Chapter Five

Java Exception Handling

MULUGETA G. and SUFIAN K.

Java Exception

- Exception is a problem that arises during the execution of a program.
- When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally.
  - which is not recommended, therefore, these exceptions are to be handled.
- Exception handling in java is a mechanism to handle runtime errors that can occur in a program.
  - It allows the program to continue execution even when an error occurs, rather than terminating abnormally.

Cont'd …

- An exception can occur for many different reasons. Following are some scenarios where an exception occurs.
  - A user has entered an invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications or the JVM has run out of memory.
  - Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Types of Exception

- Based on these, we have three categories of Exceptions.
- Checked exceptions:
  - is an exception that occurs at the compile time, these are also called as compile time exceptions.
  - cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.
  - Checked exceptions in Java occur due to events beyond the control of the program, such as:
    - ✓File I/O errors
    - ✓Network connection failures
    - ✓Database connection issues
    - ✓User input errors

Cont'd …

- Checked exceptions example:

```
import java.io.File;

import java.io.FileReader;
public class FilenotFound_Demo {

public static void main(String args[]) {

File file =  new File("E://file.txt");

FileReader fr =  new FileReader(file);

}}
```

Cont'd …

- Unchecked exceptions:-
  - An unchecked exception is an exception that occurs at the time of execution.
  - These are also called as Runtime Exceptions.
  - Runtime exceptions are ignored at the time of compilation.
  - These include programming bugs, such as logic errors or improper use of an API.
    - ✓null pointer exception
    - ✓an array index out of bounds exception
    - ✓or an illegal argument exception.

Cont'd …

```
public class Unchecked_Demo {
   public static void main(String args[]) {
      int num[] = {1, 2, 3, 4};
      System.out.println(num[5]);
   }
}
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
At
Exceptions.Unchecked_Demo.main(Unchecked_Demo.j
ava:8)
```

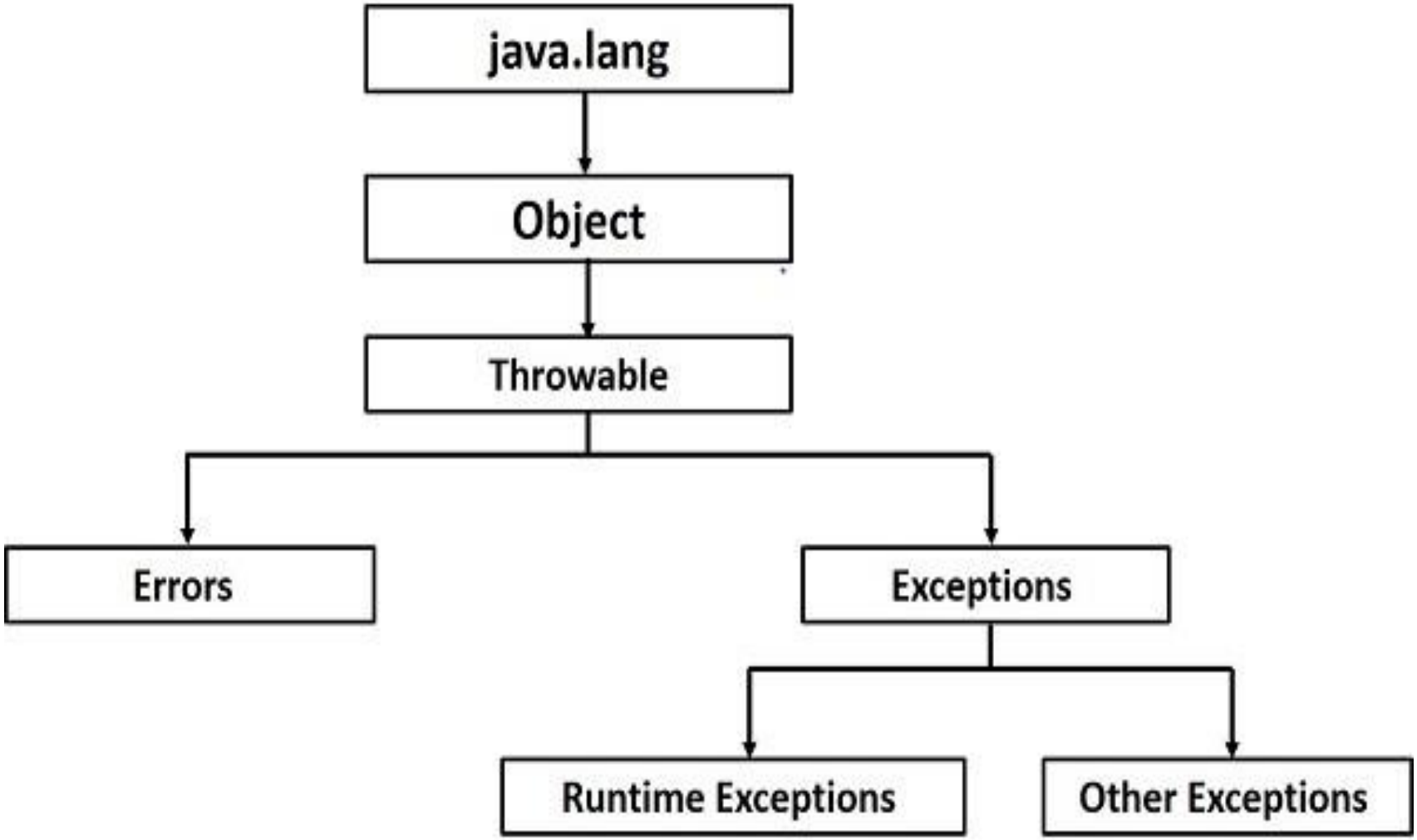
For example:

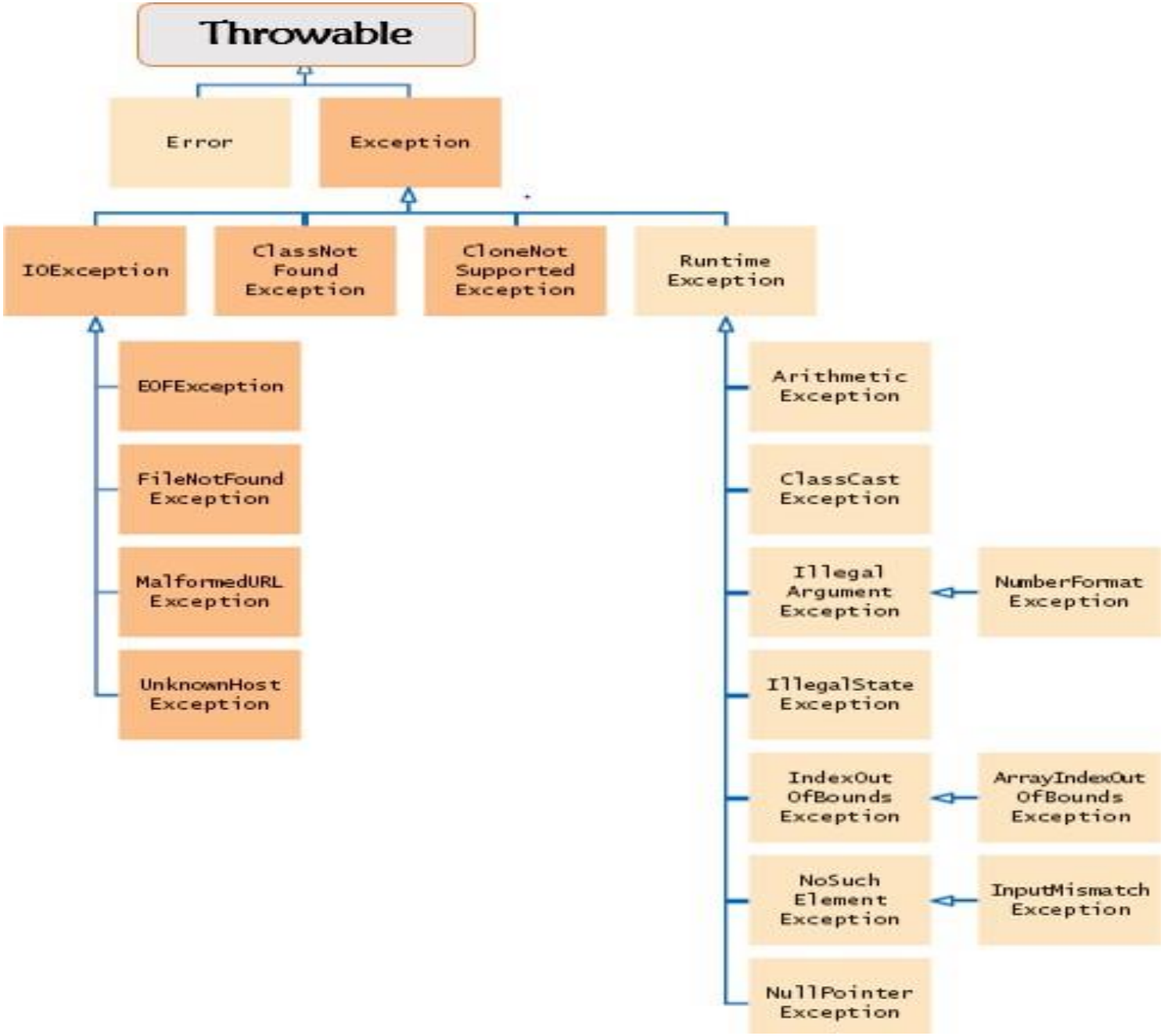
- if you have declared an array of size 4 in your program, and trying to call the 5th element of the array then an ArrayIndexOutOfBoundsExceptionexception occurs.

Cont'd …

- Errors :-
  - These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.
  - Errors are typically ignored in your code because you can rarely do anything about an error.
  - For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy




Throwable
Error
Exception
IOException
ClassNot Found Exception
CloneNot Supported Exception
Runtime Exception
EOFException
FileNotFound Exception
MalformedURL Exception
UnknownHost Exception
Arithmetic Exception
ClassCast Exception
Illegal Argument Exception
NumberFormat Exception
IllegalState Exception
IndexOut OfBounds Exception
ArrayIndexOut OfBounds Exception
NoSuch Element Exception
InputMismatch Exception
NullPointer Exception

Cont'd …

```
class ExceptionTest
{
public static void main (String k[ ])
    {
    int x=3,y=0;
    int a= x/y;        //denominator become zero
    System.out.println("a  =" + a);
 }
 }
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Exception.Program1.main(Program1.java:8)
```

Cont'd …

- An exception in Java is a signal that indicates the occurrence of some important or unexpected condition during execution.

Example:

- If exception object not handled properly by us, then the default handler handles it.
- The error handling code perform the following tasks.
  1. Find the problem (*Hit* the exception).
  2. Inform that an error has occurred (*Throw* the exception) .
  3. Received the error information (*Catch* the exception).
  4. Take corrective actions (*Handle* the exception).

Cont'd …

- The exception mechanism is built around the throw-and-catch paradigm.
- To throw an exception is to signal that an unexpected error condition has occurred.
- To catch an exception is to take appropriate action to deal with the exception.
- An exception is caught by an exception handler, and the exception need not be caught in the same context that it was thrown in.
- The runtime behavior of the program determines which exceptions are thrown and how they are caught. The throw-and-catch principle is embedded in the try-catch-finally construct.

Exception Handling Mechanisms

try-catch Block:

- Encapsulates code that might throw an exception.
- If an exception occurs, it's caught by the corresponding catch block.

```
try {
// Code that might throw an exception
// exception may be generated here
}
catch (ExceptionType name) {
// Code to handle the exception
}
```

Division by zero

```
Scanner scanner = new Scanner(System.in);
        System.out.print("Enter numerator: ");
        int numerator = scanner.nextInt();
        System.out.print("Enter denominator: ");
        int denominator = scanner.nextInt();

        try {
            int result = numerator / denominator;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Error: Cannot divide by zero.");
        }
```

Array index out of bounds

```
public static void main(String[] args) {
        int[] numbers = {10, 20, 30};
        try {
            System.out.println("Accessing index 5: " +
numbers[5]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Index out of bounds.");
        }
```

Invalid user input

```
public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.print("Enter an integer: ");
            int number = scanner.nextInt();
            System.out.println("You entered: " + number);
        } catch (InputMismatchException e) {
            System.out.println("Error: Invalid input. Please enter
an integer.");
        }
```

Multiple catch Clauses

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.
- In case of multiple catch statements exception subclasses must come before any of their superclasses.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed.

```
try {
// Code that may throw exceptions
} catch (ExceptionType1 e1) {
 // Handle ExceptionType1
} catch (ExceptionType2 e2) {
// Handle ExceptionType2
} catch (ExceptionType3 e3) {
// Handle ExceptionType3}
// ... additional catch blocks as needed
```

Example:

```
public class ExcepTest {
public static void main(String args[]) {
int a[] = new int[2];
try {
System.out.println("Access element three :" + a[3]);
}
catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Exception thrown   :" + e); }
finally {
        a[0] = 6;
System.out.println("First element value: " + a[0]);
System.out.println("The finally statement is executed");
} }}
```

Handling Multiple Exceptions

```
public static void main(String[] args) {
        try {
            int[] numbers = {10, 20, 30};
            int result = numbers[3] / 0; // May throw
ArrayIndexOutOfBoundsException or ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index is out of bounds: " +
e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic error: " +
e.getMessage());
        } catch (Exception e) {
            System.out.println("General exception: " +
e.getMessage());
```

Finally Block

- Each try statement must be followed by at least one catch or finally block.
- finally statement (block):- The finally will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- The finally block is used to execute the statements that must be executed in each and every condition like closing the opened files and freeing the resources.
- It may be add immediately after the try block or after the last catch block

```
try {
 // Code that may throw an exception
}
finally {
// Code that will always execute
}
```

```
try {
// Code that may throw an exception
} catch (ExceptionType e) {
 // Exception handling code
} finally {
// Code that will always execute}
```

```
// Demonstrate multiple catch statements.
class MultiCatch {
    public static void main(String args[]){
    try{
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[4] = 99;
        }catch(ArithmeticException e) {
            System.out.println( e.getMessage());
        }catch(ArrayIndexOutOfBoundsException e) {
          System.out.println( e.getMessage());
        }
            finally {
            System.out.println("finally blocks.");
            }}}
```

```
public static void main(String[] args) {
        try {
            String str = null;
            System.out.println(str.length());
        }
finally {
            System.out.println("Finally block executed.");
        }
        System.out.println("Rest of the code...");
    }
```

```
public static void main(String[] args) {
        try {
            int data = 25 / 0;
            System.out.println("Result: " + data);
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e);
        } finally {
            System.out.println("Finally block executed."); }
        System.out.println("Rest of the code...");
    }
```

- User defined exception are created by extending the Exception class and overriding the getMessage() method.

```
public class BalanceNotEnoughException extends Exception{
        public BalanceNotEnoughException(String msg) {
                super(msg);
                }
                public String getMessage() {
                return super.getMessage();
        }
}
```

- In java user can throw any exception manually by using *throw* keyword.

eg- throw new BalanceNotEnoughException("Amount is not enough");

```
void calc(int bal)throws BalanceNotEnoughException{
  if(bal<15000)
 throw new
   BalanceNotEnoughException("Balance"+bal+"not
   enough");
  // other operations…..
}
```

Throw statement

Used to explicitly throw an exception.

throw new IllegalArgumentException("Invalid argument");

```
public class AgeValidator {
public static void checkAge(int age) {
 if (age < 18) {
throw new IllegalArgumentException("Access denied - You must be at
least 18 years old.");        }
else {            System.out.println("Access granted - You are old
enough!");        }
 }
public static void main(String[] args) {
checkAge(15); // This will throw an exception
   }}
```

throws Keyword: Indicates that a method may throw one or more exceptions.
public void readFile() throws IOException {   // Method implementation}

Throwing checked exception

```
public class FileProcessor {
    public static void findFile() throws IOException {
        throw new IOException("File not found");
    }

    public static void main(String[] args) {
        try {
            findFile();
        } catch (IOException e) {
            System.out.println("Exception caught: " +
e.getMessage());
        }
```

```
public class CircleAreaCalculator {
    public static double calculateArea(double radius) {
        if (radius < 0.0) {
            throw new IllegalArgumentException("Radius
cannot be negative");
        }
        return Math.PI * radius * radius;
    }

    public static void main(String[] args) {
        double area = calculateArea(-5.0);
        System.out.println("Area: " + area);
    }
```

throws keyword

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that caller of the method can guard themselves against that exception.
- We do this by including a throws clause in the method's declaration.
- A throws clause lists the types of exceptions that a method might throw except Error or RuntimeExcception or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile time error will result.

eg.

```
void calc() throws BalanceNotEnoughException{…
        …..
        ….}
```

Example

```
import java.io.*;

public class className {
public void deposit(double amount) throws
RemoteException {
        // Method implementation
        throw new RemoteException();
    }   // Remainder of class definition
}
```

```
 public static void readFile(String fileName) throws
IOException {
        FileReader file = new FileReader(fileName);
        BufferedReader reader = new BufferedReader(file);
        String line = reader.readLine();
        System.out.println("First line: " + line);
        reader.close();
    }

    public static void main(String[] args) {
        try {
            readFile("example.txt");
        } catch (IOException e) {
            System.out.println("An error occurred: " +
e.getMessage());
```

Java Packages & API

- **A package in Java is used to group related classes.**
- **Think of it as a folder in a file directory.**
- **We use packages to avoid name conflicts, and to write a better maintainable code.**
- **Packages are divided into two categories:**
  - **Built-in Packages (packages from the Java API)**
  - **User-defined Packages (create your own packages)**

Built-in Packages

- The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.
- The library contains components for managing input, database programming, and much much more.
- The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.
- To use a class or a package from the library, you need to use the import keyword:

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Import a Class

- If you find a class you want to use, for example, the Scanner class, which is used to get user input, write the following code:

```
import java.util.Scanner;
```

- java.util is a package, while Scanner is a class of the java.util package.

Import a Package

- There are many packages to choose from.
- In the previous example, we used the Scanner class from the java.util package.
- This package also contains date and time facilities, random-number generator and other utility classes.

- To import a whole package, end the sentence with an asterisk sign (*).
- The following example will import ALL the classes in the java.util package:

```
import java.util.*;
```

User-defined Packages

- To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

- To create a package, use the package keyword:

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Thank You