signature***
- The number of input parameters is called an **arity**

Method (ctd)

- Two methods are said to have the same signature, if
  - They have the same name,
  - The same arity, and
  - The same sequence of types in the parameters list
- Code inside methods may be different even if their signature is the same

```
1 double doSomething(String word, int number) {
2   //some code
3 }
4
5 double doSomething(String sentence, int newNumber) {
6   //another code
7 }
```

Method (ctd)

- The following two methods have different signatures

```
1 double doSomething(String word, int number) {
2   //some code
3 }
4
5 double doSomething(int newNumber, String sentence) {
6   //another code
7 }
```

- Just a change in the sequence of parameters makes the signature different, even if the method name remains the same

Varargs

- ‘Varargs’ stands for **variable arguments** and is a different type of parameter
- It is declared as a type followed by three dots (ellipses)

```
1 String someMethod(String word, int number, double...values) {
2   //method body
3 }
```

Varargs

- When ‘someMethod’ is called, the arguments are matched from left to right
- Once the ‘varargs’ parameter is encountered,
  - An ***array of the remaining arguments is created,***
  - Named as **values,** and
  - Passed into the method

Varargs: Demo

```java
public static void main(String... args){
  someMethod("str", 42, 10, 17.23, 4);
}

private static String someMethod(String word, int number, double...values) {
  System.out.println(values[0] + " " + values[1] + " " + values[2]);
  return word;
}

//prints: 10.0, 17.23, 4.0
```

- ‘Varargs’ acts like an array of a specific type
- It can be listed as the last or the only parameter of a method

Constructor

- The purpose of a constructor is to initialize the object state to assign values to all the declared properties
- ***It is called only when a new instance of the class is created***
- If there is no constructor declared in the class
  - The Java compiler creates a default constructor without any parameters, and
  - JVM assigns default values to all the properties
- It is possible to declare any number of constructors explicitly, each taking a set of parameters to set the initial state of an object

```java
class Car {
  private String modelName;
  private String color;

  public Car(String modelName){
    this.modelName = modelName;
  }

  public Car(String color) {
    this.color = color;
  }

  public Car(String modelName, String color) {
    this.modelName = modelName;
    this.color = color;
  }
  //rest of the code
}
```

Constructor during Inheritance

- When inheritance is implemented, the parent object is created first
- Therefore, the parent class constructor is always called first
- If the parent object requires setting non-default initial values, then its constructor must be called explicitly by the child class constructor
- For this we use the ***super*** keyword

```
1 class Person {
2   private String name;
3
4   public Person(String name) {
5     this.name = name;
6   }
7   //rest of the code
8 }
9
10 class Student extends Person {
11   private String name;
12   private int studentId;
13
14   public Student(int studentId) {
15     super("");
16     this.studentId= studentId;
17   }
18
19   public Student(String name, int studentId) {
20     super(name);
21     this.studentId = studentId;
22   }
23   //rest of the code
24 }
```

Constructor during Inheritance (ctd)

- Every child class constructor tries to call the default constructor of its parent class
- But, *as soon as an explicit constructor is created, the default constructor is not provided by the compiler*
- Therefore, in our example, skipping ‘line number 17’ would result in an error
- To avoid this error, we can
  - Either add a no-args constructor in the **Person** class, or
  - Let line number 17 exist

(No-args constructor has been added in the example)

```
1  class Person {
2    private String name;
3
4    public Person() {}
5
6    public Person(String name) {
7      this.name = name;
8    }
9    //rest of the code
10 }
11
12 class Student extends Person {
13   private String name;
14   private int studentId;
15
16   public Student(int studentId) {
17     //super("");
18     this.studentId= studentId;
19   }
20
21   public Student(String name, int studentId) {
22     super(name);
23     this.studentId = studentId;
24   }
25   //rest of the code
26 }
```

The ‘new’ operator

- It is used to create an object of a class
- It does two things:
  - Allocates memory for the properties of the class, and
  - Return a reference to that memory
- We can assign this memory reference to a variable of the class type or the class’s parent type

```
1 Student student = new Student("John", 101);
2 Person person = new Student("Mary", 110);
```

The ‘new’ operator (ctd)

- Suppose, both our classes have methods as shown in the code (on the right)
- We could use any of the reference variables created in the previous slide to call these methods (as shown below)

```java
Student student = new Student("John", 101);
Person person = new Student("Mary", 110);

student.study();
student.sleep();

person.sleep();
(Student) person.study();
```

- Note that to access the child’s method using the parent class reference, we had to cast it to the child type
- This is possible because we assigned the child object to the parent’s reference type (polymorphism)

```java
class Person {
  private String name;

  public Person(String name) {
    this.name = name;
  }
  public void sleep() {
    System.out.println("Person Sleeps.");
  }
}

class Student extends Person {
  private String name;
  private int studentId;

  public Student(String name, int studentId) {
    super(name);
    this.studentId = studentId;
  }

  public void study() {
    System.out.println("Student Studies.");
  }
}
```

Class java.lang.Object

- All classes in Java are children of the **'Object'** class by default, which is declared in the **'java.lang'** package of the standard JDK library
- It has ten methods that every class inherits (listed on the right)
- The first three methods are the most often used and overridden methods
- The **'toString()'** method is typically used to print the state of the object, its default implementation in JDK is shown in the code snippet below

```
1 public String toString()
2 public int hashCode()
3 public Boolean equals(Object obj)
4 public Class getClass()
5 protected Object clone()
6 public void notify()
7 public void notifyAll()
8 public void wait()
9 public void wait(long timeout)
10 public void wait(long timeout, int nanos)
```

```
1 public String toString() {
2   return getClass().getName() + "@" + Integer.toHexString(hashCode());
3 }
```

Class java.lang.Object (ctd)

- The toString() method does not need to be called explicitly, it gets called whenever we try to print a reference variable
- Let's look at the output that we get when we try to print the reference of our **Student** class, which we created earlier
- Such an output is not user-friendly at all, so it is desirable to override the toString() method
- Using IntelliJ IDEA:
  - Right-click inside the Student class code and select **'Generate…'**, and click on **toString()**
  - Select the fields to be printed, and click on OK
- The same code now results in a new output, which is more readable and understandable

```java
public class Main {
    public static void main(String[] args) {
        Student student = new Student("John", 101);
        System.out.println(student);
    }
}
```

```
com.niit.jdp.Student@6d03e736
```

```java
@Override
public String toString() {
    return "Student{" +
            "id=" + id +
            ", name='" + name + '\'' +
            '}';
}
```

```
Student{id=101, name='null'}
```

Instance and static properties and methods

- The methods we've seen so far, were **invoked only on an object** of the class
- Such methods are called **instance methods**, and they typically use the object properties or state
- If they do not use the object properties or state, they can be made **static** and invoked **without creating an object** of the class
- **'main()'** is an example of a static method
- An example of programmer created static method is shown in the code snippet on the right, along with the way of invoking it
- Static methods can be called on objects too, but it is considered as a bad practice

```
1 class Example {
2   public static void exampleMethod() {
3     //some code
4   }
5 }
```

```
1 Example.exampleMethod();
```

Instance and static properties and methods (ctd)

- Similar to a method, a property can also be declared as static, and thus, accessed without creating an object
- Let's add a static property to our **Example** class
- This property can be accessed directly via class name too
- A static property exists as a single copy in the JVM memory and its value can be shared by all the methods
- This goes against the concept of state encapsulation and ***may cause consistency problems***

```
1 class Example {
2   public static String exampleProperty = "example";
3
4   public static void exampleMethod() {
5     //some code
6   }
7 }
```

```
1 System.out.println(Example.exampleProperty);
2
3 //prints: example
```

Instance and static properties and methods (ctd)

- Therefore, a static property is typically used for two purposes:
  - To store a constant – a value that can be read but not modified
  - To store a stateless object that keeps read-only values
- A typical example of a constant could be the name of a resource

```
1 class Example {
2   public static final String INPUT_FILE_NAME = "example.csv";
3 }
```

- The **‘final’** keyword tells the compiler and the JVM that the value, once assigned, to this property, cannot change later

Interface

- An interface hides the implementation and exposes only method signatures with return types
- Also the fields inside an interface are implicitly **public, static,** and **final**
- An example interface, and a class implementing it, are shown in the code snippets (on the right)
- Note that the methods declared inside the interface are abstract because they do not have a definition
- *The class implementing the interface must override all such methods, or else the class must be declared as abstract*
- An interface ***cannot be instantiated***, but its reference can be created
- For example, `SampleInterface sampleReference = new SampleClass();`
- With Java 8, the interface acquired the ability to have not just abstract methods but **'default', 'private',** and **'static'** ones too

```java
interface SampleInterface {
  void sampleMethodOne();
  String sampleMethodTwo(int number);
}
```

```java
class SampleClass implements SampleInterface {
  @Override
  public void sampleMethodOne() {
    //method body
  }

  @Override
  public String sampleMethodTwo(int number) {
    //method body
    return "sample-word";
  }
}
```

Default methods

- Our **SampleInterface** now has a **default** method with a definition
- And our **SampleClass** does not override it, but it is still available to be called by the object of SampleClass

```
SampleClass sampleReference = new SampleClass();
sampleReference.sampleMethodOne();
sampleReference.sampleMethodTwo(44);     //returns sample-word
sampleReference.sampleMethodThree();     //returns 42
```

- So, a default method may or may not be overridden by the implementing class
- If overridden, the interface implementation is ignored
- ***The purpose of the default method in an interface is to provide a new method to the classes implementing the interface, without changing them***

```
1  interface SampleInterface {
2    void sampleMethodOne();
3    String sampleMethodTwo(int number);
4    default int sampleMethodThree() {
5      return 42;
6    }
7  }
8  class SampleClass implements SampleInterface {
9    @Override
10   public void sampleMethodOne() {
11     //method body
12   }
13   @Override
14   public String sampleMethodTwo(int number) {
15     //method body
16     return "sample-word";
17   }
18 }
```

Private methods

- If there are several default methods in an interface, it is possible to create private methods accessible only by the default methods
- These methods cannot be accessed from outside the interface

```
1 interface SampleInterface {
2   void sampleMethodOne();
3   String sampleMethodTwo(int number);
4
5   default int sampleMethodThree() {
6     return getNumber();
7   }
8
9   default int sampleMethodFour() {
10    return getNumber() + 22;
11  }
12
13  private int getNumber() {
14    return 42;
15  }
16 }
```

Note: *all non-private methods inside an interface are* ***public*** *by default*

Interface Vs. Abstract class

- A class, too, can be declared as **'abstract'**
- Also note that a class can extend only one class (abstract or non-abstract) but it can implement multiple interfaces at the same time

Similarities

- Similar to an interface, an abstract class cannot be instantiated
- An abstract class forces every child class to implement its abstract methods, just like an interface does

Principle differences

- An abstract class can have a constructor but an interface cannot
- An abstract class can have a state while an interface cannot
- The fields of an abstract class may be public, private, protected, static, and/or final, while, in an interface, fields are always public, static, and final
- The methods in an abstract class can be public, private, or protected, whereas the interface methods can be public or private only

Overloading

```java
interface SampleInterface {
  int sampleMethod();
  int sampleMethod(String message, double value);
  default int sampleMethod(String message, int number) {
    return 1;
  }
  static int sampleMethod(String message, int number, double value) {
    return 1;
  }
}
//none of the methods have the same signature
```

```java
class SampleClass {
  int method(String message) {
    return 42;
  }
  int method(String message, double value) {
    return 0;
  }
  static int method(String message, double value, int number) {
    return 1;
  }
}
```

- Overloading means creating several methods with the same name and different parameters (i.e., different signatures) ***in the same class or interface***
- Neither the return type of the method nor its access modifiers play any role in method overloading

```java
interface One {
  int method(String message);
  int method(String message, double value);
}

interface Two extends One {
  default int method(String message, int number) {
    return 1;
  }
  static int method(String message, int number, double value) {
    return 1;
  }
}
//it does not matter where the methods with the same name are declared
//the effect is same, even during inheritance between interfaces or classes
```

Overriding

- Overriding happens **only with non-static methods**
- The method signatures **must be exactly the same,** and **belong to different classes** or **interfaces** related via **inheritance**
- The overriding method resides in the child interface or class
- **A private method cannot be overridden**
- The **'@Override'** annotation tells the compiler that the method overrides a method of one of its ancestors
- This way the compiler makes sure that there are no mistakes made (spelling mistakes, for example) while overriding the method
- Given the example code, if we call **method()** using the instance of the class, we will get the result as shown
- ***Although the example cites interfaces, the same rules apply for classes as well***

```
1 interface One {
2   default void method() {
3     System.out.println("interface One");
4   }
5 }
6 interface Two extends One {
7   @Override
8   default void method() {
9     System.out.println("interface Two");
10  }
11 }
12 class Example implements Two {}
```

```
1 Example example = new Example();
2 example.method();   //prints: interface Two
```

Hiding

- The term ‘hiding’ came from the behavior of static properties and methods of classes and interfaces
- Each static property or method exists as a single copy and is loaded into the memory only once, i.e., along with the associated class or interface
- They are not associated with an object
- Hence, the similarly named static property or method of a child **does not** override the static property or method of a parent
- If the parent and child class (or interface) contain a static method having the same signature, ***the parent method is said to be hidden by the child method***
- Same is the case with a static property

```
1 class SampleClass {
2   public static String name = "sample name";
3   public String newName = "New sample name";
4   public static void method() {
5     System.out.println("Sample Class");
6   }
7 }
8 class AnotherClass extends SampleClass {
9   public static String name = "Another name";
10  public String newName = "Another new name";
11  public static void method() {
12    System.out.println("Another class");
13  }
14 }
```

```
1 System.out.println(AnotherClass.name);      //prints: Another name
2 AnotherClass.method();                      // prints: Another Class
```

Final variable, method, and classes

- **‘final’** keyword placed in front of a variable declaration makes the variable immutable after initialization
- In the case of an object property, the property can be initialized in the constructor too

```
1 final String language = "English"; //declaration and initialization
2
3 //the initialization can also be delayed
4 final String language;
5 language = "English";
```

```
1 class Example {
2   private final String string1 = "word";
3   private final String string2;
4   private final int number; //error
5
6   public Example {
7     this.string1 = "hello"; //error
8     this.string2 = "hello world";
9   }
10 }
```

Final variable, method, and classes

- It is possible to initialize a final property in an initialization block also
- In the case of a static property, it has to be initialized either during its declaration or in a static block

```
1 class Sample {
2   private final static String string1 = "hello";
3   private final String static string2;
4   static {
5     string2 = "world";
6   }
7 }
```

```
1 class Sample {
2   private final String string1 = "hello";
3   private final String string2;
4   {
5     this.string2 = "world";
6   }
7 }
```

- A **method** declared as **final** cannot be overridden in a child class
- A **final class** cannot be inherited, which makes all the methods of such a class **effectively final**

Polymorphism in action

- Polymorphism in Java is the ability of an object to behave as if going through a ***metamorphosis***
- *Metamorphosis* is *"a change of the form or nature of a thing or person into a completely different one, by natural or supernatural means"*
- Java objects exhibit completely different behaviors under different conditions
- This concept will be discussed using an **object factory** – a specific programming implementation of a method that returns objects of varying type or class

Object factory

- Both **CalculationAlgo1** & **CalculationAlgo2** classes implement the same interface **Calculation** but use different algorithms

```
1 interface Calculation {
2   double calculate();
3 }
```

```
1 class CalculationAlgo1 implements Calculation {
2   public double calculate() {
3     return 42.1;
4   }
5 }
```

```
1 class CalculationAlgo2 implements Calculation {
2   private int value1;
3   private double value2;
4   public CalculationAlgo2(int value1, double value2) {
5     this.value1 = value1;
6     this.vaue2 = value2;
7   }
8   public double calculate() {
9     return value1 * value2;
10  }
11 }
```

Object factory (ctd)

- Suppose we want the selection of the algorithm to be made in a property file, then we can create an object factory as shown in the code snippet
- The factory selects which algorithm to use based on the value returned by the **getAlgoValueFromPropertyFile()** method
- In the case of the second algorithm, it also gets the input parameters from the property file
- But the complexity is hidden from the client, as visible in the code below

```
Calculation calculation = CalculationFactory.getCalculator();
double result = calculation.calculate();
```

```
class CalculationFactory {
  public static Calculation getCalculator() {
    String algorithm = getAlgoValueFromPropertyFile();
    switch(algorithm) {
      case "1" :
        return new CalculationAlgo1();
      case "2" :
        int value1 = getValue1FromPropertyFile();
        double value2 = getValue2FromPropertyFile();
        return new CalculationAlgo2(value1, value2);
      default :
        System.out.println("unknow value " + algorithm);
        return new CalculationAlgo1();
    }
  }
}
```

Summary

- In this session, you learned about:
  - The basic concepts of OOP and how they are implemented in Java
  - The Java language constructs of **class** and **interface** in detail
  - Concept and use of Overloading, overriding and hiding
  - Use of **final** keyword
  - Polymorphism and object factory
- In the next chapter, you will become familiar with Java language syntax, including packages, importing, access modifiers, reserved and restricted keywords, and some aspects of Java reference types