Lecture 14 — Exceptions

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- See Chapter 9 of the textbook
- try-catch
- Throwing exceptions
- Checked and unchecked exceptions
- Creating our own exceptions

Exceptions

- I hope you have found all the lectures so far to be good. Today's will be exceptional
- We have seen many errors
- Compiler errors (e.g., misspelled variable name)
- Run-time errors (e.g., accessing an array out of bounds)
- Java comes with a way to handle run-time errors
- When some kinds of run-time error occur, an exception is "thrown"
- It is possible to "catch" the exception and deal with it, preventing the program from crashing

ArrayException

```java
import java.util.Scanner;

public class ArrayException {
    public static void main(String[] args) {
        int[] a = {882, 2, 11};
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter index: ");
        int index = sc.nextInt();
        System.out.println(a[index]);
    }
}
```

- If we give the an out-of-bounds index, an error occurs
- Java throws an `java.lang.ArrayIndexOutOfBoundsException`

Try-Catch

```
try {
    // some code
} catch (ExceptionType exception) {
    // handle the exception
}
```

- A try-catch block is used to "catch" exceptions
- The code inside `try` gets executed
- If an exception occurs, it stops, and the `catch` is checked
- If the `ExceptionType` matches the exception thrown, it gets "caught"
- In this case, we can execute the `catch` code to deal with the error
- If no exception is thrown, the catch block is skipped
- Let's use it on our `ArrayException` example from before

ArrayTryCatch

```java
import java.util.Scanner;

public class ArrayTryCatch {
    public static void main(String[] args) {
        int[] a = {882, 2, 11};
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter index: ");
        int index = sc.nextInt();
        try {
            System.out.println(a[index]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught ArrayIndexOutOfBoundsException");
        }
    }
}
```

Multiple Catches

```
int[] a = {882, 2, 11};
Scanner sc = new Scanner(System.in);
System.out.print("Enter index: ");
int index = sc.nextInt();
try {
    System.out.println(a[index]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Caught ArrayIndexOutOfBoundsException");
}
```

- There is a problem with the code
- What's the catch?
- Pun translation: It is still possible for this code to crash. How?

Multiple Catches

```
Scanner sc = new Scanner(System.in);
System.out.print("Enter index: ");
int index = sc.nextInt();
```

- `Scanner` can generate an exception
- If the next token is not an int, it will throw a `java.util.InputMismatchException`
- For example, if we entered "oops", the program would crash
- A try-catch block can have any number of catches!
- `try { ... } catch (A var) { ... } catch (B var) { ... } ...`
- It will check each of them in order until one matches the exception

ArrayTryCatch2

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

public class ArrayTryCatch2 {
    public static void main(String[] args) {
        int[] a = {882, 2, 11};
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter index: ");
        try {
            int index = sc.nextInt();
            System.out.println(a[index]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught ArrayIndexOutOfBoundsException");
        } catch (InputMismatchException e) {
            System.out.println("Please enter a valid integer next time");
        }
    }
}
```

Exception Control Flow

- Let's understand what happens when an exception gets thrown
- First, something goes wrong (e.g., array index out of bounds)
- An exception is an object. One is created and "thrown"
- If the exception is not caught, the method immediately terminates
- The exception goes up to the parent method that called the current method
- This continues until either the exception is caught, or we run out of methods
- If it remains uncaught, the program crashes and reports the error

Catching Exceptions

- It is up to us to decide what to do when we catch an exception
- Sometimes, it is enough to print an error message
- Other examples:
  - A server failed to respond. Wait and try again
  - A user provided file did not exist. Ask them to enter the name again
  - The microphone stopped worked. Switch to a different one
  - The program ran out of memory. Wait for more memory to become free
  - A cosmic ray hit the computer and now the memory is corrupt. Restart the program

Exceptions and Inheritance

```
try {
    // code
} catch (ExceptionType e) {
    // deal with e
}
```

- Exceptions are a good example of inheritance
- We have said that `catch` matches the type of an exception
- What we mean is that the exception is a subclass
- In the example, the exception must be a subclass of `ExceptionType`
- This means `catch` can actually catch multiple exceptions
- All caught exceptions should be a subclass of `Exception`
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Exception.html`

Exceptions and Inheritance

```
try {
    // code
} catch (Exception e) {
    // deal with any exception
}
```

- All exceptions inherit from `Exception`
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Exception.html`
- This means we can catch (almost) all exceptions using the catch above
- There are actually other things you can catch (but you probably shouldn't)

Exceptions and Inheritance

```java
import java.util.Scanner;

public class ArrayTryCatch3 {
    public static void main(String[] args) {
        int[] a = {882, 2, 11};
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter index: ");
        try {
            int index = sc.nextInt();
            System.out.println(a[index]);
        } catch (Exception e) {
            System.out.println("Something went wrong: " + e);
        }
    }
}
```

- As a rule, catching `Exception` is not good code
- It doesn't let you deal with the specific error that occurred
- Usually only used as a "catch all"
- As we will see, you can extend `Exception` to make your own exceptions

The Throwable Hierarchy

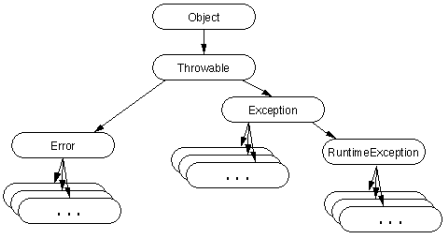


Image from https://www.whitman.edu/mathematics/java_tutorial/java/exceptions/throwable.html

- `Throwable` is anything that can be thrown
- `Error` is something you shouldn't catch as it usually represents a point of no return (e.g., the Java VM crashed)
- `Exception` is something you might want to catch
- `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Throwable.html`

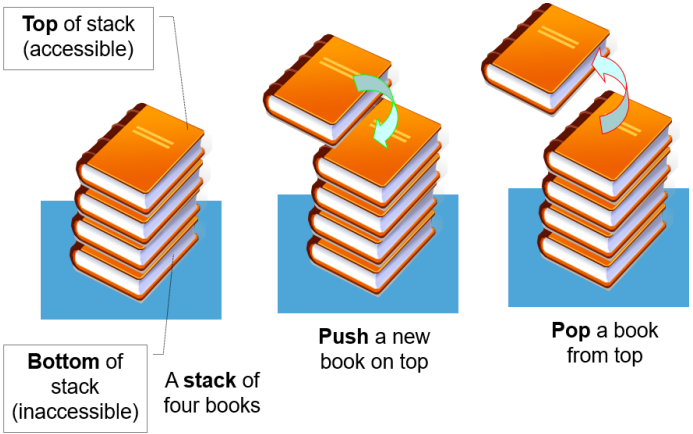
Mid-Lecture Break

StringStack class

```
public class StringStack {
    public StringStack(int capacity) { ... }
    public void push(String s) { ... }
    public String pop() { ... }
}
```

- Let's learn about throwing our own exceptions by making a StringStack class
- A stack is a minimal data structure
- Stack as in stack of plates or stack of pancakes
- You can only take the topmost thing, or put something on the top
- Our stack will be for strings
- Our stack will have a maximum capacity

StringStack class



Taken from `https://visualgo.net/en/list`

StringStack Fields & Constructor

```
private String[] data;
private int top;

public StringStack(int capacity) {
    data = new String[capacity];
    top = 0;
}
```

- Here are the fields and constructor
- We store an array containing the items in the stack
- The top field is an index to where the (exclusive) top of the stack is
- This will move as we push/pop from the stack

StringStack push()

```
public void push(String s) {
    if (top == data.length) {
        throw new RuntimeException("Stack is full");
    }
    data[top++] = s;
}
```

- This method pushes a string onto the stack
- Throws an exception if the stack is full
- The syntax for throwing an exception: `throw an-exception-object;`
- Here, we're using an existing exception class: `RuntimeException`

StringStack pop()

```
public String pop() {
    if (top == 0) {
        throw new RuntimeException("Stack is empty");
    }
    return data[--top];
}
```

- Removes and returns the top of the stack
- Similar to `push()`

StringStack Fields & Constructor

```
public static void main(String[] args) {
    StringStack ss = new StringStack(5);
    ss.push("Hello");
    ss.push("World");
    ss.push("!");
    System.out.println(ss.pop());
    System.out.println(ss.pop());
    System.out.println(ss.pop());
    System.out.println(ss.pop());
}
```

- Running this demo should cause an exception to be thrown, which crashes the program
- A question: what is `RuntimeException` and why did we use it?

RuntimeException

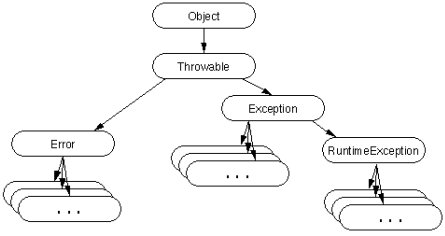


Image from https://www.whitman.edu/mathematics/java_tutorial/java/exceptions/throwable.html

- There are two kinds of exceptions: `RuntimeException` and everything else
- A `RuntimeException` is allowed to crash the program and represents a logical error in the program
- They are called "unchecked exceptions" because you don't have to check them
- Other exceptions are called "checked exceptions", and you must check them using a try-catch

StackFull class

```
public class StackFull extends Exception {
    public StackFull(String message) {
        super(message);
    }
}
```

- Here is a new exception class describing a situation where our stack is full
- We can make our own exceptions by extending `Throwable` or a subclass
- In most cases, you should only extend `Exception`, `RuntimeException`, or one of their subclasses
- `Exception` is generally for recoverable errors, and `RuntimeException` is for program errors
- `Error` is for system errors
- For our stack, a `RuntimeException` is probably a better choice, but we use `Exception` to demonstrate checked exceptions

StackFull class

```
public void push(String s) {
    if (top == data.length - 1) {
        throw new StackFull("Stack can only hold " + data.length + "
            elements");
    }
    data[top++] = s;
}
```

- Lets change the code to throw the new exception
- We will create a `StringStack2` class. Here is the `push()` method
- There is a problem: `unreported exception StackFull; must be caught or declared to be thrown`
- `Exception` is a checked exception
- If a method throws a checked exception, it must declare that using the `throws` keyword

StackFull class

```
public void push(String s) throws StackFull {
    if (top == data.length - 1) {
        throw new StackFull("Stack can only hold " + data.length + "
            elements");
    }
    data[top++] = s;
}
```

- The `throws` keyword is used as above
- A comma-separated list can be used to throw multiple exceptions:
  `throws Exception1, Excpetion2, ...`

StringStack2 class

```
public static void main(String[] args) {
    StringStack2 ss = new StringStack2(5);
    ss.push("Hello");
    ss.push("World");
    ss.push("!");
    ss.push("Hello");
    ss.push("World");
    ss.push("!");
}
```

- Java will not compile this main method
- This is because `.push()` was marked with `throws`
- A try-catch must be used

StringStack2 class

```
public static void main(String[] args) {
    StringStack2 ss = new StringStack2(5);
    try {
        ss.push("Hello");
        ss.push("World");
        ss.push("!");
        ss.push("Hello");
        ss.push("World");
        ss.push("!");
    } catch (StackFull e) {
        System.out.println(e);
    }
}
```

- If we add a try-catch, the checked exception is handled and Java will compile our program