

CMPS 251



Read Chapter 3

Lecture 05

Basic Object-Oriented Programming in Java

Outline

- Classes and Objects
- Attributes with Getters and Setters
- Methods
- Constructors
- Access Modifiers
- Variable Types
- Enumeration

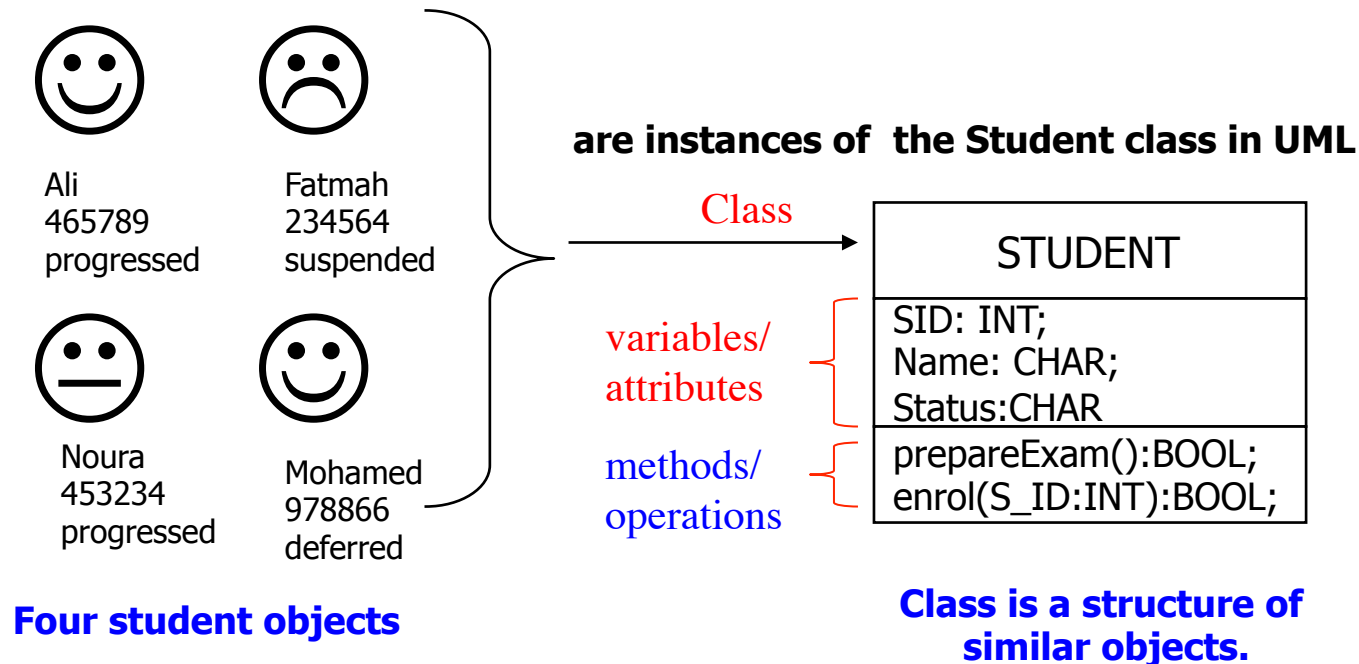
Classes and Objects

Class

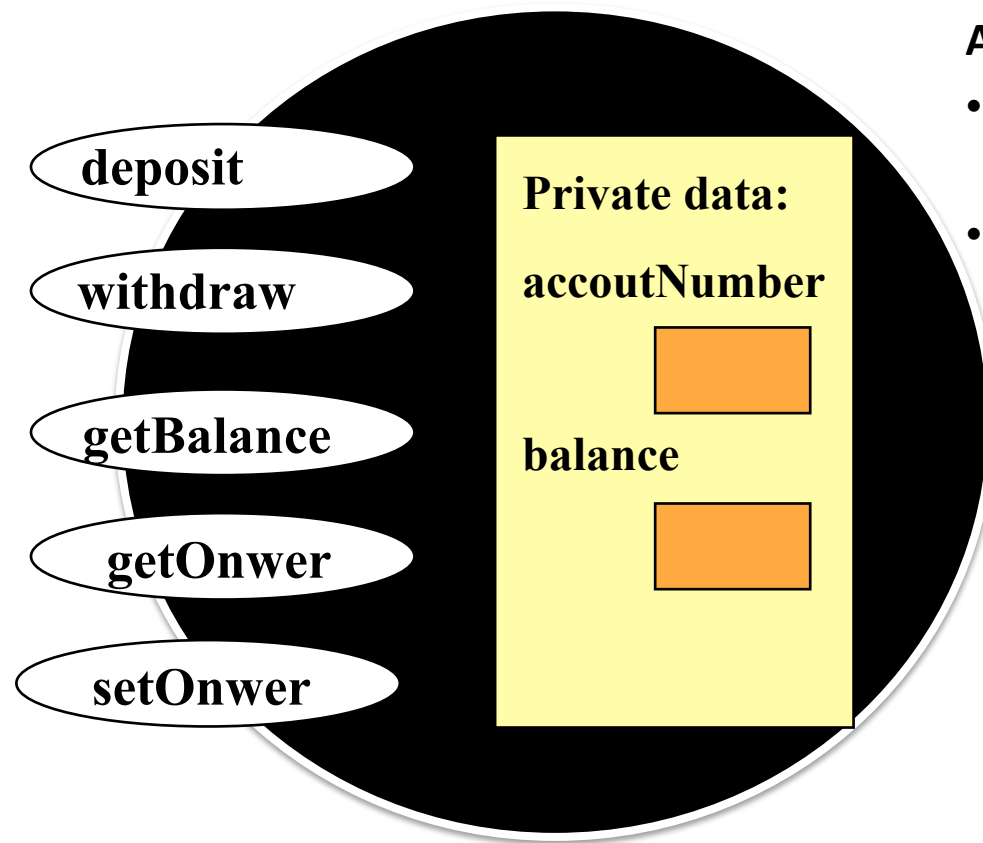
- Object-Oriented Programming (OOP) = Designing the application based on the objects discovered when analyzing the problem
- A *class* is code that describes a particular type of object. It specifies the data that an object can hold (**attributes**), and the actions that an object can perform (**methods**).
- A class is a **programmer-defined data type** and **objects are variables of that type**
 - A class contains **attributes** and **methods**
- A **Class** is a type, a structure of specific object types
- An object is a specific 'thing' of a class.
- **Example:**
 - Student (a type of persons who are students)
 - Fatima with ID 2019987 (an object, a specific 'thing')
 - An object must be uniquely identified
- **Example:**
 - Integer – a class. Number – a class. Car – a class.
 - 15 is an object of Integer, 0 is an object of integer.

Class and Objects

- A object must be uniquely identifiable and it must have state
 - my book, this pen, student Fatima,
- A class is a structure of similar objects, a single object is not identified
 - Pen, Book, City.
- An object is not a class, objects that share no common structure and behaviour cannot be grouped in a class;



BankAccount Example



An Object has:

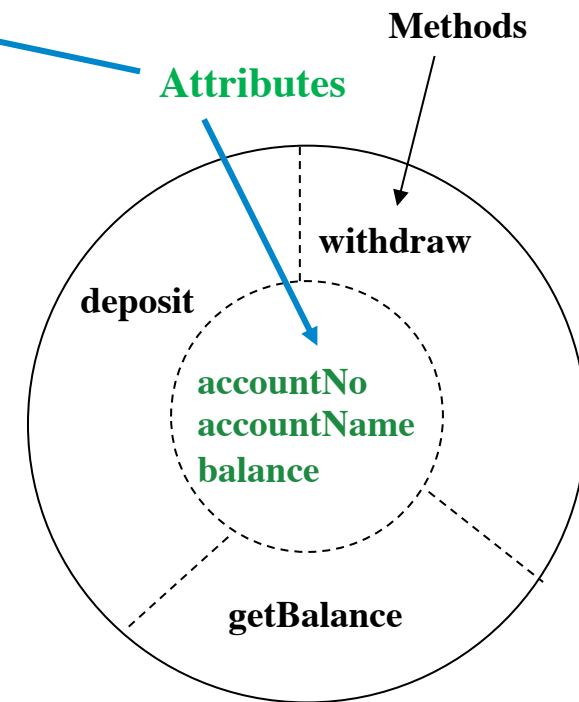
- **Attributes** – information about the object
- **Methods** – functions the object can perform

BankAccount contains **attributes**
and **methods**

Class Example

```
public class Account {  
    private int accountNo;  
    private String accountName;  
    private double balance;  
  
    public void deposit(double amount) {  
        balance += amount;  
    }  
    public void withdraw(double amount) {  
        balance -= amount;  
    }  
    public double getBalance() {  
        return balance;  
    }  
    .....  
}
```

private data



Bank Account Object

Java Naming Conventions

- Start classes with uppercase letters

```
/** Short description of the class */  
public class MyClass {  
    ...  
}
```



Use JavaDoc-style
comments

- **Start other things with lowercase letters:** attributes, local variables, methods, method parameters

```
public class MyClass {  
    private String firstName, lastName;  
  
    public String getFullName() {  
        String name = firstName + " " + lastName;  
        return name;  
    }  
}
```


Using a class: Instantiation

- To use a class you must create an object from a class (this is called **instantiation**)
- An object is an **instance** of a class
- Instantiation = Object creation with **new** keyword
e.g., `Account myAcc = new Account();`
 - This declares `myAcc` object of `Account` class.
 - The object is then created using the **new** keyword
- **Memory is allocated for the object's** attributes as defined in the class

Object Oriented Example

- Describe a rectangle
 - What properties does it have? (Attributes)
 - What operations do we want to perform with it? (Methods)
- Objects have two general capabilities:
 - Store data in **attributes**
 - Perform operations using **methods**.

Object Oriented Solution

First, define what a rectangle looks like: **Class definition**


```
public class Rectangle {  
    public int width; //These two are attributes  
    public int height;  
  
    public int computeArea() { //this is a method  
        return width*height;  
    }  
}
```

Next, use it to solve the problem: Object instantiation from class **Rectangle**

```
public class App {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle(); //object creation/instantiation  
        r.width = 5;  
        r.height = 7;  
        int area = r.computeArea();  
        System.out.printf("Area is %d\n",area);  
    }  
}
```

main() method

- The `main()` method that you can use in a class is a built-in Java method that runs your program
- Any code inside `main()` method is executed.



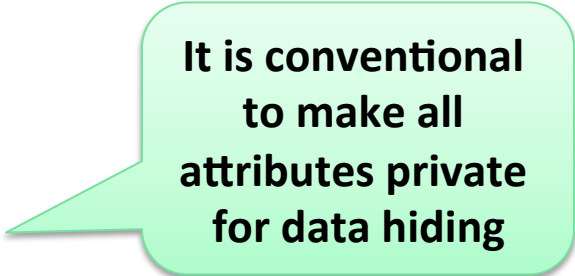
```
public class App {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle(); //object creation/instantiation  
        r.width = 5;  
        r.height = 7;  
        int area = r.computeArea();  
        System.out.printf("Area is %d\n",area);  
    }  
}
```

Attributes

Attributes

- **Attributes** = data that is stored inside an object. Also called 'Instance variables' or 'data members'
- Syntax

```
public class MyClass {  
    private SomeType attribute1;  
    private SomeType attribute2;  
}
```



It is conventional
to make all
attributes private
for data hiding

- Attributes = Local variables in a class definition
- Exist throughout the life of the object

Attributes

- **Each object of a class maintains its own copy of attributes**
 - e.g., different accounts can have different balances
- Access-specifier **private** used for Data hiding
 - Makes an attribute or a method accessible only to methods of the class
- An attempt to access a `private` attribute outside a class is a compilation error

Accessor methods

- Ideas
 - Attributes should always be private
 - And made accessible to outside world with **get** and **set** methods (also known as Accessor methods)
- Syntax

```
public class MyClass {  
    private String firstName;  
  
    public void setFirstName(String fName)  
    { firstName = fName; }  
  
    public String getFirstName()  
    { return firstName; }  
}
```
- Motivation
 - Allow data validation before assigning values to the object attributes
 - Limits accidental changes

Accessor methods

get Methods and *set* Methods

- Best practice is to provide a **get** method to **read** an attribute and a **set** method to **write** to an attribute
 - Data is protected from the client. Get and set methods are used rather than directly accessing the attributes
 - Using *set* and *get* functions allows the programmer to control how clients access private data + **allow data validation:**
 - Can return errors indicating that attempts were made to assign invalid data
 - *set* and *get* methods should also be used even inside the class

UML Diagram and Java Code

Implementation of the class **Rectangle**

Design of the class

Rectangle

Rectangle
- width : double - length : double
+ setWidth(w : double) : void + setLength(l : double): void + getWidth() : double + getLength() : double + getArea() : double

Minus sign (-) denotes private
Plus sign (+) denotes public

```
public class Rectangle
{
    private double width;
    private double length;

    public void setWidth(double w)
    {
        width = w;
    }

    public void setLength(double l)
    {
        length = l;
    }

    public double getWidth()
    {
        return width;
    }

    public double getLength()
    {
        return length;
    }

    public double getArea()
    {
        return length * width;
    }
}
```

Methods

Methods

- Definition
 - Functions that are defined inside a class. Also called “member functions”.
- Syntax

```
public ReturnType methodName(Type1 param1, Type2  
    param2, ...) {  
    ...           Use void if the method returns nothing.  
    return somethingOfReturnType;  
}
```

- Motivation
 - If the method is called only by other methods in the same class, make it private.
 - Lets an object calculate values or do computations, usually based on its current state (i.e. attributes)
 - In OOP, objects have **state** and **behavior**. The **methods provide the behavior**

Calling Methods

- The usual way that you call a method is by doing the following:

`objectName.methodName(argumentsToMethod);`

- For example,

```
Rectangle r1 = new Rectangle();  
r1.setWidth(3.0); //calling the method  
r1.setLength(4.0); //calling the method  
System.out.println(r1.getArea());  
//calling the method and print the return value.
```

Method Visibility

- public/private distinction
 - A declaration of **private** means that “outside” methods cannot call it – only methods within the same class can
 - Attempting to call a private method outside the class would result in an error at compile time
 - Only use **public** for methods that *your class will make available to users*

Methods Overloading

- Idea
 - Classes can have more than one method with **the same name**
 - The methods have to differ from each other by **having different number or types of parameters** (or both), so that Java can always tell which one you mean
 - A method's name and number and type of parameters is called the **signature**
- Syntax

```
public class MyClass {  
    public double getRandomNum() { ...};  
    public double getRandomNum(double range)  
    { ... }  
}
```
- Motivation
 - Lets you have the same name for methods doing similar operations (**ease learning and understanding your program**)

Overloading and Return Type

- You must not overload a method where the only difference is the type of value returned

```
/**  
 Returns the weight of the pet.  
 */  
public double getWeight()  
  
/**  
 Returns '+' if overweight, '-' if  
 underweight, and '*' if weight is OK.  
 */  
public char getWeight()
```



Constructors

Overview

- Definition
 - A **constructor** in Java is a **special method** that is used to initialize objects.
 - The **constructor** is called when an object of a class is created.
 - **Constructors** = 'Methods' used to initialize an object's attributes when it is created
 - Constructor is a special initialization method called when an object is created with **new**.
- Syntax
 - Constructor has the same name as the class and has no return type (not even void).

```
public class MyClass {  
    public MyClass(Type paramName, ... ) { ... }  
}
```

Initializing Objects with Constructors

- A class can have one or more different constructors
 - We distinguish between constructors using different number and types of parameters
- Default constructor has **no parameters** and has an empty body
 - Java will define this automatically if the class does not have any constructors
 - If you do define a constructor, Java will not automatically define a default constructor

Adding constructors to the Rectangle class

```
public class Rectangle{
    private double width;
    private double length;

    public Rectangle(){ //constructor
    }

    public Rectangle(double w, double l){ //constructor
        width = w;
        length = l;
    }
    public void setWidth(double w){
        width = w;
    }
    public void setLength(double l){
        length = l;
    }
    public double getWidth(){
        return width;
    }
    public double getLength(){
        return length;
    }
    public double getArea(){
        return length * width;
    }
}
```

The `this` Variable

- The `this` object reference can be used **inside any non-static method** to **refer to the current object**
 - `this` is used to **refer to the object** from inside its class definition
 - The keyword `this` stands for the receiving object
- `this` is commonly used to **resolve name conflicts**
 - Using **`this`** permits the use of attributes in methods that have local variables / parameters with the same name

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

This is a parameter

This is an attribute of `setName()` method. To avoid confusion we add **`this.`** in-front of it

Constructor Example

- Constructor = special method that handles the object initialization
- A constructor **is used to initialize the objects during object construction.**
- Example:
Account saraAcct = **new Account**(123, "Sara", 100.0);

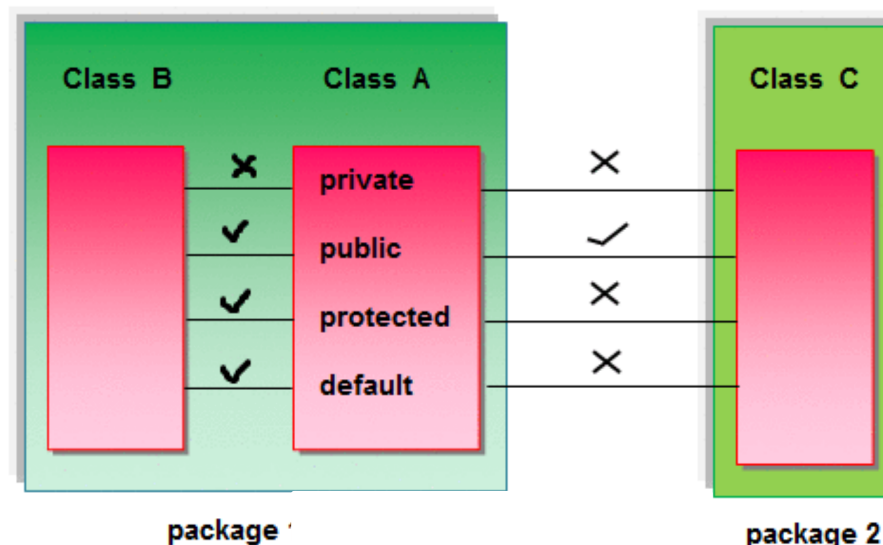
```
public class Account {  
    private int id;  
    private String name;  
    private double balance;  
  
    public Account (int id, String name,  
double balance) {  
        this.id = id;  
        this.name = name;  
        this.balance = balance;  
    }  
  
    ...  
}
```

Constructor

Access Modifiers

Access Modifiers

- Java language has four access modifier to control access to classes, attributes, methods and constructors.
 - **Private:** visible only within the class
 - **Default (no modifier):** visible only within the same package.
Very rarely used. Don't omit modifier without a good reason.
 - **Protected:** visible within the same package and to **sub classes outside the package.**
 - **Public:** visible everywhere



Access Modifiers

- To use a modifier, you include its keyword (**public, private, protected**) in the definition of a class, method, or variable.
- The modifier precedes the rest of the statement.

Access Modifiers Summary

Modifier	Class	Package	Subclass	Global
Public	✓	✓	✓	✓
Protected	✓	✓	✓	✗
Default	✓	✓	✗	✗
Private	✓	✗	✗	✗

Variables

- **Local variables:**
 - variables defined inside methods, constructors or blocks are called local variables.
 - The variable will be declared and initialized within the method and the variable will be destroyed when the method is completed.
- **Instance variables:**
 - Instance variables are variables defined inside a class but outside any method.
 - These variables are instantiated when objects are created. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Static (or, Class) variables:**
 - Class variables are variables declared within a class, outside any method, with the static keyword.

Local variables

- Local variables are declared in methods, constructors, or blocks.
- Access modifiers **cannot** be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- There is **no default value for local variables** so local variables should be declared and an initial value should be assigned before the first use.

Instance variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the key word 'new' and destroyed when the object is destroyed.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class.
- Normally it is recommended to make these variables private (access level).
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class.

Class/static variables

- Class variables also known as static variables are declared with the *static keyword* in a class, *but outside a method, constructor or a block*.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
 - All objects of that class *share* that same one copy (*global variable*)
 - Any change in the static variable can be seen by all objects
- When declaring class variables as public *static final*, then variables names (*constants*) are all in upper case. Constant variables never change from their initial value.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables.
- Static variables can be accessed by calling with the class name.
ClassName.VariableName.

Static Methods

- A static method belongs to the class rather than the object of the class.
 - Also known as “class methods” (vs. “instance methods”)
- A static method can only access the static attributes.
- A static method can be called without the need for creating an instance of the class.
 - You call a static method through the class name

ClassName.functionName(arguments);

- For example, the **Math** class has a static method called **cos**
 - You can call **Math.cos(3.5)** without creating an object of the **Math** class
- E.g., the `main` method is a static method so the system can call it without first creating an object

static versus **public** Methods and Attributes

- You will often see Java programs that have either **static** or **public** attributes and methods.
- A **static** method can be accessed without creating an object of the class, unlike **public**, which can only be accessed by objects

Example: Static versus Public

```
public class Main {  
    // Static method  
    static void myStaticMethod() {  
        System.out.println("Static methods can be called without creating  
objects");  
    }  
  
    // Public method  
    public void myPublicMethod() {  
        System.out.println("Public methods must be called by creating  
objects");  
    }  
  
    // Main method  
    public static void main(String[] args) {  
        myStaticMethod(); // Call the static method  
        // myPublicMethod(); This would compile an error  
  
        Main myObj = new Main(); // Create an object of Main  
        myObj.myPublicMethod(); // Call the public method on the object  
    }  
}
```

Wrapper Class Example: Static methods in Character class

Name	Description	Argument Type	Return Type	Examples	Return Value
toUpperCase	Convert to uppercase	char	char	Character.toUpperCase('a') Character.toUpperCase('A')	'A' 'A'
toLowerCase	Convert to lowercase	char	char	Character.toLowerCase('a') Character.toLowerCase('A')	'a' 'a'
isUpperCase	Test for uppercase	char	boolean	Character.isUpperCase('A') Character.isUpperCase('a')	true false
isLowerCase	Test for lowercase	char	boolean	Character.isLowerCase('A') Character.isLowerCase('a')	false true
isLetter	Test for a letter	char	boolean	Character.isLetter('A') Character.isLetter('%')	true false
isDigit	Test for a digit	char	boolean	Character.isDigit('5') Character.isDigit('A')	true false
isWhitespace	Test for whitespace	char	boolean	Character.isWhitespace(' ') Character.isWhitespace('A')	true false
Whitespace characters are those that print as white space, such as the blank, the tab character ('\t'), and the line-break character ('\n').					

Enumerations

- An `enum` is a special class that represents a group of constants
- To create an `enum`, use the `enum` keyword, and separate the constants with a comma
- Constants should be in uppercase letters:

```
enum Level {  
    LOW,  
    HIGH }
```

- You can access `enum` with a `dot` syntax: `Level myVar = Level.HIGH;`
- You can group a set of **related constant values** in an `enum` type (short for enumeration).
- It makes sense to create an `enum` type when the possible values of a variable can be enumerated (e.g., Gender, Direction, Days of week, Months of year)
- The `enum` type may optionally include constructors, attributes and methods

Enumerations (Cont.)

- Each **enum** internally implemented by using Class with the following restrictions:
 - **enum** constants are implicitly **final**, because they declare constants that shouldn't be modified.
 - **enum** constants are implicitly **static**.
 - Any attempt to create an object of an enum type with operator **new** results in a compilation error.
 - **enum** constants can be used anywhere constants can be used.
- For every **enum**, the compiler generates the **static** method **values()** that returns an array of the **enum**'s constants.
- When an **enum** constant is converted to a **String**, the constant's identifier is used as the **String** representation.

Example 1

```
public class Main {  
    enum Level {  
        LOW,  
        MEDIUM,  
        HIGH  
    }  
  
    public static void main(String[] args) {  
        Level myVar = Level.MEDIUM;  
        System.out.println(myVar);  
    }  
}
```

Enum has a values() method, which returns an array of all enum constants. It is useful when you want to use it with loop:

```
for (Level myVar : Level.values()) {  
    System.out.println(myVar);  
}
```

Example 2

```
enum Level {  
    LOW,  
    EDIUM,  
    HIGH  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Level myVar = Level.MEDIUM;  
  
        switch(myVar) {  
            case LOW:  
                System.out.println("Low level");  
                break;  
            case MEDIUM:  
                System.out.println("Medium level");  
                break;  
            case HIGH:  
                System.out.println("High level");  
                break;  
        }  
    }  
}
```

enum is actually a class



/ Internally above enum Color is converted to Color class */*

```
class Color
{
    public static final Color RED = new Color();
    public static final Color BLUE = new Color();
    public static final Color GREEN = new Color();
}
```

TrafficLight Enum Example

- You can enhance the enum class with instance attributes and methods

```
public enum TrafficLight {  
    // Each object initialize its associated duration.  
    GREEN(50),  
    YELLOW(4),  
    RED(60);  
  
    private final int duration;  
    // Private constructor to set the duration.  
    private TrafficLight(int duration) {  
        this.duration = duration;  
    }  
  
    // Public accessor to get the duration.  
    public int getDuration() {  
        return duration;  
    }  
}
```


Enum Usage Example

```
public static void main(String[] args) {  
    TrafficLight lightState = TrafficLight.GREEN;  
    // String to Enum value  
    lightState = TrafficLight.valueOf("GREEN");  
  
    System.out.println("LightState value: " + lightState +  
        " - lightState.toString(): " +  
            lightState.toString());  
  
    for (var state : TrafficLight.values()) {  
        System.out.println(state + " stays on for " +  
            state.getDuration() + "s");  
    }  
}
```

```
lightState value: GREEN - lightState.toString(): GREEN  
GREEN stays on for 50s  
YELLOW stays on for 4s  
RED stays on for 60s
```

Difference between Enums and Classes

- An **enum** can, just like a class, have attributes and methods.
- The only difference is that enum constants are **public**, **static** and **final** (unchangeable - cannot be overridden).
- An **enum** cannot be used to create objects, and it cannot extend other classes (but it can implement interfaces).

Why And When To Use **enum**?

- Use **enum** when you have values that you know aren't going to change, like month days, days, colors, deck of cards, etc.

Programming Conventions

- Make all attributes **private** and provide **getters** and **setters**
- Class names start with **upper case**
- Method names and variable names start with **lower case**
- Choose **meaningful names** for classes, methods and variables
 - makes programs more readable and understandable
- Use JavaDoc-style comments to generate useful documentation
- Indent nested blocks consistently to communicate the structure of your program
 - **Proper indentation helps comprehension**