Catching Exceptions

v103

by Dave Slemon, Mohawk College

Java Program Error Types

- **Syntax Errors** - program won't compile until these errors are resolved

- **Logic Errors** (i.e. semantic errors) - program runs but output is not correct. (harder to track down these errors)

- **Run-time Errors** - errors that crash the program.
This is where CATCHING EXCEPTIONS comes in.

Checked and Unchecked Exceptions

In Java, exceptions are categorized into two types: **checked exceptions** and unchecked exceptions. The distinction lies in how these exceptions are handled by the Java compiler and at runtime.

Checked exceptions are exceptions that the Java compiler requires the developer to handle explicitly using try-catch blocks

or

declare them in the method signature using the "**throws**" keyword. If a method throws a **checked exception**, the calling code must either catch the exception or propagate it further up the call stack.

```
public static void pause(int fps) throws InterruptedException {
}
```

Checked Exception Example

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class CheckedExceptionExample {

  public static void main(String[] args) {
    try {
      readFromFile("nonexistent.txt");
    } catch (IOException e) {
      System.out.println("Error occurred while reading the file: " + e.getMessage());
    }
  }

  public static void readFromFile(String filename) throws IOException {
    BufferedReader reader = new BufferedReader(new FileReader(filename));
    String line;
    while ((line = reader.readLine()) != null) {
      System.out.println(line);
    }
    reader.close();
  }
}
```

Main program is dealing gracefully with this checked exception. (possible error: the file may not exist)

The readFromFile method throws the compulsory responsibility of dealing with a possible checked exception to the calling program, i.e. the main() program

Examples of **Checked** Exceptions

IOException: This exception is thrown when there is an error during input/output operations, such as reading or writing data to files or streams.

```
import java.io.*;
public class FileHandler {
  public void readFile(String fileName) throws IOException {
    BufferedReader reader = new BufferedReader(new FileReader(fileName));
    // ...
  }
}
```

InterruptedException: This exception is thrown when a thread is interrupted while it is waiting, sleeping, or otherwise occupied.

```
public class MyRunnable implements Runnable {
  public void run() {
    try {
      // Some time-consuming operation
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      // Handle the interruption
    }
  }
}
```

Examples of **Checked** Exceptions

ClassNotFoundException: This exception is thrown when attempting to load a class dynamically at runtime, but the class cannot be found.

```
public class ClassLoaderExample {
  public void loadClass(String className) throws ClassNotFoundException {
    Class<?> myClass = Class.forName(className);
    // ...
  }
}
```

Checked exceptions are used for scenarios where exceptional conditions can be anticipated and handled explicitly in the code, promoting more robust error handling and improving code reliability. When a method throws a checked exception, the calling code must either catch the exception or propagate it (declare it using throws in its method signature).

Unchecked Exceptions

Unchecked exceptions (also known as runtime exceptions) do not need to be explicitly handled using try-catch blocks or declared in the method signature. They can be caught if desired, but it is not mandatory. These exceptions usually represent programming errors or unexpected conditions that occur during the execution of a program.

Examples of **Unchecked** Exceptions

NullPointerException: Occurs when you attempt to perform an operation on an object reference that points to **null**.

```
String str = null;
int length = str.length(); // This will throw a NullPointerException.
```

Examples of **Unchecked** Exceptions

IllegalArgumentException: Occurs when an inappropriate argument is passed to a method.

```
public void printAge(int age) {
   if (age < 0) {
      throw new IllegalArgumentException("Age cannot be negative.");
   }
   System.out.println("Age: " + age);
}
```

NumberFormatException: Occurs when attempting to convert a string to a numeric type, but the string does not have the appropriate format.

```
String strNumber = "abc";
int number = Integer.parseInt(strNumber); // This will throw a NumberFormatException.
```

Examples of **Unchecked** Exceptions

UnsupportedOperationException: Occurs when an unsupported operation is called.

```
List<String> immutableList = List.of("A", "B", "C");
immutableList.add("D"); // This will throw an UnsupportedOperationException.
```

Remember that unchecked exceptions are usually caused by programming errors or unexpected conditions that may not be recoverable during runtime. Handling these exceptions properly in your code is essential to ensure robustness and graceful degradation in exceptional scenarios.

Examples of **Unchecked** Exceptions

ArrayIndexOutOfBoundsException: Occurs when you try to access an array element using an index that is outside the valid range of the array.

```
int[] numbers = {1, 2, 3};
int value = numbers[5]; // This will throw an ArrayIndexOutOfBoundsException.
```

ArithmeticException: Occurs when an arithmetic operation results in an error, such as division by zero.

```
int result = 10 / 0; // This will throw an ArithmeticException.
```

ClassCastException: Occurs when an attempt is made to cast an object to a type that it is not compatible with.

```
Object obj = "Hello";
Integer num = (Integer) obj; // This will throw a ClassCastException.
```

Study this method below, what do you observe?

1. The pause method might generate an InterruptedException, (i.e. a checked exception which MUST be dealt with) but instead of pause dealing with the error, it throws that responsibility to any calling program which calls pause.

```
/**
  * Pauses the program based on how many frames per second the user wants.
  * For example, if they want 20 frames per second, the pause time should be
  * 1000/20 = 50 ms.
  *
  * @param fps Number of frames per second
  * @throws java.lang.InterruptedException
  */
 public static void pause(int fps) throws InterruptedException {
    int pauseTime = 1000 / fps;
    Thread.sleep(pauseTime);
    System.out.println("Paused for " + pauseTime + " ms.");
 }
```

2. Division by 0 might occur, which is an unchecked ArithmeticException

```
public class Except1 {
   /**
    * Pauses the program based on how many frames per second the user wants.
    * For example, if they want 20 frames per second, the pause time should be
    * 1000/20 = 50 ms.
    *
    * @param fps Number of frames per second
    * @throws java.lang.InterruptedException
    */
   public static void pause(int fps) throws InterruptedException {
      int pauseTime = 1000 / fps;
      Thread.sleep(pauseTime);
      System.out.println("Paused for " + pauseTime + " ms.");
   }



   public static void main(String[] args) {
         Scanner sc = new Scanner(System.in);
         System.out.println("How many frames per second?");
         int framesPerSecond =  sc.nextInt();
         pause(framesPerSecond);
     }
   }
```

What do we need to do with this code?

```
public class Except1 {
  /**
   * Pauses the program based on how many frames per second the user wants.
   * For example, if they want 20 frames per second, the pause time should be
   * 1000/20 = 50 ms.
   * @param fps Number of frames per second
   * @throws java.lang.InterruptedException
   */
  public static void pause(int fps) throws InterruptedException {
     int pauseTime = 1000 / fps;
     Thread.sleep(pauseTime);
     System.out.println("Paused for " + pauseTime + " ms.");
  }
  public static void main(String[] args) {
       Scanner sc = new Scanner(System.in);
       System.out.println("How many frames per second?");
       int framesPerSecond =  sc.nextInt();
        try {
           pause(framesPerSecond);
        } catch (InterruptedException e) {
           System.out.println("Error occurred: " + e.getMessage( ) );
        }
     }
  }
```

```
public class Except1 {
   public static void pause(int fps) throws InterruptedException {
      int pauseTime = 1000 / fps;
      Thread.sleep(pauseTime);
      System.out.println("Paused for " + pauseTime + " ms.");
   }
  public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.println("How many frames per second?");
       try {
         int framesPerSecond = sc.nextInt();
         pause(framesPerSecond);
       }
       catch (InterruptedException e)  {
              System.out.println("Error occurred: " + e.getMessage( ) );    }
       catch (InputMismatchException  e )  {
              System.out.println("Error: please supply an integer");
              sc.next();     }
       catch (IllegalArgumentException  e)  {
             System.out.println("Error: must be a positive integer, please try again");   }
       catch (ArithmeticException e) {
             System.out.println("Error: integer must be > 0");   }
      catch (Exception e ) {
            System.out.println("Error: unknown, please try again ");  }
 }}
```

What 4 other unchecked exceptions might occur?

InputMismatchException
  Error: please supply an integer

IllegalArgumentException
  Error: must be a positive integer

ArithmeticException
  Error: integer must be > 0,

Exception
  Error: unknown, please try again

```
public class Except1 {
  public static void pause(int fps) throws InterruptedException {
     int pauseTime = 1000 / fps;
     Thread.sleep(pauseTime);
     System.out.println("Paused for " + pauseTime + " ms.");
  }
 public static void main(String[] args) {
     Scanner sc = new Scanner(System.in);
     System.out.println("How many frames per second?");
      try {
         int framesPerSecond = sc.nextInt();
         pause(framesPerSecond);
      }
      catch (InterruptedException e)  {
               System.out.println("Error occurred: " + e.getMessage( ) );    }
      catch (InputMismatchException  e )  {
               System.out.println("Error: please supply an integer");
               sc.next();     }
      catch (IllegalArgumentException  e)  {
              System.out.println("Error: must be a positive integer, please try again");   }
      catch (ArithmeticException e) {
              System.out.println("Error: integer must be > 0");   }
     catch (Exception e ) {
             System.out.println("Error: unknown, please try again ");  }
 }}
```

When a method throws a checked exception, any calling method (in this case, the main method) is aware of this possibility and must either catch the exception or declare that it may propagate it further.

In the main method, we are catching the unchecked ArithmeticException using the catch block, which allows us to handle the exception gracefully by displaying an error message to the user and giving them a chance to try again.

Keep in mind that if a method throws an unchecked exception (subclass of RuntimeException), you don't need to declare it in the throws clause.

Unchecked exceptions do not need to be caught or declared explicitly, which is different from checked exceptions.

Practice Test Question.

Write code to accept an integer input from the keyboard using a **Scanner** called **input** .

The input should be stored in a new variable called **number**. If the user enters something that is not an integer, the variable **number** should be assigned the value **-999**.

NB: uncheck exception: **InputMismatchException**

```
public class ExceptionDemo {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      int number ;
         try{
               System.out.print("Enter an integer: ");
               number = input.nextInt();

               System.out.println( "The number entered is " + number);

          }
            catch (InputMismatchException ex) {
               System.out.println("Try again. (" +
                    "Incorrect input: an integer is required)");
               number = -999;
               input.nextLine();   //to clear the line
             }
          input.nextLine();   //to clear the line

          }  //main
}
```

```
public class InputMismatchExceptionDemo {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      boolean continueInput = true;
      int number ;
     do {
                    try{
                       System.out.print("Enter an integer: ");
                       number = input.nextInt();

                       System.out.println( "The number entered is " + number);

                       continueInput = false;
                    }
                    catch (InputMismatchException ex) {
                       System.out.println("Try again. (" +
                             "Incorrect input: an integer is required)");
                       number = -999;
                       input.nextLine();   //to clear the line
                     }
             } while (continueInput);
             input.nextLine();   //to clear the line

             }   //main
}
```