

## Packages, Enumeration & Exceptions

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

# Outline





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

# Packages



# Packages




- Packages in Java are a **way of functionality grouping related classes** together.
- 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
- Packages are mapped to **nested folders** on disk to provide physical grouping of `.java` files. E.g.,

**Package** ▾  > qu.bank  
    >  > Account.java  
    >  > Bank.java  
    >  > BankUI.java

Mapped to

**Folders  
and  
files  
on Disk**

qu > bank

Name
 Account.java
 Bank.java
 BankUI.java

# 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

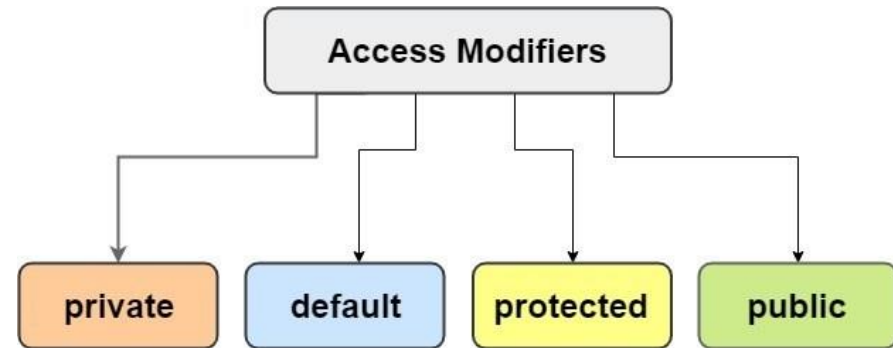
- To create a package, you add **package** statement with the package name at the top of every source file that you want to include in the package

```
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;  
    ...  
}
```

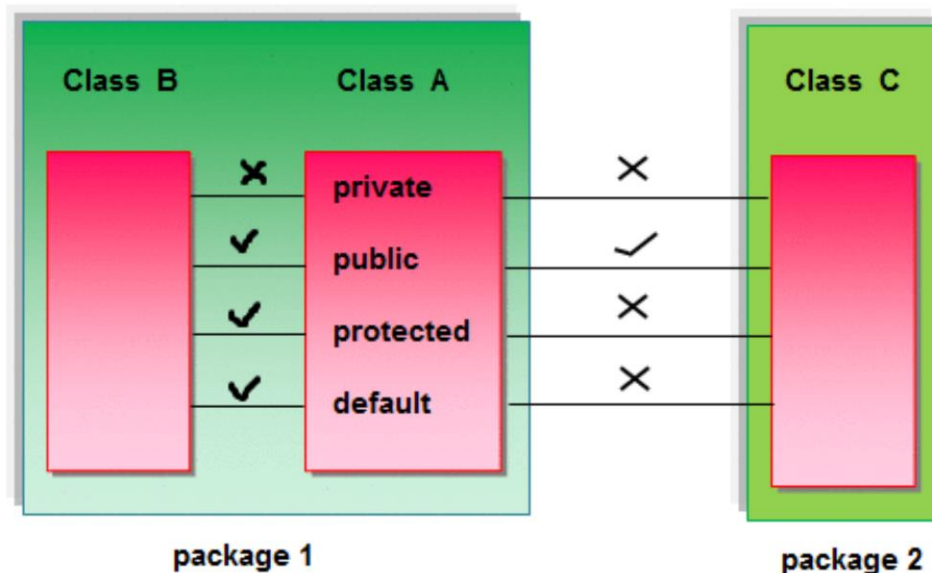
- All .java files in **qu.bank** package will be saved in **qu\bank** folder
- Package names are usually written in lowercase

# Access Modifiers



# Access Modifiers

- Java language has four access modifier to control access to classes, attributes, methods and constructors.
  - Private:** visible only within the class
  - Default (no modifier):** visible only within the same package
  - Protected:** visible within the same package and to **sub classes outside the package**.
  - Public:** visible everywhere



If *Class C* is child of *Class A* then the protected members of *A* will be visible in *C*



# Access Modifiers Summary

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



# Enumeration

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

# Enumerations

- You can group a set of **related constant values** in an **enum** type (short for enumeration).
  - For example, gender enum values could be Male, Female. The direction could be North, South, East, and West.
- 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, Month of year)
- An enum type is declared with: **enum EnumName**  
Then a comma-separated list of enum constants
- 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, 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



```
/* 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;  
  
    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



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  $\geq 0$  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 2

```
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

- Use **throw** to return an exception object to the caller

```
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)