EXCEPTIONS AND THREADS IN JAVA

Dr. Krishnendu Guha

Assistant Professor/ Lecturer

School of Computer Science and Information Technology

University College Cork

WHAT ARE JAVA EXCEPTIONS?

- When we execute a code in Java, various **errors** can occur:
  - **coding errors** made by programmer,
  - errors due to **wrong input**, or
  - **other** unforeseeable things.

- When an error occurs, Java will **normally stop and generate an error message**.

- The technical term for this is: Java will throw an **exception** (or simply generate an error).

HANDLING JAVA EXCEPTIONS (TRY… CATCH)

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

The catch statement allows us 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:

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

EXAMPLE:

```
public class Main {
        public static void main(String[ ] args) {
                int[] numList = {10, 20, 30};
                System.out.println(numList[100]); // error!
        }
}
```

This will generate an error, because **numList[100]** does not exist.

- try...catch block can
- **catch the error** and
- **execute some code to handle it**

- **Some output will be generated** and
- the **program will not generate an error message**

```
public class Main {
        public static void main(String[ ] args) {
                try {
                        int[] numList = {10, 20, 30};
                        System.out.println(numList[100]);
                }
                catch (Exception e) {
                        System.out.println("Something went wrong.");
                }
        }
}
```

```
Something went wrong.
```

Finally

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

```
public class Main {
        public static void main(String[] args) {
                try {
                        int[] numList = {10, 20, 30};
                        System.out.println(numList[100]);
                }
                catch (Exception e) {
                        System.out.println("Something went wrong.");
                }
                finally {
                        System.out.println(“Program finished after try catch");
                }
        }
}
```

```
Something went wrong.
Program finished after try catch
```

The throw keyword

The **throw statement** allows us to **create a custom error**.
The throw statement is used together with an **exception type**.

There are many exception types available in Java:

- ArithmeticException,
- FileNotFoundException,
- ArrayIndexOutOfBoundsException,
- SecurityException, etc:

The program throws an exception if **age** is below 18 (print "Access denied"), else prints "Access granted":

```
public class Main {
	static void checkAge(int age) {
		if (age < 18) {
			throw new
ArithmeticException("Access denied");
		}
		else {
			System.out.println("Access
granted");
		}
	}
	public static void main(String[] args) {
		checkAge(15); // Set age to 15 (which is
below 18...)
	}
}
```

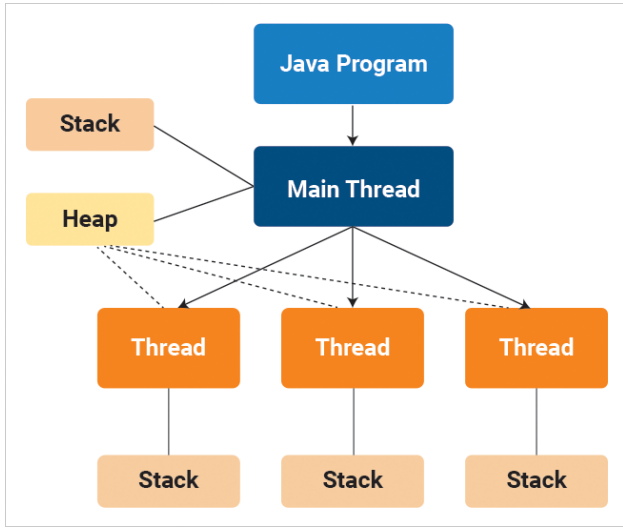
```
Exception in thread "main" java.lang.ArithmeticException: Access denied
        at Main.checkAge(Main.java:4)
        at Main.main(Main.java:11)
```

```
public class Main {
	static void checkAge(int age) {
		if (age < 18) {
			throw new
ArithmeticException("Access denied");
		}
		else {
			System.out.println("Access
granted");
		}
	}
	public static void main(String[] args) {
		checkAge(20); // Set age to 20
	}
}
```

```
Access granted
```

THREADS

- Threads **allows a program to operate more efficiently** by **doing multiple things at the same time**.
- Threads can be used to **perform complicated tasks in the background without interrupting the main program**.



Creating a Thread

There are two ways to create a thread.
It can be created by **extending** the Thread class and **overriding** its run() method:

Syntax for Extending the Thread Class

```
public class Main extends Thread {
        public void run() {
                System.out.println("This code is running in a thread");
        }
}
```

Syntax for implementing the Runnable interface:

```
public class Main implements Runnable {
        public void run() {
                System.out.println("This code is running in a thread");
        }
}
```

RUNNING THREADS

If the class **extends** the Thread class, ***the thread can be run by creating an instance of the class*** and call its start() method:

```
public class Main extends Thread {
        public static void main(String[] args) {
                Main thread = new Main();
                thread.start();
                System.out.println("This code is outside of the thread");
        }
        public void run() {
                System.out.println("This code is running in a thread");
        }
}
```

```
This code is outside of the thread
This code is running in a thread
```

If the class **implements** the Runnable interface, the ***thread can be run by passing an instance of the class to a Thread object's constructor*** and then calling the thread's start() method:

```
public class Main implements Runnable {
        public static void main(String[] args) {
                Main obj = new Main();
                Thread thread = new Thread(obj);
                thread.start();
                System.out.println("This code is outside of the thread");
        }
        public void run() {
                System.out.println("This code is running in a thread");
        }
}
```

```
This code is outside of the thread
This code is running in a thread
```

Differences between "extending" and "implementing" Threads

The major difference is that when a class **extends** the Thread class, **you cannot extend any other class**, but by **implementing the Runnable interface**, it is **possible to extend from another class as well**, like:
class MyClass extends OtherClass implements Runnable.

CONCURRENCY PROBLEMS

What is Concurrency Problem?

- Because ***threads run at the same time as other parts of the program***, there is **no way to know in which order the code will run.**
- When the **threads and main program are reading and writing the same variables**, the **values are unpredictable**.
- The problems that result from this are called concurrency problems.

A code example where the value of the variable **amount** is unpredictable:

```
public class Main extends Thread {
        public static int amount = 0;
        public static void main(String[] args) {
                Main thread = new Main();
                thread.start();
                System.out.println(amount);
                amount++;
                System.out.println(amount);
        }
        public void run() {
                amount++;
        }
}
```

- To **avoid concurrency problems**, it is best to **share as few attributes between threads** as possible.
- If **attributes need to be shared**, one possible solution is to use the **isAlive() method** of the thread to *check whether the thread has finished running, before using any attributes that the thread can change*

```
public class Main extends Thread {
        public static int amount = 0;
        public static void main(String[] args) {
                Main thread = new Main();
                thread.start();
                // Wait for the thread to finish
                while(thread.isAlive()) {
                        System.out.println("Waiting...");
                }
                // Update amount and print its value
                System.out.println("Main: " + amount);
                amount++;
                System.out.println("Main: " + amount);
        }
        public void run() {
                amount++;
        }
}
```

```
Waiting...
Main: 1
Main: 2
```

```
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Main: 1
Main: 2
```


THANK YOU
Any Questions?