

OOP Design & Testing

Exceptions

COMP3607

Object Oriented Programming II

Week 11

Outline

- Exceptions
 - What are they?
 - How to handle?
 - When not to handle?

Exceptions

Exceptions occur when the normal or expected flow of a program is disrupted or the program ends unnaturally. These are usually related to program errors.

The JVM will normally stop and generate an error message when errors occur. The technical term for this is: Java will throw an **exception** (throw an error).

Common Causes

An exception can occur for many different reasons.

Examples:

- ➔ A user has entered an invalid data.
- ➔ Operations on missing 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.
- ➔ System running out of memory

Main Types of Exceptions

- ◉ Programming Errors
 - Clients can't recover from these
- ◉ Prohibited Invocations
 - Client may recover with alternative course of action if enough information is provided about the error
- ◉ Resource Failure
 - Client can retry after some time or halt the system

Exceptions in Java

Two main types of exceptions:

1. Checked exceptions
 2. Unchecked exceptions
- If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception

<https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html>

Checked Exceptions

These are generated by an object providing a service to a client where the client is forced to acknowledge and deal with the exception.

These are verified at compile time and are generally outside of the control of the program.

Methods can 'advertise' that exceptions occur and therefore a client has to handle it.

Unchecked Exceptions

These tend to occur when there is an error within a program and are therefore not checked at compile time.

Exception Handling

“Exception handling involves building an application to detect and recover from exceptional conditions.” (Mohan 2003).

The goal is to produce robust, reliable software.

Try/Catch Block

The **try** statement allows you to define a block of code to be tested for errors while it is being executed.

The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.

The **try** and **catch** keywords come in pairs:

Syntax

```
try {  
    // Block of code to try  
}  
catch(Exception e) {  
    // Block of code to handle errors  
}
```

Example

This will generate an error, because **myNumbers[10]** does not exist.

```
public class MyClass {  
    public static void main(String[ ] args) {  
        int[] myNumbers = {1, 2, 3};  
        System.out.println(myNumbers[10]); // error!  
    }  
}
```

The output will be something like this:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10  
    at MyClass.main(MyClass.java:4)
```

Run example »

Example - try/catch

Example

```
public class MyClass {  
    public static void main(String[] args) {  
        try {  
            int[] myNumbers = {1, 2, 3};  
            System.out.println(myNumbers[10]);  
        } catch (Exception e) {  
            System.out.println("Something went wrong.");  
        }  
    }  
}
```

The output will be:

```
Something went wrong.
```

Run example »

Finally

The finally statement lets you execute code, after try...catch, regardless of the result.

This is used for cleaning up after an exception e.g.
closing file streams, disconnecting database objects.

Example - try/catch/finally

Example

```
public class MyClass {  
    public static void main(String[] args) {  
        try {  
            int[] myNumbers = {1, 2, 3};  
            System.out.println(myNumbers[10]);  
        } catch (Exception e) {  
            System.out.println("Something went wrong.");  
        } finally {  
            System.out.println("The 'try catch' is finished.");  
        }  
    }  
}
```

The output will be:

```
Something went wrong.  
The 'try catch' is finished.
```

[Run example »](#)

Example with Checked Exceptions

We will use a simple Account class hierarchy to show how to create checked exceptions that a client must handle.


First we look at the Account, ChequingAccount and SavingsAccount classes and how errors can occur.

Then we will create checked exception classes to handle the errors gracefully.

Account

```
1  public class Account{
2      ⚡ private int number;
3      private double balance;
4
5      public Account(int num, double bal){
6          number = num;
7          balance = bal;
8      }
9
10     public void withdraw(double amount){
11         if(amount < balance){
12             balance = balance-amount;
13         }
14     }
15
16
17     public void deposit(double amount){
18         balance = balance + amount;
19     }
20
21     public String toString(){
22         return number + " " + balance;
23     }
24
25 }
```


Savings Account

```
1 public class SavingsAccount extends Account{  
2       
3     public SavingsAccount(int number, double balance){  
4         super(number, balance);  
5     }  
6  
7     public String toString(){  
8         String s = "Savings :"+super.toString();  
9         return s;  
10    }  
11 }  
12
```

```
1 public class ChequingAccount extends Account{
2
3     private int numTransactions;
4     private double fees;
5
6     public ChequingAccount(int number, double balance){
7         super(number, balance);
8     }
9
10    public void calculateFees(){
11        fees = numTransactions * 0.75;
12    }
13
14    public void withdraw(double amount){
15        super.withdraw(amount);
16        numTransactions++;
17    }
18
19    public String toString(){
20        String s = "Chequing :"+super.toString();
21        s= s + " " + numTransactions + " " + fees;
22        return s;
23    }
24
25 }
```

**Allowed
withdrawal**

```
1  public class BankApp {  
    Run | Debug  
2      public static void main(String[] args){  
3          Account a = new ChequingAccount (10, 50000.00);  
4          a.withdraw(11000.00);  
5          System.out.println(a.toString());  
6      }  
7  }  
8  }  
9  }
```

OUTPUT

TERMINAL

...

1: Java Process Console

v



```
Chequing :10 39000.0 1 0.0  
bash-3.2$
```

**Not allowed
withdrawal**

```
1 public class BankApp {  
    Run | Debug  
2     public static void main(String[] args){  
3         Account a = new ChequingAccount (10, 50000.00);  
4         a.withdraw(10000000.00);  
5         System.out.println(a.toString());  
6     }  
7  
8 }  
9
```

OUTPUT PROBLEMS 3 TERMINAL ...

1: Java Process Console

```
Chequing :10 50000.0 1 0.0  
bash-3.2$
```

**Should not
be allowed
but was
allowed
withdrawal**

```
1 public class BankApp {  
    Run | Debug  
2     public static void main(String[] args){  
3         Account a = new ChequingAccount (10, 50000.00);  
4         a.withdraw(-110000000.00);  
5         System.out.println(a.toString());  
6     }  
7  
8 }  
9
```

OUTPUT

TERMINAL

...

1: Java Process Console

▼

+

□

🗑

^

×

```
Chequing :10 1.1005E8 1 0.0  
bash-3.2$
```

Runner class

**Should not
be allowed
but was
allowed
withdrawal**

```
1 public class BankApp {  
    Run | Debug  
2     public static void main(String[] args){  
3         Account a = new ChequingAccount (10, 50000.00);  
4         a.withdraw(49999.99);  
5         System.out.println(a.toString());  
6     }  
7  
8 }  
9
```

OUTPUT PROBLEMS 3 TERMINAL ...

1: Java Process Console




```
Chequing :10 0.010000000002037268 1 0.0  
bash-3.2$
```


Creating Checked Exceptions

Now, we will introduce two custom exception classes:

1. `InsufficientFundsException`
2. `TooLargeWithdrawalException`

Custom Exception Classes

```
1 public class InsufficientFundsException extends Exception{  
2       
3     public InsufficientFundsException(){  
4         super("Not enough funds to permit operation");  
5     }  
6  
7     public InsufficientFundsException(String message){  
8         super(message);  
9     }  
10  
11  
12 }
```

```
1 public class TooLargeWithdrawalException extends Exception{  
2       
3     public TooLargeWithdrawalException(){  
4         super("Not enough funds to permit operation");  
5     }  
6  
7     public TooLargeWithdrawalException(String message){  
8         super(message);  
9     }  
10  
11  
12 }
```


Account with checked exception

```
1 public class Account{
2     private int number;
3     private double balance;
4
5     > public Account(int num, double bal){ ...
6
7
8
9
10    public void withdraw(double amount) throws InsufficientFundsException,
11        TooLargeWithdrawalException{
12
13        if(amount > 10000){
14            String message = "You cannot withdraw more than $10,000 in one transaction";
15            throw new TooLargeWithdrawalException(message);
16        }
17
18        if(amount < balance){
19            balance = balance-amount;
20        }
21        else{
22            throw new InsufficientFundsException();
23        }
24    }
25
26
27    > public void deposit(double amount){ ...
28
29
30
31    > public String toString(){ ...
32
33
34
35 }
```

Chequing Account with checked exception

```
1  public class ChequingAccount extends Account{
2
3      private int numTransactions;
4      private double fees;
5
6  >  public ChequingAccount(int number, double balance){ ...
9
10 >  public void calculateFees(){ ...
13
14
15      public void withdraw(double amount) throws InsufficientFundsException,
16          TooLargeWithdrawalException{
17          try{
18              super.withdraw(amount);
19              numTransactions++;
20          }
21          catch(InsufficientFundsException ife){
22              throw ife;
23          }
24          catch(TooLargeWithdrawalException tlwe){
25              throw tlwe;
26          }
27      }
28
29
30 >  public String toString(){ ...
35
36 }
```

```
1 public class BankApp{
2
3     Run | Debug
4     public static void main(String[] args){
5         Account a = new ChequingAccount (10, 50000.00);
6         try{
7             a.withdraw(11000.00);
8         }
9         catch(InsufficientFundsException ife){
10             System.out.println(ife.getMessage());
11         }
12         catch(TooLargeWithdrawalException tlwe){
13             System.out.println(tlwe.getMessage());
14         }
15         System.out.println(a.toString());
16     }
17 }
```

OUTPUT PROBLEMS 8 DEBUG CONSOLE TERMINAL

```
You cannot withdraw more than $10,000 in one transaction
Chequing :10 50000.0 0 0.0
bash-3.2$
```

**Custom error
message 1**

```
1 public class BankApp{
2
3     Run | Debug
4     public static void main(String[] args){
5         Account a = new ChequingAccount (10, 800.00);
6         try{
7             a.withdraw(1000.00);
8         }
9         catch(InsufficientFundsException ife){
10            System.out.println(ife.getMessage());
11        }
12        catch(TooLargeWithdrawalException tlwe){
13            System.out.println(tlwe.getMessage());
14        }
15        System.out.println(a.toString());
16    }
```

OUTPUT PROBLEMS 2 DEBUG CONSOLE TERMINAL

1: Java Pr

```
Not enough funds to permit operation
Chequing :10 800.0 0 0.0
bash-3.2$
```

**Custom error
message 2**

References

- https://www.tutorialspoint.com/java/java_exceptions.htm
- https://www.w3schools.com/java/java_try_catch.asp
- <https://www.baeldung.com/java-checked-unchecked-exceptions>
- <https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html>
-