

## Packages, Enumeration & Exceptions

**Dr. Abdelkarim Erradi**  
**CSE@QU**

# Outline

- **Java Packages**
- **Access Modifiers**
- **Enumeration**
- **Exceptions**

# Packages



# Packages

- Packages in Java are a **way of grouping related classes** together:
  - Classes performing a specific set of tasks or providing similar functionality.
- **Package = directory.** A package name is the same as the directory (folder) name which contains the .java files.
- Two main reasons packages are used:
  - Code organization: grouping functionally related classes into a package to make it easier to find and use classes
  - Avoid names collision: distinguish between classes with the same name but belong to different packages

# Built-in Packages



- Java fundamental classes are in *java.lang*, classes for reading and writing (input and output) are in *java.io*, lists and collections in *java.util* and so on.
- To use a class from a package, first **import** it. E.g.,

```
import java.util.ArrayList;  
import java.util.List;
```

# Creating a Package

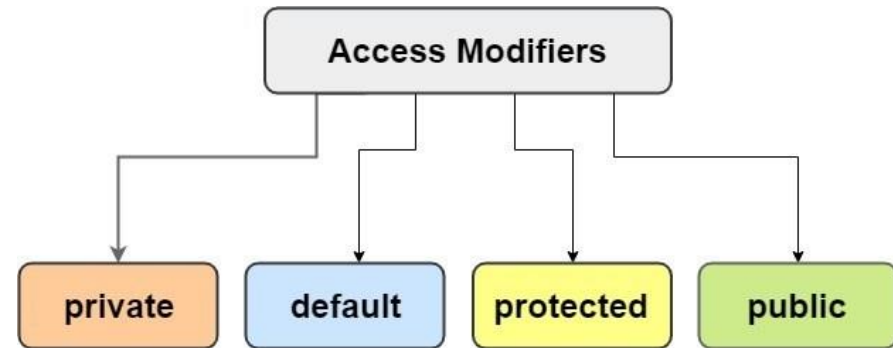
- To create a package, you add **package** **statement** with the package name at the top of every source file that you want to include in the package

```
package quBank;
```

```
public class Account {  
    // OOP Principle of Encapsulation:  
    // all attributes are private  
    private int id;  
    private String name;  
    private String type;  
    private double balance;  
    ...  
}
```

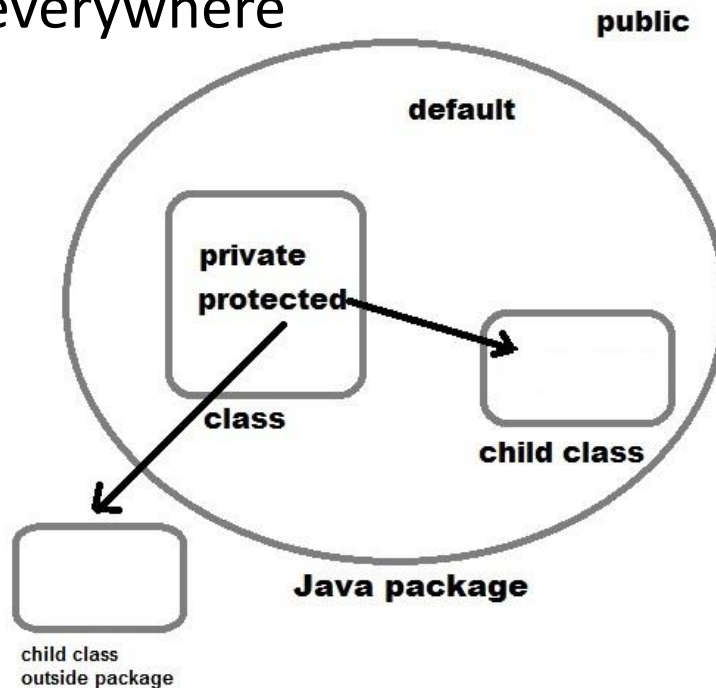
- All .java files in **quBank** package will be saved in quBank folder
- Package names are usually written in lowercase

# Access Modifiers



# Access Modifiers

- Java language has four access modifier to control access to classes, attributes, methods and constructors.
  - Private: visible only within the classes
  - Default: visible only inside the same package
  - Protected: visible within the package and all sub classes
  - Public: is visible everywhere





# Access Modifiers Summary

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



# Enumeration

```
enum LightState {...}
```

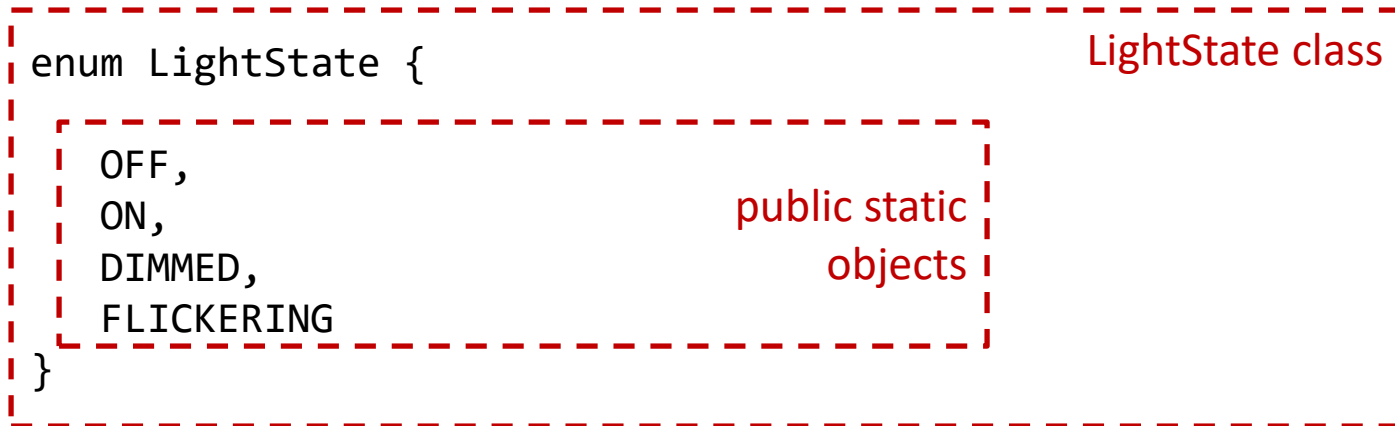
# Enumerations

- The basic enum type defines a set of constants represented as unique identifiers
- An enum type is declared with an **enum declaration**, which is a comma-separated list of enum constants
- The declaration may optionally include constructors, attributes and methods

# Enumerations (Cont.)

- Each `enum` declaration declares an `enum` 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, such as in the `case` labels of `switch` statements and the condition of an `if` statement.
- 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.

# enum is actually a class



## <EnumDemo.java>

```
public class EnumDemo {
    enum LightState {
        // Each object is initialized to a color.
        OFF("black"),
        ON("white"),
        DIMMED("gray"),
        FLICKERING("red");

        private final String colorField;

        // Private constructor to set the color.
        private LightState(String color) {
            colorField = color;
        }

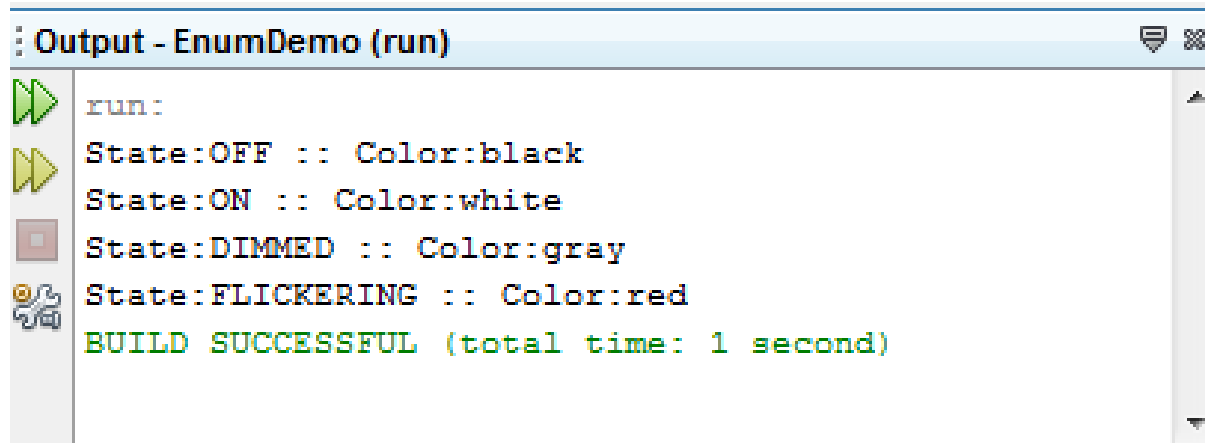
        // Public accessor to get color.
        public String getColor() {
            return colorField;
        }
    }

    public static void main(String[] args) {
        LightState off = LightState.OFF;
        LightState on = LightState.ON;
        LightState dimmed = LightState.DIMMED;
        LightState flickering = LightState.FLICKERING;
    }
}
```

You can enhance the enum class with instance variables and methods

# <EnumDemo.java>

```
System.out.println("State:" + off.toString() +  
                    " :: Color:" + off.getColor());  
System.out.println("State:" + on.toString() +  
                    " :: Color:" + on.getColor());  
System.out.println("State:" + dimmed.toString() +  
                    " :: Color:" + dimmed.getColor());  
System.out.println("State:" + flickering.toString() +  
                    " :: Color:" + flickering.getColor());  
    }// end main()  
}// end EnumDemo
```



The screenshot shows an IDE's Output window titled "Output - EnumDemo (run)". The window contains the following text:

```
run:  
State:OFF :: Color:black  
State:ON :: Color:white  
State:DIMMED :: Color:gray  
State:FLICKERING :: Color:red  
BUILD SUCCESSFUL (total time: 1 second)
```

On the left side of the output window, there are four icons: a green play button, a yellow play button, a red square, and a blue icon with a magnifying glass.



Exceptions

**Error.**



# Throwing Exceptions

```
1 // Time1.java
2 // Time1 class declaration maintains the time in 24-hour format.
3
4 public class Time1 {
5     private int hour; // 0 - 23
6     private int minute; // 0 - 59
7     private int second; // 0 - 59
8
9     // set a new time value using universal time; throw an
10    // exception if the hour, minute or second is invalid
11    public void setTime(int hour, int minute, int second) {
12        // validate hour, minute and second
13        if (hour < 0 || hour >= 24 || minute < 0 || minute >= 60 ||
14            second < 0 || second >= 60) {
15            throw new IllegalArgumentException(
16                "hour, minute and/or second was out of range");
17        }
18
19        this.hour = hour;
20        this.minute = minute;
21        this.second = second;
22    }
```

# Throwing Exceptions

- Method `setTime` declares three `int` parameters and uses them to set the time.
- Lines 13–14 test each argument to determine whether the value is outside the proper range.
- For incorrect values, `setTime` throws an exception of type `IllegalArgumentException`
  - Notifies the client code that an invalid argument was passed to the method.
  - The **throw statement** creates a new object of type `IllegalArgumentException` and specifies a custom error message.
  - `throw` statement immediately terminates method `setTime` and the exception is returned to the calling method that attempted to set the time.

# try and catch

---

```
18 // attempt to set time with invalid values
19 try {
20     time.setTime(99, 99, 99); // all values out of range
21 }
22 catch (IllegalArgumentException e) {
23     System.out.printf("Exception: %s%n%n", e.getMessage());
24 }
25
26 // display time after attempt to set invalid values
27 displayTime("After calling setTime with invalid values", time);
28 }
29
30 // displays a Time1 object in 24-hour and 12-hour formats
31 private static void displayTime(String header, Time1 t) {
32     System.out.printf("%s%nUniversal time: %s%nStandard time: %s%n",
33         header, t.toUniversalString(), t.toString());
34 }
35 }
```

---

Lines 19 to 24 use **try...catch** to catch and handle the exception (e.g., display the error message to the user)