

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

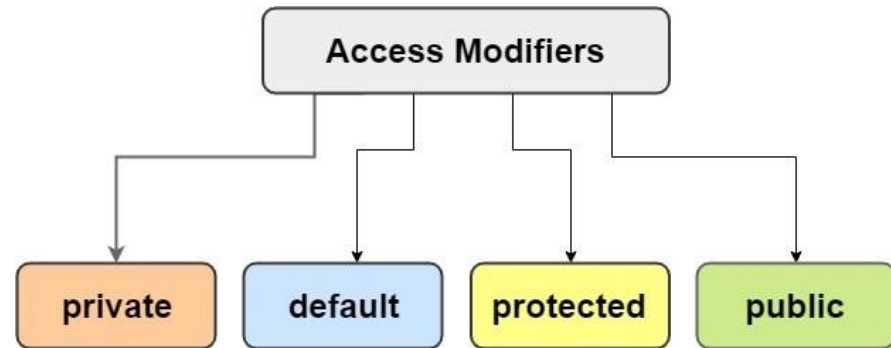
- To create a package, you add **package** **statement** with the package name at the top of every source file that you want to include in the package

```
package qu.bank;
```

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

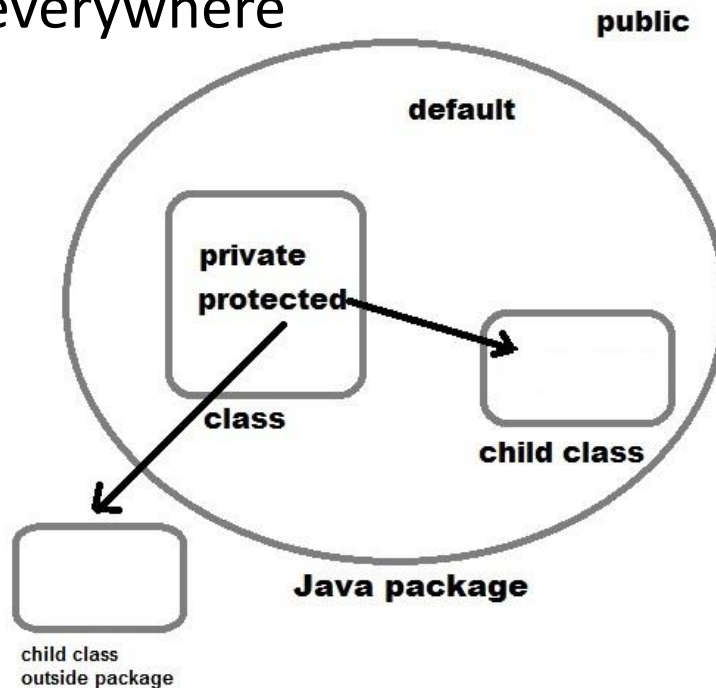
- All .java files in **qu.bank** package will be saved in qu\bank folder
- Package names are usually written in lowercase

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 {...}
```

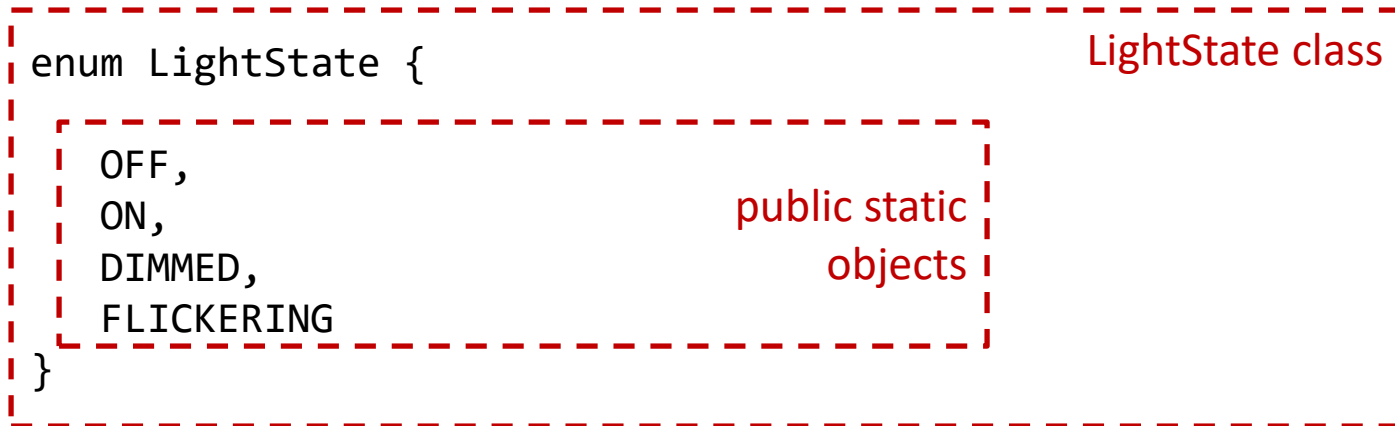
Enumerations

- 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



LightState Enum Example

```
public enum LightState {  
    // Each object is initialized to a color.  
    OFF("black"),  
    ON("white"),  
    DIMMED("gray"),  
    FLICKERING("red");  
  
    private final String color;  
  
    // Private constructor to set the color.  
    private LightState(String color) {  
        this.color = color;  
    }  
  
    // Public accessor to get color.  
    public String getColor() {  
        return color;  
    }  
}
```

You can enhance the enum class with instance attributes and methods

Enum Usage Example

```
LightState off = LightState.OFF;  
LightState on = LightState.ON;  
LightState dimmed = LightState.DIMMED;  
LightState flickering = LightState.FLICKERING;
```

```
System.out.println("Value:" + off  
    + " State:" + off.toString()  
    + " :: Color:" + off.getColor());  
System.out.println("State:" + on + " :: Color:" +  
on.getColor());  
System.out.println("State:" + dimmed + " :: Color:" +  
dimmed.getColor());  
System.out.println("State:" + flickering.toString() +  
" :: Color:" + flickering.getColor());
```

```
Value:OFF State:OFF :: Color:black  
State:ON :: Color:white  
State:DIMMED :: Color:gray  
State:FLICKERING :: Color:red
```



Exceptions

Error.

What is an Exception?

- An exception indicates a problem that occurs while a program executes.
- When the Java Virtual Machine (JVM) or a method detects a problem, such as an *invalid array index* or an *invalid method argument*, it **throws an exception**.
- e.g., trying to access an array element outside the bounds of the array.
 - Java doesn't allow this.
 - JVM checks that array indices to ensure that they are ≥ 0 and $< \text{the array's size}$. This is called **bounds checking**.
 - If a program uses an invalid index, JVM throws an exception to indicate that an error occurred in the program at execution time.

Handling Exceptions

- Exception handling helps you create **fault-tolerant programs** that can resolve (or handle) exceptions.
- To handle an exception, place any code that might throw an exception in a **try statement**.
- The **catch block** contains the code that *handles* the exception.
 - You can have many catch blocks to handle different *types* of exceptions that might be thrown in the corresponding try block
 - An exception object's **.toString** or **.getMessage** method returns the exception's error message

Handling Exceptions - Example

```
try {  
    int nums[] = {3, 5, 9};  
    System.out.println(nums[3]);  
    System.out.println("nums array size: " + nums.length);  
}  
catch (IndexOutOfBoundsException ex){  
    System.out.println(ex.getMessage());  
}
```

- The program attempts to access an element *outside* the bounds of the array
 - the array has only 3 elements (with an index 0 to 2).
- JVM throws **ArrayIndexOutOfBoundsException** to notify the program of this problem.
- At this point the **try block** terminates and the **catch block** begins executing
 - if you declared any local variables in the try block, they're now out of scope.

Handling Exceptions - Example

```
try {  
    int[] nums = null;  
    System.out.println("nums array size: " + nums.length);  
}  
catch (NullPointerException ex){  
    System.err.println(ex.toString());  
}
```

- A **NullPointerException** occurs when you try to call a method on a **null reference**.
- Ensuring that references are not null before you use them to call methods prevents Null Pointer Exceptions.

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)