NUS
National University of Singapore
School of Computing

# CS1020 Data Structures and Algorithms I
Lecture Note #8

## Exceptions
Handling exceptional events

## Objectives

- Understand how to use the mechanism of **exceptions** to handle errors or exceptional events that occur during program execution

# References

Book

- Chapter 1, Section 1.6, pages 64 to 72

CS1020 website → Resources → Lectures

- http://www.comp.nus.edu.sg/~cs1020/2_resources/lectures.html

## Outline

1. Motivation
2. Exception Indication
3. Exception Handling
4. Execution Flow
5. Checked vs Unchecked Exceptions
6. Defining New Exception Classes

## 1. Motivation (1/4)

- Three types of errors
- Syntax errors ⇦ *Easiest to detect and correct*
  - Occurs when the rule of the language is violated
  - Detected by compiler
- Run-time errors
  - Occurs when the computer detects an operation that cannot be carried out (eg: division by zero; x/y is syntactically correct, but if y is zero at run-time a run-time error will occur)
- Logic errors ⇦ *Hardest to detect and correct*
  - Occurs when a program does not perform the intended task

## 1. Motivation (2/4)

Example.java

```
import java.util.Scanner;

public class Example {

   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter an integer: ");
      int num = sc.nextInt();
      System.out.println("num = " + num);
   }
}
```

← If error occurs here

← The rest of the code is skipped and program is terminated.

```
Enter an integer: abc
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:909)
        at java.util.Scanner.next(Scanner.java:1530)
        at java.util.Scanner.nextInt(Scanner.java:2160)
        at java.util.Scanner.nextInt(Scanner.java:2119)
        at Example1.main(Example1.java:8)
```

## 1. Motivation (3/4)

- Consider the factorial() method:
  - What if the caller supplies a negative parameter?

```
public static int factorial(int n) {
    int ans = 1;
    for (int i = 2; i <= n; i++) ans *= i;
    return ans;
}
```

What if *n* is negative?

- Should we terminate the program?

```
public static int factorial(int n) {
    if (n < 0) {
        System.out.println("n is negative");
        System.exit(1);
    }
    //Other code not changed
}
```

**`System.exit(`*`n`*`)`** terminates the program with exit code *n*. In UNIX, you can check the exit code immediately after the program is terminated, with this command: `echo $?`

- Note that factorial() method can be used by other programs
  - Hence, difficult to cater to all possible scenarios

## 1. Motivation (4/4)

- Instead of deciding how to deal with an error, Java provides the **exception** mechanism:
  1. Indicate an error (exception event) has occurred
  2. Let the user decide how to handle the problem in a separate section of code specific for that purpose
  3. Crash the program if the error is not handled
- Exception mechanism consists of two components:
  - Exception indication
  - Exception handling
- Note that the preceding example of using exception for ($n$ < 0) is solely illustrative. Exceptions are more appropriate for harder to check cases such as when the value of $n$ is too big, causing overflow in computation.

## 2. Exception Indication: Syntax (1/2)

- To indicate an error is detected:
  - Also known as **throwing an exception**
  - This allows the user to detect and handle the error

SYNTAX

```
throw ExceptionObject;
```

- Exception object must be:
  - An object of a class derived from **class Throwable**
  - Contain useful information about the error
- There are a number of useful predefined exception classes:
  - `ArithmeticException`
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `IllegalArgumentException`

## 2. Exception Indication: Syntax (2/2)

- The different exception classes are used to **categorize the type of error**:
  - There is no major difference in the available methods

| **Constructor** | |
|---|---|
| | ***`ExceptionClassName`*** **`(String Msg)`**<br>Construct an exception object with the error message Msg |
| **Common methods for Exception classes** | |
| `String` | **`getMessage()`**<br>Return the massage stored in the object |
| `void` | **`printStackTrace()`**<br>Print the calling stack |

## 2. Exception Handling: Example #1 (1/2)

ExampleImproved.java

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

public class ExampleImproved {

   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      boolean isError = false;
      do {
         System.out.print("Enter an integer: ");
         try {
            int num = sc.nextInt();
            System.out.println("num = " + num);
            isError = false;
         }
         catch (InputMismatchException e) {
            System.out.print("Incorrect input: integer required. ");
            sc.nextLine(); // skip newline
            isError = true;
         }
      } while (isError);
   }
}
```

## 2. Exception Handling: Example #1 (2/2)

```
    do {
       System.out.print("Enter an integer: ");
       try {
          int num = sc.nextInt();
          System.out.println("num = " + num);
          isError = false;
       }
       catch (InputMismatchException e) {
          System.out.print("Incorrect input: integer required. ");
          sc.nextLine(); // skip newline
          isError = true;
       }
    } while (isError);
```

```
Enter an integer: abc
Incorrect input: integer required. Enter an integer: def
Incorrect input: integer required. Enter an integer: 1.23
Incorrect input: integer required. Enter an integer: 92
num = 92
```

## 2. Exception Indication: Example #2

```java
public static int factorial(int n)
   throws IllegalArgumentException {

   if (n < 0) {
      IllegalArgumentException exObj
            = new IllegalArgumentException(n + " is invalid!");
      throw exObj;
   }

   int ans = 1;
   for (int i = 2; i <= n; i++)
      ans *= i;
   return ans;
}
```

This declares that method factorial() *may* throw IllegalArgumentException

Actual act of throwing an exception (Note: ‘throw’ and not ‘throws’ ). These 2 statements can be shortened to:

```java
throw new
   IllegalArgumentException(n + " is invalid!");
```

- Note:
  - A method can throw more than one type of exception

## 3. Exception Handling: Syntax

- As the user of a method that can throw exception(s):
  - It is your responsibility to handle the exception(s)
  - Also known as **exception catching**

```
try {
    statement(s);
}
```

// try block
// exceptions might be thrown
// followed by one or more catch block

```
catch (ExpClass1 obj1) {
    statement(s);
}
catch (ExpClass2 obj2) {
    statement(s);
}
```

// a catch block
// Do something about the exception
// catch block for another type of
exception

```
finally {
    statement(s);
}
```

// finally block – for cleanup code

## 3. Exception Handling: Example

```
public class TestException {

  public static int factorial(int n)
    throws IllegalArgumentException { //code not shown }

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter n: ");
    int input = sc.nextInt();

    try {
      System.out.println("Ans = " + factorial(input));
    }
    catch (IllegalArgumentException expObj) {
      System.out.println(expObj.getMessage());
    }
  }
}
```

We choose to print out the error message in this case. There are other ways to handle this error. See next slide for more complete code.

# 4. Execution Flow (1/2)

TestException.java

```
public static int factorial(int n)
    throws IllegalArgumentException {
        System.out.println("Before Checking");
        if (n < 0) {
            throw new IllegalArgumentException(n + " is invalid!");
        }
        System.out.println("After Checking");
        //... other code not shown
}
```

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter n: ");
    int input = sc.nextInt();
    try {
        System.out.println("Before factorial()");
        System.out.println("Ans = " + factorial(input));
        System.out.println("After factorial()");
    } catch (IllegalArgumentException expObj) {
        System.out.println("In Catch Block");
        System.out.println(expObj.getMessage());
    } finally {
        System.out.println("Finally!");
    }
}
```

```
Enter n: 4
Before factorial()
Before Checking
After Checking
Ans = 24
After factorial()
Finally!
```

```
Enter n: -2
Before factorial()
Before Checking
In Catch Block
-2 is invalid!
Finally!
```

## 4. Execution Flow (2/2)

- Another version
  - Keep retrying if n < 0

TestExceptionRetry.java

```java
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int input;
    boolean retry = true;
    do {
        try {
            System.out.print("Enter n: ");
            input = sc.nextInt();
            System.out.println("Ans = " + factorial(input));
            retry = false; // no need to retry
        } catch (IllegalArgumentException expObj) {
            System.out.println(expObj.getMessage());
        }
    } while (retry);
}
```

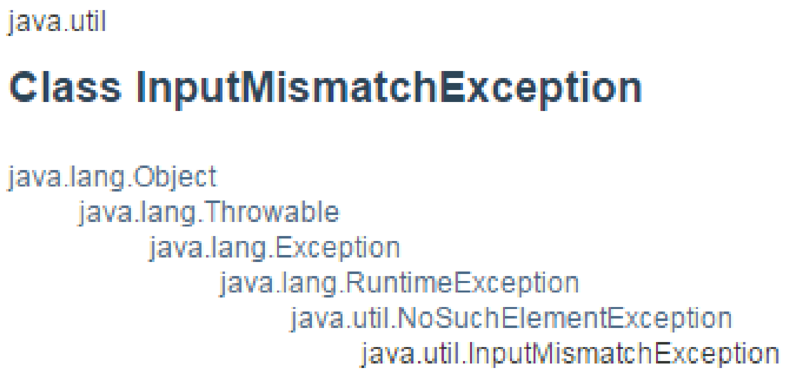
```
Enter n: -2
-2 is invalid!
Enter n: -7
-7 is invalid!
Enter n: 6
Ans = 720
```

## 5. Checked vs Unchecked Exceptions (1/2)

- **Checked exceptions** are those that require handling during compile time, or a compilation error will occur.
- **Unchecked exceptions** are those whose handling is not verified during compile time.
  - RuntimeException, Error and their subclasses are unchecked exceptions.
  - In general, unchecked exceptions are due to programming errors that are not recoverable, like accessing a null object (NullPointerException), accessing an array element outside the array bound (IndexOutOfBoundsException), etc.
  - As unchecked exceptions can occur anywhere, and to avoid overuse of try-catch blocks, Java does not mandate that unchecked exceptions must be handled.

## 5. Checked vs Unchecked Exceptions (2/2)

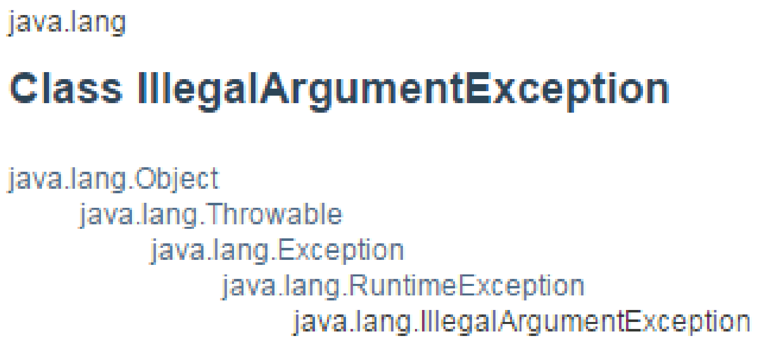
- InputMismatchException and IllegalArgumentException are subclasses of RuntimeException, and hence they are unchecked exceptions. (Ref: ExampleImproved.java and TestException.java)



java.util

**Class InputMismatchException**

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.util.NoSuchElementException
                    java.util.InputMismatchException
```



java.lang

**Class IllegalArgumentException**

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
```

## 6. Defining New Exception Classes

- New exception classes can be defined by deriving from class Exception:

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

- The new exception class can then be used in `throw` statements and `catch` blocks:

```
throw new MyException("MyException: Some reasons");
```

```
try {
    ...
} catch (MyException e) {
    ...
}
```

## 6. Example: Bank Account (1/5)

```
public class NotEnoughFundException extends Exception {

   private double amount;

   public NotEnoughFundException(String s, double amount) {
      super(s);
      this.amount = amount;
   }

   public double getAmount() {
      return amount;
   }
}
```

NotEnoughFundException.java

## 6. Example: Bank Account (2/5)

BankAcct.java

```
class BankAcct {

   private int acctNum;
   private double balance;

   public BankAcct() {
      // By default, numeric attributes are initialised to 0
   }

   public BankAcct(int aNum, double bal) {
      acctNum = aNum;
      balance = bal;
   }

   public int getAcctNum() {
      return acctNum;
   }

   public double getBalance() {
      return balance;
   }
```

## 6. Example: Bank Account (3/5)

BankAcct.java

```
    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) throws
                             NotEnoughFundException {
        if (balance >= amount) {
            balance -= amount;
        } else {
            double needs = amount - balance;
            throw new NotEnoughFundException(
                       "Withdrawal Unsuccessful", needs);
        }
    }

} // class BankAcct
```

## 6. Example: Bank Account (4/5)

TestBankAcct.java

```java
public class TestBankAcct {

   public static void main(String[] args) {

      BankAcct acc = new BankAcct(1234, 0.0);

      System.out.println("Current balance: $" +
                         acc.getBalance());

      System.out.println("Depositing $200...");
      acc.deposit(200.0);

      System.out.println("Current balance: $" +
                         acc.getBalance());
```

```
Current balance: $0.0
Depositing $200...
Current balance: $200.0
```

# 6. Example: Bank Account (5/5)

TestBankAcct.java

```
      try {
         System.out.println("Withdrawing $150...");
         acc.withdraw(150.0);
         System.out.println("Withdrawing $100...");
         acc.withdraw(100.0);
      }
      catch (NotEnoughFundException e) {
          System.out.println(e.getMessage());
          System.out.println("Your account is short of $" +
                                   e.getAmount());
      }
      finally {
         System.out.println("Current balance: $" +
                                  acc.getBalance());
      }
   } // main

} // class TestBankAcct
```

```
Current balance: $0.0
Depositing $200...
Current balance: $200.0
Withdrawing $150...
Withdrawing $100...
Withdrawal Unsuccessful
Your account is short of $50.0
Current balance: $50.0
```

## Practice Exercise

- Practice Exercises
  - #24: Finding a Root of a Quadratic Equation
  - #25: Making a Date

## Summary

- We learned about exceptions, how to raise and handle them
- We learned how to define new exception classes

End of file