

נושאים להיום

- מתודות ושדות סטטיים
- API
- ENCAPSULATION
- כלים שימושיים

Static

Static Keyword

- המילה השמורה 'static' ב Java משמשת להגדיר אלמנטים ברמת המחלקה המשותפים לכל המופעים של המחלקה.
- כאשר חבר (משתנה או שיטה) מוגדר כסטטי, הוא שייך למחלקה עצמה ולא לאף מופע מסוים של המחלקה.
- יש רק עותק אחד של חבר סטטי, ללא קשר לכמה מופעים של המחלקה נוצרו.
- דוגמאות נפוצות לחברים סטטיים כוללות קבועים, ממוצעים ומשאבים משותפים.
- משתנים סטטיים מאותחלים פעם אחת **כאשר המחלקה נטענת** ונשמרים לאורך זמן הריצה של התוכנית.
- שיטות סטטיות ניתן לקרוא להן ישירות על המחלקה עצמה, מבלי ליצור אינסטנס.
- גישה לחברים סטטיים: `ClassName.memberName`
- **זהירות:** חברים סטטיים עשויים ליצור tight coupling ובעיות אפשריות בסביבה מרובת threads, לכן יש להשתמש בהם בזהירות.
- יתרון: יעילות בשימוש בזיכרון מאחר ולא משכפל את אותו המידע עבור כל מופע.

Examples:


Multithreaded Environments:

In programming, multithreading involves running multiple threads (smaller units of a program) concurrently. When static members are accessed and modified by multiple threads simultaneously, it can lead to race conditions, where the order of operations is not deterministic and can vary with different thread execution speeds. This can result in unexpected and incorrect behavior of the program.

WRONG!

```
public class Counter {  
    2 usages  
    private static int count = 0;  
    no usages  
    public static void increment() {  
        count++;  
    }  
    no usages  
    public static int getCount() {  
        return count;  
    }  
}
```

RIGHT



```
public class Counter {  
    private static int count = 0;  
    private static Object lock = new  
Object();  
    public static void increment() {  
        synchronized (lock) {  
            count++;  
        }  
    }  
  
    public static int getCount() {  
        synchronized (lock) {  
            return count;  
        }  
    }  
}
```

API

What is an API?!

הגדרה: "API מייצג ממשק תכנות יישומים. הוא מגדיר קבוצה של כללים ופרוטוקולים לבנייה ואינטראקציה עם יישומי תוכנה."

- מקל על שימוש חוזר בקוד
- מעודד עיצוב מודולרי
- מקל על שיתוף פעולה בין רכיבי תוכנה שונים

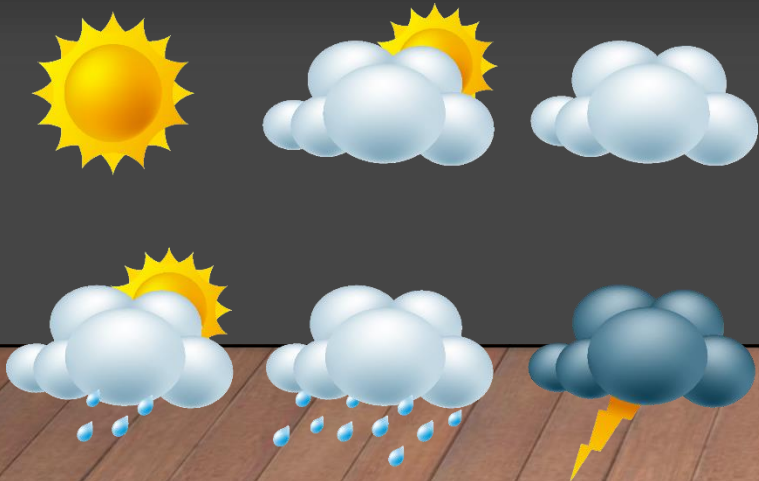
כמה שיותר קטן יותר טוב

חשוב לא לפרט איך הפרטים ממומשים, אלא רק איך להשתמש ב API!
שמירה על ממשק API מינימליסטי חיונית מכמה סיבות:

1. ממשק API מינימליסטי קל יותר למפתחים להבין ולהשתמש בו.
2. פשטות בתחזוקה:
בדרך כלל קל יותר לתחזק ממשק API מינימליסטי. עם פחות תכונות, יש פחות נקודות כשל פוטנציאליות, וסביר להניח ששינויים לא יהיו בעלי השלכות לא מכוונות.
3. פיתוח מהיר יותר למשתמשים:
מפתחים יכולים לשלב ממשק API מינימליסטי מהר יותר באפליקציות שלהם. זה יתרון במיוחד כאשר זמן היציאה לשוק הוא גורם קריטי.
4. תיעוד ברור יותר:
ממשקי API מינימליסטיים מובילים לתיעוד ברור ותמציתי יותר. מפתחים יכולים למצוא במהירות את המידע שהם צריכים מבלי להיות מוצפים בפרטים מיותרים.
תאימות ויכולת פעולה הדדית:

REAL LIFE EXAMPLE

```
1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 import java.net.HttpURLConnection;
5 import java.net.URL;
6
7 public class Main {
8     private static final String API_KEY = "c708d99e720ab089337418a57dd9942b";
9     private static final String API_URL = "http://api.openweathermap.org/data/2.5/forecast";
10
11     public static void main(String[] args) {
12         double lat = 33.44;
13         double lot = -94.04;
14
15         try {
16             String weatherData = getWeatherData(lat, lot);
17             System.out.println("Weather Information for " + lat + ", " + lot + "
18             System.out.println(weatherData);
19         } catch (IOException e) {
20             e.printStackTrace();
21         }
22     }
23
24     private static String getWeatherData(double latitude, double longitude) throws IOException {
25         String apiUrl = String.format("%s?lat=%s&lon=%s&appid=%s", API_URL, latitude, longitude,
26         API_KEY);
27
28         URL url = new URL(apiUrl);
29         HttpURLConnection connection = (HttpURLConnection) url.openConnection();
30
31         try (
32             BufferedReader reader = new BufferedReader(
33                 new InputStreamReader(connection.getInputStream()))
34         ) {
35             StringBuilder response = new StringBuilder();
36             String line;
37             while ((line = reader.readLine()) != null) {
38                 response.append(line);
39             }
40             return response.toString();
41         } catch (IOException e) {
42             // Handle HTTP response code other than 2xx (success)
43             if (connection.getResponseCode() != HttpURLConnection.HTTP_OK) {
44                 throw new IOException("HTTP error code: " + connection.getResponseCode());
45             }
46             throw e;
47         } finally {
48             connection.disconnect();
49         }
50     }
51 }
```



> Encapsulation

This key principle tells us to keep data and code that can manipulate this data together. It is also about keeping data and the code safe from external interference.

> Encapsulation

☐ Private

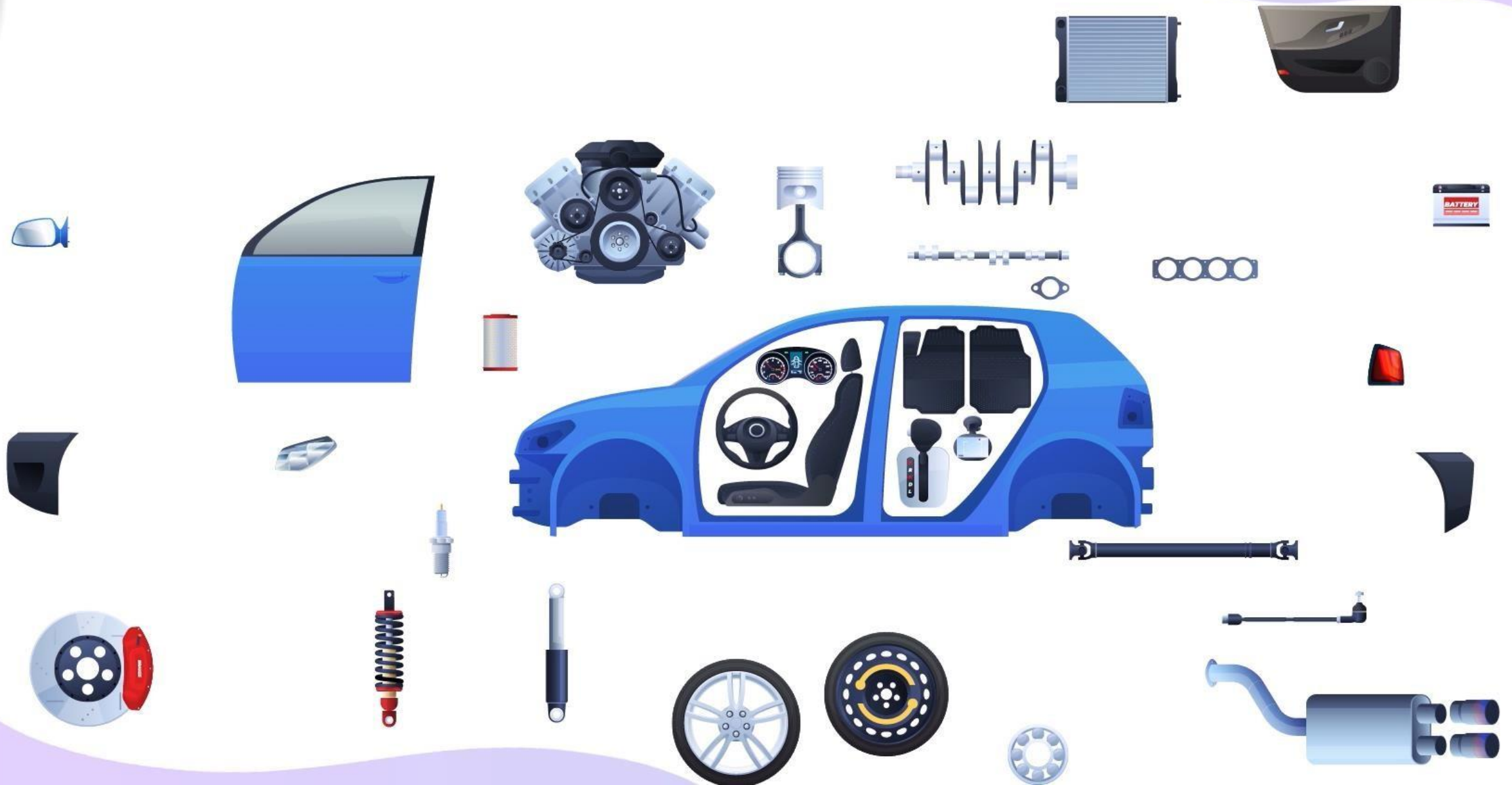
☐ Default

☐ Protected

☐ Public

```
▼ services
  ▼ impl
    > DefaultOrderManagementService.java
    > DefaultProductManagementService.java
    > DefaultUserManagementService.java
    > OrderManagementService.java
    > ProductManagementService.java
    > UserManagementService.java
```

> Encapsulation



CLASSES 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

```
public class Person {  
    private String name; // private = restricted access  
  
    // Getter  
    public String getName() {  
        return name;  
    }  
  
    // Setter  
    public void setName(String newName) {  
        this.name = newName;  
    }  
}
```

CLASSES MODIFIER

Modifier	Description
<code>public</code>	The class is accessible by any other class
<code>default</code>	The class is only accessible by classes in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
<code>public</code>	The code is accessible for all classes
<code>private</code>	The code is only accessible within the declared class
<code>default</code>	The code is only accessible in the same package. This is used when you don't specify a modifier. You will learn more about packages in the Packages chapter
<code>protected</code>	The code is accessible in the same package and subclasses . You will learn more about subclasses and superclasses in the Inheritance chapter

Non-Access Modifiers

For **classes**, you can use either `final` or `abstract` :

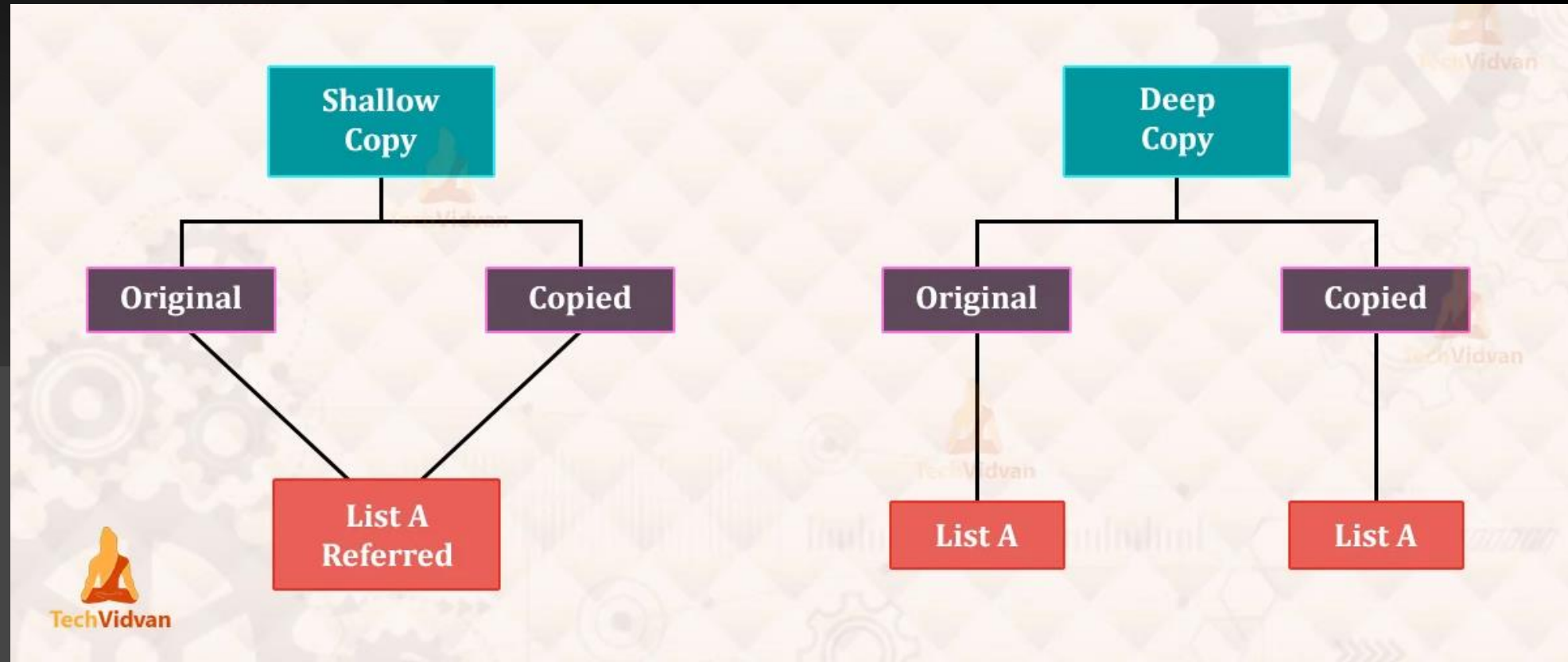
Modifier	Description
<code>final</code>	The class cannot be inherited by other classes (You will learn more about inheritance in the Inheritance chapter)
<code>abstract</code>	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters)

NAMING CONVENTIONS

Naming Convention

- ☐ Classes are started with capital letter (Integer, String)
- ☐ Method are started with lower-case letter and upercasing all other first letters
- ☐ Constants are named with all upper case letters and underscores

DEEP COPY



<https://www.youtube.com/watch?v=OaCYMgyprtc>

DEEP COPY

Shallow copy	Deep copy
משתנים פרימיטיביים	מחלקות
מחלקות שלא ניתנות לשינוי	מערכים

<https://www.youtube.com/watch?v=OaCYMgyprtc>

SHORTEN IF

The value of a variable often depends on whether a particular Boolean expression is or is not true and on nothing else.

For instance, one common operation is setting the value of a variable to the maximum of two quantities. In Java you might write

```
if (a > b) {  
    max = a;  
}  
else {  
    max = b;  
}
```

Setting a single variable to one of two states based on a single condition is such a common use of if-else that a shortcut has been devised for it

, the conditional operator, `?:`. Using the conditional operator you can rewrite the above example in a single line like this:

```
max = (a > b) ? a : b;
```

FOR EACH

```
String[] stringArr = {"\nThis", "is", "for", "each", " :)\n"};  
for (String s : stringArr) {  
    System.out.print(s+" ");  
}
```

Out:

```
This is for each :)  
|
```


ENUM

An Enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).

To create an Enum, use the Enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

```
public enum Priority {  
    HIGH, MEDIUM, LOW;  
}
```

Example

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

You can access `enum` constants with the **dot** syntax:

```
Level myVar = Level.MEDIUM;
```


ENUM

```
public class EnumDemo {  
  
    public static void main(String[] args) {  
  
        Priority priority = Priority.HIGH;  
  
        switch (priority) {  
            case HIGH:  
                System.out.println("High priority");  
                break;  
            case MEDIUM:  
                System.out.println("Medium priority");  
                break;  
            case LOW:  
                System.out.println("Low priority");  
                break;  
        }  
  
        System.out.println("===== Enum valueOf()");  
  
        Priority priority2 = Priority.valueOf("HIGH");  
        System.out.println(priority2);  
  
        // priority2 = Priority.valueOf("high"); // java.lang.IllegalArgumentException: No enum constant com.itbulls.learnit.javacore.enumerations.Priority.high  
    }  
}
```

ENUM

```
System.out.println("===== Enum comparison");

System.out.println("Priority.HIGH == Priority.MEDIUM: "
                  + (priority == Priority.MEDIUM));    // false

System.out.println("Priority.HIGH == Priority.HIGH: "
                  + (priority == Priority.HIGH));    // true

System.out.println("===== Enum ordinal()");

System.out.println("Priority.HIGH.ordinal(): " + Priority.HIGH.ordinal());
System.out.println("Priority.MEDIUM.ordinal(): " + Priority.MEDIUM.ordinal());

System.out.println("===== Enum iteration");

Priority[] values = Priority.values();
for (Priority priority3 : values) {
    System.out.println(priority3);
}
```

ENUM

```
public enum Month {  
  
    JANUARY(31), FEBRUARY(28), MARCH(31), APRIL(30), MAY(31), JUNE(30), JULY(31),  
    AUGUST(31), SEPTEMBER(30), OCTOBER(31), NOVEMBER(30), DECEMBER(31);  
  
    private int daysAmount;  
  
    private Month(int daysAmount) {  
        this.daysAmount = daysAmount;  
    }  
  
    public int getDaysAmount() {  
        return this.daysAmount;  
    }  
  
}
```

```
System.out.println("===== Enum fields and methods");
```

```
System.out.println("Month.JANUARY.getDaysAmount(): " + Month.JANUARY.getDaysAmount());
```