Exception Handling and Recursion

CS102A Lecture 15

James YU
`yujq3@sustech.edu.cn`

Department of Computer Science and Engineering
Southern University of Science and Technology

Dec. 21, 2020

Objectives

- What exceptions are
- How exception handling works
- Exception class hierarchy
- Checked/unchecked exceptions
- Stack traces and chained exceptions
- Recursion

Exception

- An *exception* is an indication of a problem that occurs during a program’ s execution. It would disrupt the normal flow of instructions.

```java
public static void main(String[] args) {
  Scanner scanner = new Scanner(System.in);
  System.out.print("Enter an integer numerator: ");
  int numerator = scanner.nextInt();
  System.out.print("Enter an integer denominator: ");
  int denominator = scanner.nextInt();
  int result = quotient(numerator, denominator);
  System.out.printf("Result: %d / %d = %d\n", numerator, denominator,
       result);
  scanner.close();
}
public static int quotient(int numerator, int denominator) {
  return numerator / denominator;
}
```

Three executions of the program

```
Enter an integer numerator: 3
Enter an integer denominator: 2
Result: 3 / 2 = 1
```

```
Enter an integer numerator: 3
Enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ExceptionExample.quotient(ExceptionExample.java:14)
    at ExceptionExample.main(ExceptionExample.java:10)
```

- An execution where the “/ by zero” exception is thrown and the program terminates.

Three executions of the program

```
Enter an integer numerator: 3
Enter an integer denominator: a
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Unknown Source)
    at java.util.Scanner.next(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at java.util.Scanner.nextInt(Unknown Source)
    at ExceptionExample.main(ExceptionExample.java:9)
```

- An execution where the “InputMismatch” exception is thrown and the program terminates.

Exception

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ExceptionExample.quotient(ExceptionExample.java:14)
    at ExceptionExample.main(ExceptionExample.java:10)
```

- `java.lang.ArithmeticException`: The name of the exception.
- Followed by stack trace: the method call stack when the exception occurs.
    - Stack trace contains the path of execution that led to the exception.

Exception

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at ExceptionExample.quotient(ExceptionExample.java:14)
    at ExceptionExample.main(ExceptionExample.java:10)
```

```
 4 public static void main(String[] args) {
 5   Scanner scanner = new Scanner(System.in);
 6   System.out.print("Enter an integer numerator: ");
 7   int numerator = scanner.nextInt();
 8   System.out.print("Enter an integer denominator: ");
 9   int denominator = scanner.nextInt();
10   int result = quotient(numerator, denominator);
11   ...
12 }
13 public static int quotient(int numerator, int denominator) {
14   return numerator / denominator;
15 }
```

Exception handling

- An exception would disrupt program execution flows (for example, causing crashes).
- *Exception handling* is a nice feature of the Java language that can help you write **robust and fault-tolerant programs**.
- With exception handling, a program can **continue executing (rather than terminating)** after dealing with a problem. It is very useful in mission-critical or business-critical computing.

try-catch statement syntax

```
try {
  // code that might throw an exception
} catch( ExceptionType1 e1 ) {
  // code that handles type1 exception
} catch( ExceptionType2 e2 ) {
  // code that handles type2 exception
} catch( ExceptionType3 e3 ) {
  // code that handles type3 exception
} ...
```

- Exception parameter `e1` is a local variable in the catch block.
- At least one `catch` block or a `finally` block must immediately follow the `try` block ("immediately" means no content in between).

Handling the two exceptions

```java
public static void main(String[] args) {
  Scanner scanner = new Scanner(System.in);
  boolean continueLoop = true;
  do {
    try {
      System.out.print("Enter an integer numerator: ");
      int numerator = scanner.nextInt();
      System.out.print("Enter an integer denominator: ");
      int denominator = scanner.nextInt();
      int result = quotient(numerator, denominator);
      System.out.printf("Result: %d / %d = %d\n", numerator,
           denominator, result);
      scanner.close();
```

- Enclose the code that might throw an exception in a `try` block.

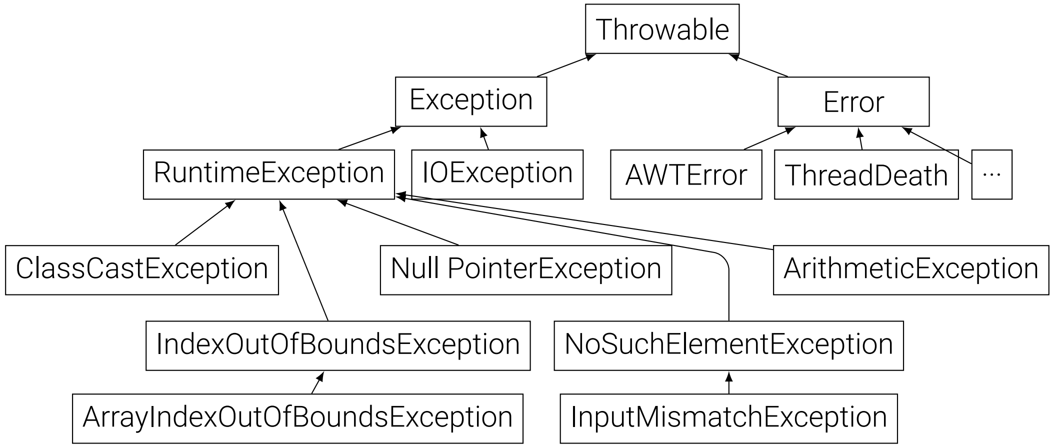
Handling the two exceptions

```
            continueLoop = false;
        } catch(InputMismatchException inputMismatchException) {
            System.err.printf("Exception: %s\n", inputMismatchException);
            scanner.nextLine(); // discard input so user can try again
        } catch(ArithmeticException arithmeticException) {
            System.err.printf("Exception: %s\n", arithmeticException);
        }
    } while(continueLoop);
}
```

- Each `catch` block (exception handler) handles a certain type of exception.
- The type is specified in the exception parameter.

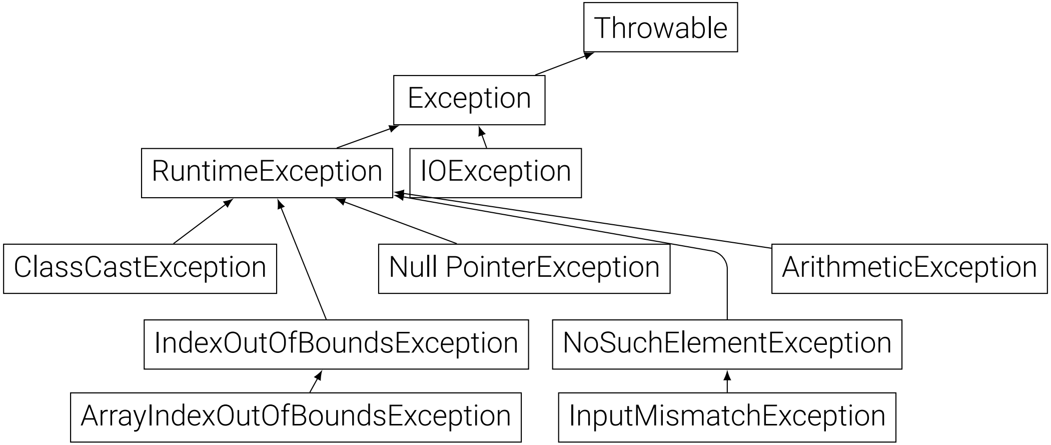
Java exception hierarchy

- In Java, only `Throwable` objects can be used with the exception-handling mechanism.



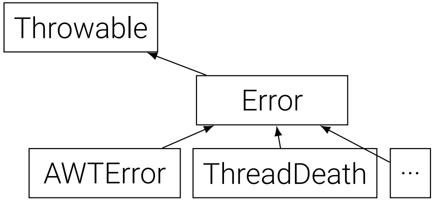
Java exception hierarchy

- `Exception` (in `java.lang`) and its subclasses represent exceptional situations that can occur in a Java program and should be caught by applications.



Java exception hierarchy

- `Error` and its subclasses represent abnormal situations that happen in the JVM (e.g., JVM out of memory) and should not be caught by applications. It’ s usually impossible for applications to recover from `Error`s.



Checked vs. unchecked exceptions

- Java distinguishes between *checked exception*s and *unchecked exception*s.
- All exception types that are direct or indirect subclasses of the class `RuntimeException` are unchecked exceptions.
- Unchecked exceptions are typically caused by defects in your program's code. Examples include `ArithmeticException`, `InputMismatchException`, `NullPointerException`.

Checked vs. unchecked exceptions

- All exception types that inherit from the class `Exception` but not `RuntimeException` are checked exceptions.
- Checked exceptions are typically **caused by conditions that are not under the control of the program**. For example, in file processing, the program can't open a file because the file does not exist.
- Unlike unchecked exceptions, checked exceptions cannot be ignored at the time of compilation (must be taken care of by programmers). Java compiler **enforces a catch-or-declare requirement** for checked exceptions.

Example: Unchecked exceptions

```
public static void main(String[] args) {
  Scanner scanner = new Scanner(System.in);
  System.out.print("Enter an integer numerator: ");
  int numerator = scanner.nextInt(); // potential runtime exception
  System.out.print("Enter an integer denominator: ");
  int denominator = scanner.nextInt(); // potential runtime exception
  int result = quotient(numerator, denominator);
  System.out.printf("Result: %d / %d = %d\n", numerator, denominator,
      result);
  scanner.close();
}
public static int quotient(int numerator, int denominator) {
  return numerator / denominator; // potential runtime exception
}
```

- It's fine if programmers do not take care of unchecked exceptions in the code.

Example: Checked exceptions

```java
public static void main(String[] args) {
    File f = new File("test.txt");
    FileReader reader = new FileReader(f);
    reader.close();
}
```

```
Unhandled exception type FileNotFoundException
Unhandled exception type IOException
```

- Fix: `catch` or *declare*.

The catch solution

```java
public static void main(String[] args) {
  try {
    File f = new File("test.txt");
    FileReader reader = new FileReader(f);
    reader.close();
  } catch(FileNotFoundException e) {
    // handle file not found exception
  } catch(IOException e) {
    // handle IO exception
  }
}
```

The *declare* solution

```java
public static void main(String[] args)
    throws FileNotFoundException, IOException {
  File f = new File("test.txt");
  FileReader reader = new FileReader(f);
  reader.close();
}
```

- The `throws` clause declares the exceptions that might be thrown when the method is executed and **let the callers handle the exceptions**.

More on unchecked exceptions

```
import java.lang.RuntimeException;
public static void main(String[] args) { ... }
public static int quotient(int numerator, int denominator) throws
    RuntimeException {
  return numerator / denominator; // potential runtime exception
}
```

- Still can compile as `quotient()` throws unchecked exception.

More on unchecked exceptions

```java
import java.lang.RuntimeException;
public static void main(String[] args) { ... }
public static int quotient(int numerator, int denominator) throws
    FileNotFoundException {
  return numerator / denominator; // potential runtime exception
}
```

- Fail to compile as `quotient()` throws checked exception. The caller must try and catch it.

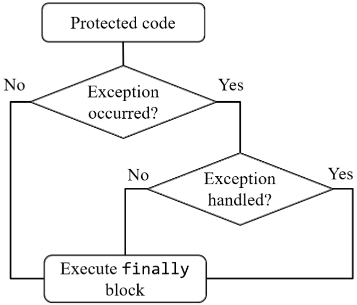
Catching subclass-type exceptions

- If a catch handler is written to catch superclass-type exceptions, it can also catch subclass-type exceptions. This makes the exception handling code concise (when the handling behavior is the same for all types) and allows for polymorphic processing of exception objects.

```java
public static void main(String[] args) {
  try {
    File f = new File("test.txt");
    FileReader reader = new FileReader(f);
    reader.close();
} catch(Exception e) {
    // catch and handle multiple types of exceptions
  }
}
```

finally block

```
try {
  // protected code
}
// catch blocks (optional)
finally {
  // always execute
  //when try block exits
}
```



- finally is useful for more than just exception handling. It prevents cleanup code from being accidentally bypassed by statements like return. Putting cleanup code in a finally block is a good practice, even when no exceptions are anticipated.

`finally` block

- In the example below, the `finally` block ensures that the used resource is closed regardless of whether the `try` statement completes normally or abruptly.

```
public String readFirstLineFromFile(String path) throws IOException {
  BufferedReader br;
  try {
    br = new BufferedReader(new FileReader(path));
    return br.readLine();
  } finally {
    if (br != null)
    br.close();
  }
}
```

Common methods of exceptions

- `printStackTrace`: output the stack trace to the standard error stream.
- Standard output stream (`System.out`) and standard error stream (`System.err`) are sequences of bytes. The former displays a program's output in the command prompt and the latter displays errors.
- Using two streams helps separate error messages from other output.

Common methods of exceptions

```java
public class CheckedExceptionExample {
  public static void main(String[] args) {
    try {
      File f = new File("not-exist.txt");
      FileReader reader = new FileReader(f);
      reader.close();
    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

```
java.io.FileNotFoundException: not-exist.txt
    at java.io.FileInputStream.open0(Native Method)
    at java.io.FileInputStream.open(Unknown Source)
    at java.io.FileInputStream.<init>(Unknown Source)
    at java.io.FileReader.<init>(Unknown Source)
    at CheckedExceptionExample.main(CheckedExceptionExample.java:5)
```

Common methods of exceptions

```java
public class CheckedExceptionExample {
  public static void main(String[] args) {
    try {
      method1();
    } catch (Exception e) {
      StackTraceElement[] traceElements = e.getStackTrace();
      for(StackTraceElement element : traceElements) {
        System.out.printf("%s\t%s\t%s\n", element.getFileName(), element.
            getLineNumber(), element.getMethodName());
      }
    }
  }
  public static void method1() throws Exception {
    throw new Exception("Exception thrown in method1");
  }
}
```

Common methods of exceptions

```java
public class CheckedExceptionExample {
  public static void main(String[] args) {
    try {
      method1();
    } catch (Exception e) {
      System.err.println(e.getMessage());
    }
  }
  public static void method1() throws Exception {
    throw new Exception("Exception thrown in method1");
  }
}
```

```
Exception thrown in method1
```

Chained exceptions

```java
public class CheckedExceptionExample {
  public static void main(String[] args) throws Exception {
    try {
      method1();
    } catch (Exception e) {
      throw new Exception("Exception thrown in main", e);
    }
  }
  public static void method1() throws Exception {
    throw new Exception("Exception thrown in method1");
  }
}
```

```
Exception in thread "main" java.lang.Exception: Exception thrown in main
    at CheckedExceptionExample.main(CheckedExceptionExample.java:6)
Caused by: java.lang.Exception: Exception thrown in method1
    at CheckedExceptionExample.method1(CheckedExceptionExample.java:10)
    at CheckedExceptionExample.main(CheckedExceptionExample.java:4)
```

User-defined exceptions (checked)

```java
public class MyException extends Exception {
  public MyException(String s) {
    super(s);
  }
}
```

```java
public class UserDefinedExceptionDemo {
  public static void main(String args[]) {
    try {
      throw new MyException("User-defined exception");
    } catch (MyException e) {
      System.err.println(e.getMessage());
    }
  }
}
```

User-defined exceptions (unchecked)

```
public class MyExceptions extends RuntimeException {
  public MyException2(String s) {
    super(s);
  }
}
```

```
public class UserDefinedExceptionDemo2 {
  public static void main(String args[]) {
    throw new MyException2("User-defined exception");
  }
}
```

```
Exception in thread "main" MyException2: User-defined exception
    at UserDefinedExceptionDemo2.main(UserDefinedExceptionDemo2.java:3)
```

Assertions

- When implementing and debugging a class, it’s sometimes useful to state conditions that should be true at a particular point in a method.
- These conditions, called *assertions*, help ensure a program’s correctness by catching potential bugs (such as logic errors) during development.
- Java has two versions of assert statements

```
assert expression; // throws an AssertionError if expression is false
assert expression1 : expression2; // throws an AssertionError with
    expression2 as the error message if expression1 is false
```

Assertions

```java
public class AssertionExample {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a number between 0 and 10: ");
    int number = input.nextInt();
    assert (number >=0 && number <= 10) : "bad number: " + number;
  }
}
```

- You must explicitly enable assertions: `java -ea AssertionExample`

```
Enter a number between 0 and 10: 12
Exception in thread "main" java.lang.AssertionError: bad number: 12
    at AssertionExample.main(AssertionExample.java:8)
```

Recursion

- Consider the *factorial* of a positive integer n, written $n!$.
  - $n! = n \cdot (n-1) \cdot (n-2) \cdots 1$
- One way of computing $n!$ is using `for` loop:

```
factorial = 1;
for (int counter = number; counter >= 1; counter--)
  factorial *= counter;
```

- The following relationship can be observed:
  - $n! = n \cdot (n-1)!$
- We can implement the same problem by *recursion*.

Recursion

```java
public class FactorialCalculator {
  // recursive method factorial (assumes its parameter is >= 0)
  public static long factorial(long number) {
    if (number <= 1) // test for base case
      return 1; // base cases: 0! = 1 and 1! = 1
    else // recursion step
      return number * factorial(number - 1);
  }

  // output factorials for values 0-21
  public static void main(String[] args) {
    // calculate the factorials of 0 through 21
    for (int counter = 0; counter <= 21; counter++)
      System.out.printf("%d! = %d%n", counter, factorial(counter));
  }
} // end class FactorialCalculator
```

Recursion

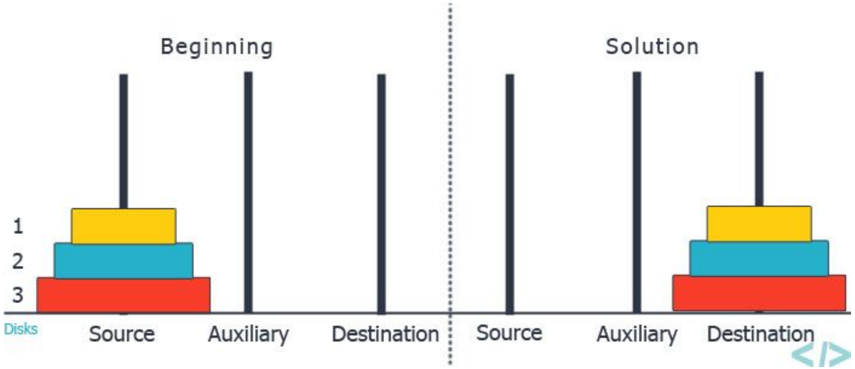
```
...
12! = 479001600
...
20! = 2432902008176640000
21! = -4249290049419214848
```

- $12!$ causes overflow for `int` variables.
- $21!$ causes overflow for `long` variables.

Recursion vs. iteration

- Both iteration and recursion are based on a control statement.
  - Iteration uses a repetition statement (e.g., `for`, `while` or `do-while`).
  - Recursion uses a selection statement (e.g., `if`, `if-else` or `switch`).
- Both iteration and recursion involve repetition.
  - Iteration explicitly uses a repetition statement.
  - Recursion achieves repetition through repeated method calls.
- Iteration and recursion each involve a termination test.
  - Iteration terminates when the loop-continuation condition fails.
  - Recursion terminates when a base case is reached.
- Both iteration and recursion can occur infinitely.
  - An infinite loop occurs with iteration if the loop continuation test never becomes `false`.
  - An infinite recursion occurs if the recursion step does not reduce the problem each time in a manner that converges on the base case, or if the base case is not tested.

Case study: Towers of Hanoi



Case study: Towers of Hanoi

- Moving n disks can be viewed in terms of moving only $n-1$ disks (hence the recursion) as follows:
  1. Move $n-1$ disks from peg 1 to peg 2, using peg 3 as a temporary holding area.
  2. Move the last disk (the largest) from peg 1 to peg 3.
  3. Move $n-1$ disks from peg 2 to peg 3, using peg 1 as a temporary holding area.
- The process ends when the last task involves moving $n=1$ disk (i.e., the base case). This task is accomplished by moving the disk, without using a temporary holding area.

Case study: Towers of Hanoi

```java
public class TowersOfHanoi {
  // recursively move disks between towers
  public static void solveTowers(int disks, int sourcePeg, int
      destinationPeg, int tempPeg) {
    // base case -- only one disk to move
    if (disks == 1) {
      System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);
      return;
    }
    // recursion step -- move (disk - 1) disks from sourcePeg
    // to tempPeg using destinationPeg
    solveTowers(disks - 1, sourcePeg, tempPeg, destinationPeg);
    // move last disk from sourcePeg to destinationPeg
    System.out.printf("%n%d --> %d", sourcePeg, destinationPeg);
    // move (disks - 1) disks from tempPeg to destinationPeg
    solveTowers(disks - 1, tempPeg, destinationPeg, sourcePeg);
  }
```

Case study: Towers of Hanoi

```java
  public static void main(String[] args) {
    int startPeg = 1; // value 1 used to indicate startPeg in output
    int endPeg = 3; // value 3 used to indicate endPeg in output
    int tempPeg = 2; // value 2 used to indicate tempPeg in output
    int totalDisks = 3; // number of disks
    // initial nonrecursive call: move all disks.
    solveTowers(totalDisks, startPeg, endPeg, tempPeg);
  }
} // end class TowersOfHanoi
```

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```