1DV532 – Starting Out with Java

# Lesson 7

# Classes and Objects

Dr. Nadeem Abbas

nadeem.abbas@lnu.se

# Classes and Objects

- The concept of *Classes* and *Objects* is central to Object-Oriented Programming
- An object-oriented program is essentially a set of interacting objects
- **Objects** are data abstractions with an interface of named **operations** and a hidden local **state**, and have an associated **type (class)**
  - Each object is an instance of a certain **class**
- We learned basics about Classes and Objects in Lesson 6;
  - This lesson focus on realizing the concept of classes and objects in Java

# Classes in Java

**Class Definition – General Form:**

```
class ClassName{ // start of a class body
        // Data Members or Fields
        // Member Methods
 } // end of a class body
```

- As shown above, a class definition in Java begins with a keyword *class* followed by an identifier that specifies *ClassName*
- A class definition mainly defines two things:
  - **Data Members,**
  - **Member methods**
- Both data members and member methods are defined within a **class body** which is marked by curly braces {}

# Java Class Definition – General Form

```
[access modifier] class ClassName {
    [access modifier] [static] type variable1;
    [access modifier] [static] type variable2;
    ...
    [access modifier] [static] type variableN;

    //constructors
    [access modifier] ClassName (parameter-list) {
                        //body of constructor
    }

    [access modifier] [static] type methodname1(parameter-list) {
    // body of method
    }
    [access modifier] [static] type methodname2(parameter-list) {
    // body of method
    }
    // ...
    [access modifier] [static] type methodnameN(parameter-list) {
    // body of method
    }
}
```

Data Members

Member Methods

# Java Class Definition – Example

```
[access modifier] class ClassName {
[access modifier] type instance-variable1;
[access modifier] type instance-variable2;
…
[access modifier] type instance-variableN;

[access modifier] type
      methodname1(parameter-list) {
      // body of method
}
[access modifier] type
      methodname2(parameter-list) {
// body of method
}
// …
[access modifier] type
      methodnameN(parameter-list) {
// body of method
}
}
```

```
public class Person {
      /* Data Members or fields */
      private String name;
      private int height;
      private int weight;
      private static int personCounter;

      /* Constructors */
      public Person(String n, int h, int w) {name = n;
      height = h;weight = w; personCounter++;}
      public Person() {name = ""; height = 0;weight = 0;
      personCounter++;}

      /* Member Methods */
      public void setName(String n) {name = n;}
      public void setHeight(int h) { height = h;}
      public void setWeight(int w) { weight = w;}
      public static int getPersonCounter() {return
      Person.personCounter;}

      public void printPerson() {
      System.out.println("Name: " + name + ", Height: " +
      height + ", Weight: " + weight);
      } }
```

# Data Members

- Data members are variables defined within a class body to specify **states** or attributes that objects of a class will have.
  - Data members are often referred as ***fields*** or ***instance variables***

- Following is a general form of how a data member is defined.

  `[access modifier] [static]` **`type variable1;`**

  - Here, **`type`** refers to a data type and **`variable1`** is the identifier used to name a data member.
    - The Person class shown on previous slide, for example, defines four data members:
      - `private String name;`
      - `private int height;`
      - `private int weight;`
      - `private static int` *`personCounter;`*

  - `Access modifier` and `static` will be explained later.

# Member Methods

- Member methods, or simply **methods** are parts of a class definition that are used to define **behavior or operations** associated with objects of a class.
  - All the operations that objects of a class can perform are specified as methods
- Following is a general form of a Method definition:

```
[access modifier] [static] type methodname1(parameter-list) {
    // body of method
}
```

Here,

- **`type`** often called as return type is a data type of the value returned by a method as a result of its operations. The type is specified as **`void`** if the method does not return a value.
- **`methodname1`** represents an identifier used to name a method.
- **`parameter-list`** represents a comma separated list of input parameters, preceded by their data types.
  - The input parameters are used to pass data to a method.
  - Empty parentheses are used if there is no data passed to a method.

# Member Methods – Example 1

```
int sum(int a, int b){
   return a + b;
}
```

- The above example code defines a method named `sum`.
- The `parameter-list` specifies two input parameters of type int, a and b. This means to use this method, we need to pass two integer type values.
- The operation performed by the method is that it adds values passed to the two input parameters, a and b, and return result of the addition.
- As the value returned by the method is of type integer thus its return type is specified as `int`

# Member Methods – Example 2

```
public void setName(String n) {name = n;}
```

- The above example code shows a method named `setName` from the example class Person shown earlier in this lesson.
- The `parameter-list` specifies one input parameters of type String n. This means that the method accepts one String type argument (input parameter) which is used to set value of the member data called *name*.
- The operation performed by the method is that it adds values passed to the two input parameters, a and b, and return result of the addition.
- The setName method has return type void, i.e., it does not return anything to the caller
- As the value returned by the method is of type integer thus its return type is specified as `int`

# Constructors / Creating Objects

- Constructors are a special kind of methods that are invoked when an object is created.
  - They are often used to initialize data members

- **General syntax**

```
[access modifier] <ClassName> ( <parameters> ) {
        <statements>
}
```

- As shown above, a constructor definition looks similar to a method definition except following:
  - **No return type, not even *void***
  - **Name must be the same as the class name**

- A class may have zero or more constructors.

# Constructor - Example

```
/* Constructors */
public Person(String n, int h, int w) {
    name = n;
    height = h;
    weight = w;
    personCounter++;
}
```

- The above code shows a Constructor from the class Person.
  - Note that the constructor has same name, *Person*, as that of the class it belong to.

# Creating Objects – *new* Operator

Objects of a class are created using ***new*** keyword, often called as *new operator* as shown in example below:

```
– Person trump = new Person("Trump", 170, 90);
  Person putin = new Person();
```

Each of the above statements has three parts:

1. **Declaration**: The code shown as bold are all object declarations that associate object name with an object type.
2. **Instantiation**: The *new* keyword is a Java operator that creates the object and and returns a reference to the created object.
3. **Initialization:** The new operator is followed by a call to a constructor, which initializes the new object.

# Default Constructor

- If you do not define any constructors in your class, Java will automatically create a ***default*** **or** ***no-argument*** constructor.
  - The default constructor takes no arguments, thus also reffered as empty or no-argument constructor.

- If you include even one constructor in your class, Java will not provide this default constructor
  - In this case you have to define your own default or no-argument constructor.

# A Class Is a Type

- Classes in java work like *data types* such a*s int, float, double, etc.,* that can be used to declare and instantiate variables called *objects*.

- A class is a special kind of programmer-defined type, and we can declare data members or other variables of a class type

- A variable of a class type is called an *object* or *an instance of the class*

- Class of an object determines the types of data that an object can contain, as well as the actions it can perform

# Primitive Type Values vs. Class Type Values

- A primitive type value is a single piece of data, whereas a class type value or **object** can have multiple pieces of data (data members), as well as actions called *methods*
  - All objects of a class have same methods
  - All objects of a class have the same data members, e.g. all objects of the example Person class have four data members: *name, height, weight,* and *personCounter.*
  - For a given object, each date member can hold a different value
  - All objects have their own copy of *non-static data members or instance variables*. On the hand *static or class data members* are shared by all objects of a class.
    - *static and non-static* will be discussed later in the lesson.

# Using Classes and Objects –
# Accessing Data Members and Member Methods

- Once you have defined a *class* you would like to use it for some tasks
- Classes are often used by creating objects.
  - For example, if we want to use the class Person , we need to create its object as follows.
    - **Person putin** = new Person();

- The above statement will create an object of type *Person* and return its reference to the the object reference variable *putin* of type Person.
- As there can be more than one persons involved in a task, so we may create as many as required objects of the Person class, for examples.
    - **Person x** = new Person();
    - **Person y** = new Person();
    - …
    - And so on.

# Using Classes and Objects –
# Accessing Data Members and Member Methods

- A class's members (both data members and methods) are accessed by their names
  - Thus, members should be named *unambiguously*, i.e., should have different names or method signatures.

- The above member access method works only within a class to which the members belong to.

- To access a member outside its class, we need an **object reference**, followed by the **dot (.) operator**, followed by a data member or method name, as in:
  - *objectReference**.fieldName***
    - For example, `putin.name = “Putin”;`
  - *objectReference**.methodName**()*;
    - For example, `putin.setName("Putin");`
- `E`xample program *PersonMain.java* demonstrates how we can create objects of a class and access its member methods.

# Using Classes and Objects – Accessing Data Members and Member Methods

- Data members of an object created using default or no-argument constructor are initialized to either *default values* or *values given by a programmer's defined no-argument constructor*.
  - **Person putin** = new Person();
- In the above line of code, a Person type object named *putin* is created using no-argument constructor. All data members of the *putin* object will have default values assigned by the constructor.
  - For such objects created using no-argument constructor, we can access their data members either directly or by calling some member method, as shown below:

```
putin.name = “Putin”; // direct access to data member
Putin.height = 180; // direct access to data member
putin.setName("Putin"); // access through a member method
```

- Access to the class members depends upon their access modifiers.

# Access/Visibility Modifiers

- Access to a class and its members (both data members and methods) is controlled through *access or visibility modifiers*.

- The access modifiers determine whether a class or class members (fields and methods) are visible and accessible to other classes or not.

- Java provides four different access modifiers as follows:
  1. **public**
  2. **protected**
  3. **default or package-private**
  4. **private**

- These modifiers are in-fact Java *keywords* that are used with class and class members definitions, as shown in the example code on next slide.

# Access/Visibility Modifiers - Example

```
[access modifier] class ClassName {
[access modifier] type instance-variable1;
[access modifier] type instance-variable2;
…
[access modifier] type instance-variableN;

[access modifier] type methodname1(parameter-
        list) {
        // body of method
}
[access modifier] type methodname2(parameter-
        list) {
// body of method
}
// …
[access modifier] type methodnameN(parameter-
        list) {
// body of method
}
}
```

```
public class Person {
        /* Data Members or fields */
        private String name;
        private int height;
        private int weight;
        private static int personCounter;

        /* Constructors */
        public Person(String n, int h, int w) {name = n; height =
        h;weight = w; personCounter++;}
        public Person() {name = ""; height = 0;weight = 0;
        personCounter++;}

        /* Member Methods */
        public void setName(String n) {name = n;}
        public void setHeight(int h) { height = h;}
        public void setWeight(int w) { weight = w;}
        public static int getPersonCounter() {return
        Person.personCounter;}

        public void printPerson() {
                System.out.println("Name: " + name + ", Height:
        " + height + ", Weight: " + weight);
        } }
```

# Access/Visibility Modifiers

1. **public:** Classes, methods, and data members defined using *public* keyword as an access modifier are visible and accessible to and from all classes
2. **protected:** Data members and methods defined using *protected* keyword as an access modifier are visible and accessible only to the class itself, its sub-classes and other classes defined in the same package.
3. **default or package-private:** Classes, methods, and data members defined without any access modifier have default or package-private access. Such Classes and class members are visible and accessible only to the class itself and to other classes defined in the same package.
4. **private:** Data members and methods defined using *private* keyword as an access modifier are visible and accessible only within the class in which they are declared and cannot be accessed from any other class

Note: protected and private access modifier are members only, and cannot be used with class declaration.

# Access Modifiers and Encapsulation

- Encapsulation is one of the four fundamental OOP concepts that we learned in Lesson 6.
- Java supports encapsulation using access modifiers as follows
  - Members that must not be viewable and accessable outside a class are declared as *private*
    - A rule of thumb is that all *data members* should be declared as private.
      - For example, all data members in the example class Person are declared as *private*
  - Members that need to be viewable and accessable outside a class are declared as *public*
    - A rule of thumb is that *member methods* should be declared as public
      - For example, all member methods including constructors in the example class Person are declared as *public*

# Getter Setter Methods

- Following the Encapsulation principle of OOP, data members are defined as *private*
  - No one from outside the class can access *private* members
- Access to the private data members of a class is provided through getter (accessor) and setter (mutators) methods that are declared as *public*
- **Getter method**, as the name indicates, allows you to get or obtain read only value of a data member
- **Setter Method**, as the name indicates, allows you to set or change value of a data member

# Getter Setter Methods - Examples

- Following are the examples of Getter and Setter methods from the example Class, Person.java

- **Getter Method**

```
public String getName() {
        return name;
}
```

  - This getter methods allows to get a value of a private data member called *name.*

- **Setter Method**

```
public void setName(String n) {
        name = n;
}
```

  - This setter methods allows to set a value of a private data member called *name.*

# Static Data Members

- If we create objects of a class, e.g., Person, each object will have its own distinct copies of data members so that objects cannot interfere with each other's data and each object may maintain its own data.

- Suppose that we want to keep planet data for all Person objects to record on which planet humans represented by Person object are living.

- Now assuming that all persons are living on the same planet Earth, we want to have a common data member that records same planet data for all objects of type Person.
  - This is accomplished in Java by declaring data members as ***static,*** syntax to declare a static data member is as follows:
    - `private static String planet;`
    - `private static int personCounter;`

- **`static`** is a Java keyword used to declare **static data members** that are common to all objects of a class.

# Static (Class) Vs. Non-Static (Instance) Members

- Static data members are common to all instances of a class and thus are also known as Class variables
- Non-static data members, declared without using static keyword, are separate for each instance of a class and thus are known as instance variables.

**Characteristics of Static Members**

1. Accessed using class reference

**General Form**

*ClassName.variable;*

*ClassName.method(arguments);*

**Examples**

```
Person.counter++;
Person.getPersonCounter();
```

2. Cannot directly access non-static members
3. Cannot use **this** or **super** *

**Characteristics of Non-Static Members**

1. Accessed using instance reference,

**General Form**

*instanceName.variable;*

*instanceName.method(arguments);*

**Examples**

```
Person trump=new Person();
trump.height=170;
trump.setName(“Trump”);
```

2. Can access static members
3. Can use **this** or **super**

* **this** and **super** both are Java keywords that will be explained in next steps of the course.

# Method Overloading

- **Method Overloading** is a feature that allows a class to have two or more methods with same names but different *signatures*
  - Method’s signature consists of the method’s name and parameter list
    - Access modifiers, static keyword, and return type of a method are not not part of the method’s signature.

- **Method Overloading Examples:**
  - `public static int `**`max(int num1, int num2)`**`{ //body }`
  - `public static double `**`max(double num1, double num2)`**`{//body}`
  - `public static long `**`max(long num1, long num2)`**` {//body}`

- Example program *MethodOverloadingDemo.java* demonstrates use of the method overloading.

# Constructor Overloading

- **Constructor Overloading** is similar to method overloading and allows a class to have two or more constructors with same names but different parameter lists.

  - Person.java has two overloaded constructors, both with same names but different parameter list.

    1. `public Person(String n, int h, int w){ //body }`
    2. `public Person() { //body }`

# Suggested Readings

- Absolute Java, Global Edition, 6/E by Walter J. Savitch, Chap 4, Defining Classes I, Chap 5, Defining Classes II
- Introduction to Java Programming, Brief Version, Global Edition, 11/E Liang, Chapter 9 “Objects and Classes”

- Java Tutorials
  - https://docs.oracle.com/javase/tutorial/java/concepts/index.html
  - https://docs.oracle.com/javase/tutorial/java/javaOO/index.html

Lnu.se