CCS0023/L

Object Oriented Programming (Java)

Exception

08

LESSON #8

Exception

EXCEPTION

➢**Exception** denotes an abnormal event that occurs during the execution of the program and disrupts its normal flow.

CONCEPTS OF EXCEPTION

1. When an error occurs within a method, the method creates an object called ***exception object*** and hands it off to the runtime system.

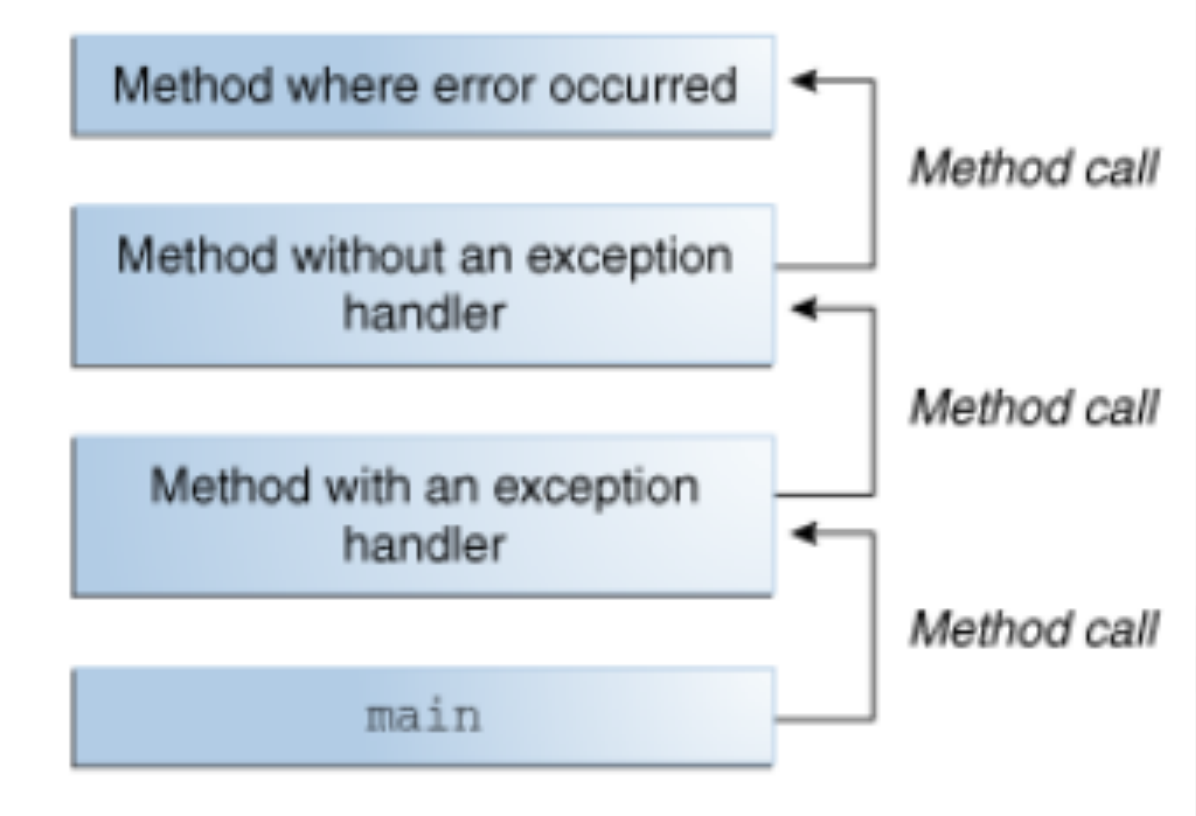
2. *Exception object* contains information about the error, including its type and the state of the program when the error occurred.

CONCEPTS OF EXCEPTION

3. Creating an *exception object* and handing it to the runtime system is called ***throwing an exception***.

4. After a method throws an exception, the runtime system attempts to find a list of methods, called the ***call stack*** to handle it.

The *call stack*



https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

CONCEPTS OF EXCEPTION

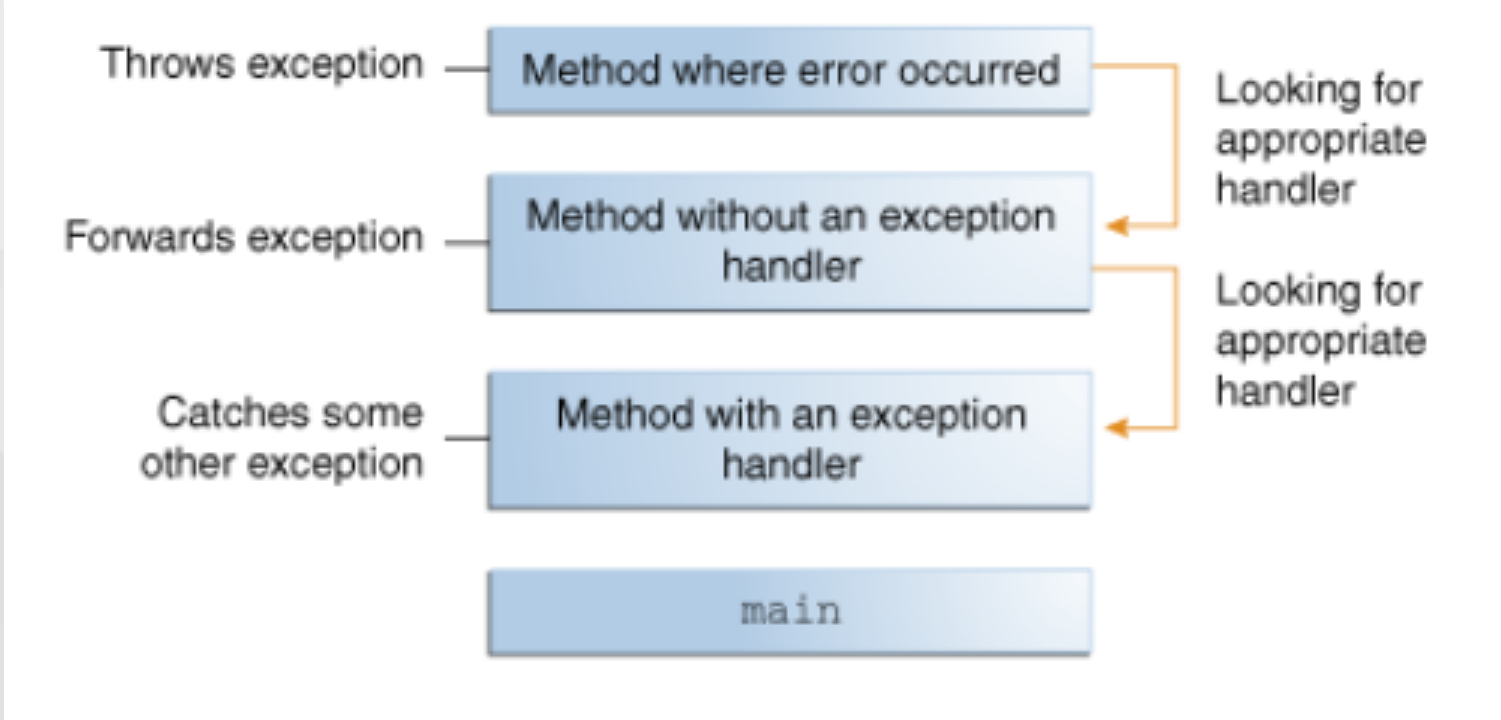
5. The runtime system searches the *call stack* for a method that contains a block of code that can handle the exception called ***exception handler***.

6. When an appropriate handler is found, the runtime system passes the exception to the handler. An *exception handler* is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

CONCEPTS OF EXCEPTION

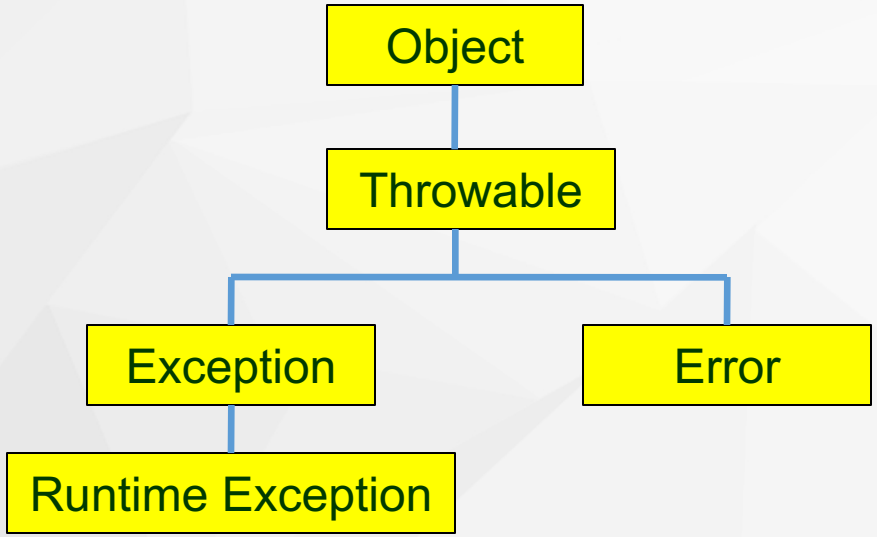
7. The exception handler chosen is said to ***catch the exception***. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.

Searching the *call stack* for the *exception handler*



https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Exception Class Hierarchy



- Exception

Exception-Handling Keywords

- try
- catch
- throw
- throws
- finally

Exception-Handling Techniques

- **Declare the exception**

  The programmer can declare that he is aware of the exception by including the clause *throws <name-of-exception>.* This is the specific requirement of the Java language.

Exception-Handling Techniques

- **Handle the exception**

  This is the catch requirement of the Java language. Handle the exception by enclosing the code in a *try-catch block*.

Exception-Handling Techniques

- **Re-throw the exception and declare this in the method signature**

  Use the *try-catch* block to catch the exception and re-throw the exception to another subclass of the exception. In this event, the programmer is passing the exception up the call stock.

The try / catch and finally Blocks

```
try{
    //codes that may throw exception
}
catch(ExceptionType e1){
  //codes that handle exception e1
}
catch(ExceptionType e2){
  //codes that handle exception e2
}
finally{
  /*codes to execute whether or not errors are
    encountered*/
}
```

Exception

Exceptions thrown during execution of the try block can be caught and handled in a **catch** block.
The code in the **finally** block is always executed.

Exception

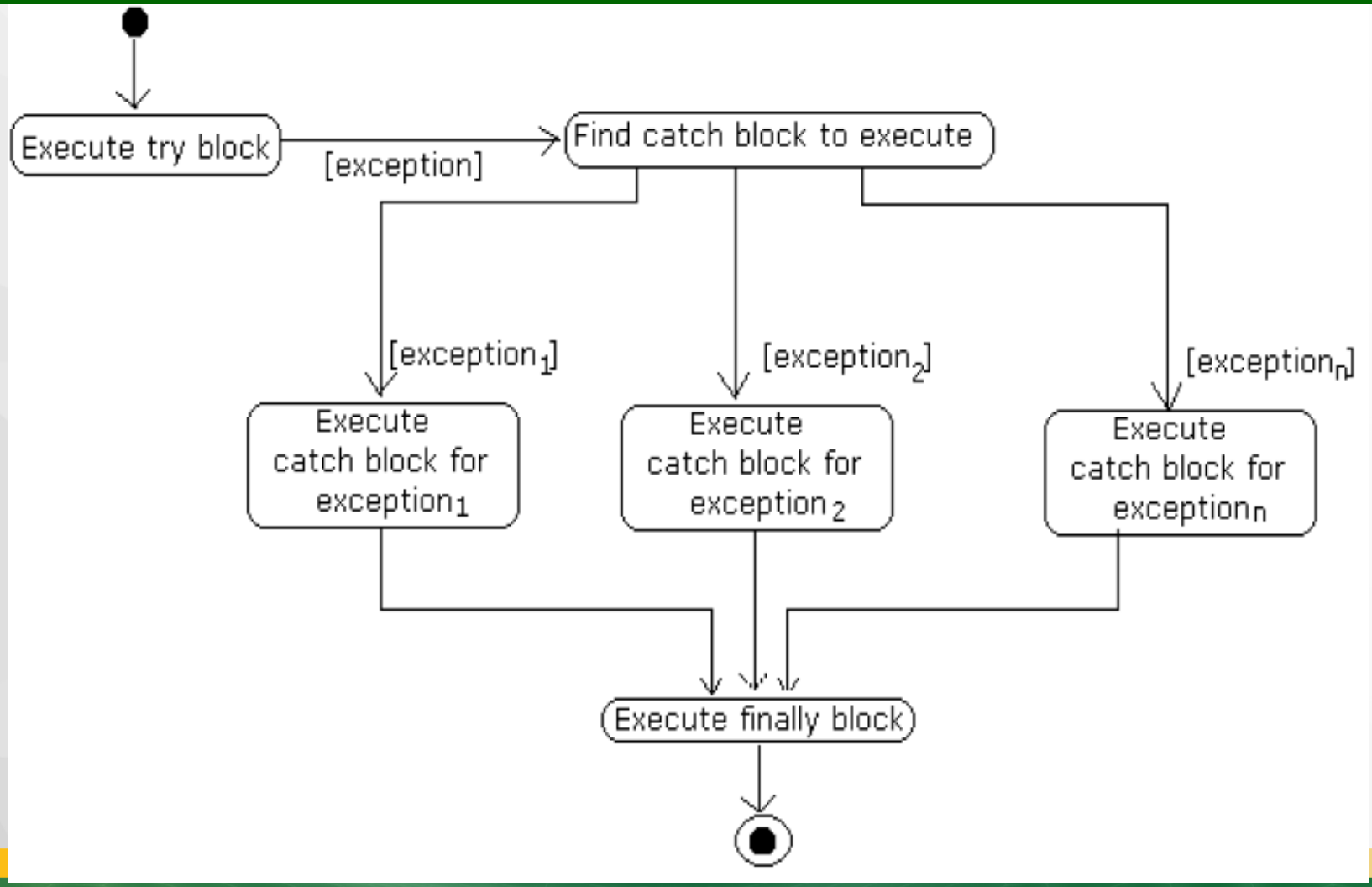
The following are the key aspects about the syntax of the try-catch-finally construct:

- The block notation is mandatory.
- For each try block, there can be one or more catch blocks, but only one finally block.
- The catch blocks and finally blocks must always appear in conjunction with the try block, and in the above order.

Exception

- A try block must be followed by at least one catch block OR one finally block, or both.
- Each catch block defines an exception handle. The header of the catch block takes exactly one argument, which is the exception its block is willing to handle. The exception must be of the **Throwable** class or one of its subclasses.

Exception



- Exception

The *throw* and *throws* Statements

```
public void methodName throws ExceptionList
{
   //statements
throw ThrowableInstance;
}
```

where:
ExceptionList – list of exception types separated by comma
ThrowableInstance – an instance (object) of an exception

Common Predefined Exceptions

- **ArithmeticException –** is thrown when an exceptional arithmetic condition occurred such as dividing integers by zero.

Common Predefined Exceptions

- **NullPointerException –** is thrown when an application attempts to use a null object that has not been initialized or instantiated.

Common Predefined Exceptions

- **ArrayIndexOutOfBoundsException –** is thrown when an attempt is made to access an element in the array that is beyond the index of the array.

User-defined Exceptions

Steps on how to create and use customized exceptions:

1. Create a class that extends from Exception.
2. In a method of another class, throw a new instance of the Exception.
3. Use the method that throws exception in a try-catch block.

Example

```
public class ExceptionExample
{
    public static void main( String[] args ){
    try{
        System.out.println( args[1] );
    }catch( ArrayIndexOutOfBoundsException exp ){
        System.out.println("Exception caught!");
     }
    }
}
```

Throwing Exceptions (throw)

```
class ThrowDemo {
    public static void main(String args[]){
        String input = “invalid input”;
        try {
        if (input.equals(“invalid input”)) {
                throw new RuntimeException("throw demo");
        } else {
                System.out.println(input);
        }
                System.out.println("After throwing");
        } catch (RuntimeException e) {
            System.out.println("Exception caught here.");
            System.out.println(e);
        }
    }
}
```

- Exception

Throwing Exceptions (throws)

```
public class ThrowingClass {
    static void myMethod() throws ClassNotFoundException {
        throw new ClassNotFoundException ("just a demo");
}
}


      class ThrowsDemo {
      public static void main(String args[]) {
          try {
              ThrowingClass.myMethod();
          } catch (ClassNotFoundException e) {
              System.out.println(e);
          }
          }
      }
```

- Exception

Creating your own Exception

```
class HateStringException extends RuntimeException{
/* No longer add any member or constructor */
}

class TestHateString {
      public static void main(String args[ ]) {
      String input = "invalid input";
      try {
      if (input.equals("invalid input")) {
      throw new HateStringException();
      }
      System.out.println("String accepted.");
      } catch (HateStringException e) {
      System.out.println("I hate this string: " + input +
      ".");
            } } }
```

ASSERTION

- An **assertion** is a statement in Java that enables you to test your assumptions about your program.
- Each assertion contains a boolean expression that you believe will be true when the assertion executes.

- By verifying that the boolean expression is indeed true, the assertion confirms your assumptions about the behavior of your program, increasing your confidence that the program is free of errors.

Types of Assertion:

- **Pre-condition** – an assertion which invokes when a method is invoked.
- **Post-condition** – an assertion which invokes after a method finishes.

The assertion statement has two forms:

The first is:

assert Expression1;

where: Expression1 = boolean expression.

When the system runs the assertion,
it evaluates Expression1 and if it is false
throws an AssertionError with no details.

ASSERTION

Enabling assertions onNetBeans

To enable runtime assertion checking in NetBeans IDE, simply follow these steps:

1. Look for the project name in the Projects panel on the left. Right click on the project name or icon.

2. Select Properties. This is the last option of the drop-down menu.

3. Click on Run from the Categories panel on the left.

4. Enter -ea for VM Options text field.

5. Click on OK button.

6. Compile and run the program as usual.

ASSERTION

Here is an example that uses the simple form of assert.

```
class AgeAssert {
    public static void main(String args[]) {
        int age = 0;
        assert(age>0);
        /* if age is valid (i.e., age>0) */
        if (age >= 18) {
        System.out.println("Congrats! You're an adult!
        =)");
        }
    }
}
```

ASSERTION

Here is an example that uses the simple form of assert.

```
class AgeAssert {
    public static void main(String args[]) {
        int age = 0;
        assert(age>0) : "age should at least be 1";
        /* if age is valid (i.e., age>0) */
        if (age >= 18) {
        System.out.println(“Congrats! You're an adult!
        =)”);
        }
    }
}
```

Complex Assertion Form

- Use the second version of the assert statement to provide a detailed message for the AssertionError.

- The system passes the value of Expression2 to the appropriate AssertionError constructor, which uses the string error message.

Complex Assertion Form

- The purpose of the message is to communicate the reason for the assertion failure
- Don’t use assertions to flag user errors.

When an Assertion Fails

- Assertion failures are labeled in stack trace with the file and line number from which they were thrown.

- Second form of the assertion statement should be used in preference to the first when the program has some additional information that might help diagnose the failure

When To Use Assertions

- Internal Invariants

  Many programmers use comments to indicate programming assumptions.

```
if (i % 3 == 0) { ... }
else if (i % 3 == 1) { ... }
else { // We know (i % 3 == 2)
... }
```

When To Use Assertions

Now we can use assertions to guarantee the behavior.

```
if (i % 3 == 0) { ... }
else if (i % 3 == 1) { ... }
else { assert i % 3 == 2 : i; ... }
```

When To Use Assertions

- Control Flow

If a program should never reach a point, then a constant false assertion may be used.

```
void foo() {
        for (...) {
            if (...)
            return;
        }
        assert false; // Execution should never get here
}
```

Example: AssertionExample.java

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

public class AssertionExample{
    public static void main( String args[] ){
         Scanner scanner = new Scanner( System.in );

         System.out.print( "Enter a number between 0 and 20: " );
         int value = scanner.nextInt();
         assert( value >= 0 && value <= 20 ) :
         "Invalid number: " + value;
         System.out.printf( "You have entered %d\n", value );
        }
    }
```

To run the example:

Compile the example with:

javac AssertionExample.java

Run the example with:

java -ea AssertionExample

To enable assertions at runtime,

-ea command-line option is used

THANK YOU