

JAVAGURU INTRODUCTION TO JAVA

LESSON 13

EXCEPTIONS

OVERVIEW

WHAT ARE EXCEPTIONS?

- ▶ An *exception* is an **event**, which occurs during the execution of a program, that **disrupts the normal flow** of the program's instructions (e.g. accessing null reference, array access out of bound, etc.)
- ▶ An *exception* is an **object** that represents an **error** that occurred within a method and contains:
 - ▶ Information about the error including its **type**
 - ▶ The **state** of the program when the error occurred
- ▶ Exception objects can be *thrown* and *caught*

CLASSIFYING ERRORS AND EXCEPTIONS

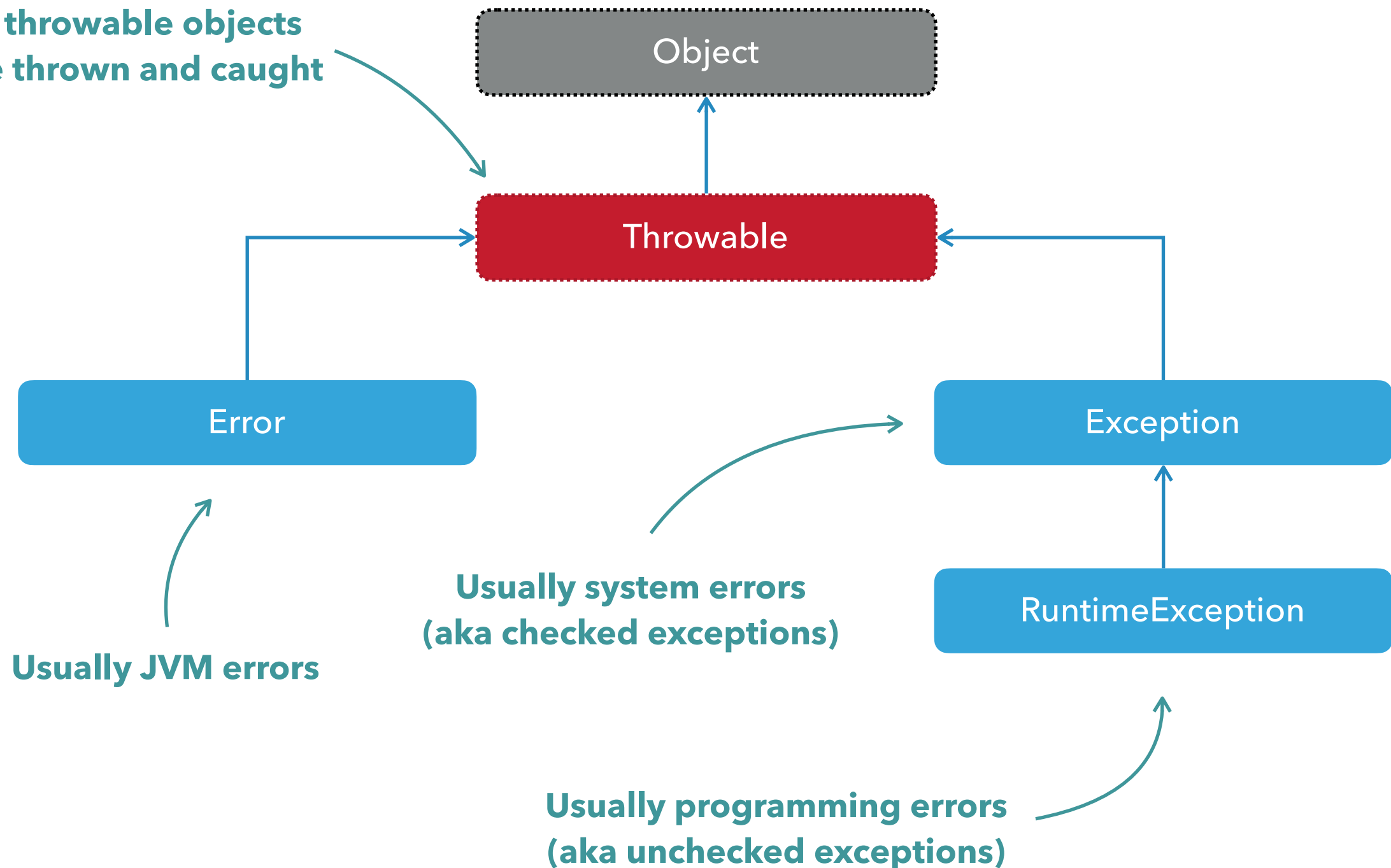
- ▶ JVM errors
 - ▶ e.g. OutOfMemoryError, StackOverflowError, etc.
- ▶ System errors
 - ▶ e.g. FileNotFoundException, IOException, etc.
- ▶ Programming errors
 - ▶ e.g. NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, etc.

WHY USE EXCEPTIONS?

- ▶ Exceptions **separate** error **handling** code from **regular** code
 - ▶ Algorithms become **cleaner** and **less** clutter
- ▶ Exceptions **propagate** errors up the call stack
 - ▶ Nested methods do not have to **explicitly** catch-and-forward errors
- ▶ Exception classes **group** and **differentiate** error types
 - ▶ You can group errors by their **parent** class (polymorphism)
 - ▶ Differentiate errors by their **actual** class
- ▶ Exceptions **standardise** error handling

EXCEPTION TYPES HIERARCHY

Only throwable objects
can be thrown and caught



CHECKED EXCEPTIONS

- ▶ Normally used to denote **expected system failures** with **reasonable recover** (e.g. missing file on the file system or connection establishment failure)
- ▶ The compiler **enforces** that you **handle** them explicitly
- ▶ Methods that might produce checked exceptions **must declare** it in method **signature**
- ▶ Methods that **invoke** other methods that throw checked exceptions must either:
 - ▶ **Handle** them (i.e. they can be reasonably expected to recover)
 - ▶ Declare in method signature that it might produce checked exception thus allowing it to **propagate**
- ▶ Exception class and its derivatives are **checked** (except RuntimeException)

UNCHECKED EXCEPTIONS

- ▶ Normally used to denote **unexpected programming** or **logical** errors
- ▶ The compiler **does not enforce** that you **handle** them explicitly
- ▶ It is assumed that the application **cannot do anything** to **recover** from these exceptions at runtime
- ▶ Error and RuntimeException and its derivatives are **unchecked**

UNCHECKED EXCEPTION: CODE EXAMPLE

**String does not reference
to anything**

```
String emptyString = null;  
emptyString.isEmpty();
```

**Calling method on something that
does not exist is impossible
and will raise NullPointerException**

CHECKED EXCEPTION: METHOD SIGNATURE CODE EXAMPLE

Declaring that our
method might throw exception
and its type

```
public String readFile(String path) throws IOException {  
    byte[] bytes = Files.readAllBytes(Paths.get(path));  
    return new String(bytes);  
}
```

Unsafe operation, because
file might be broken or for
some reason can't be read

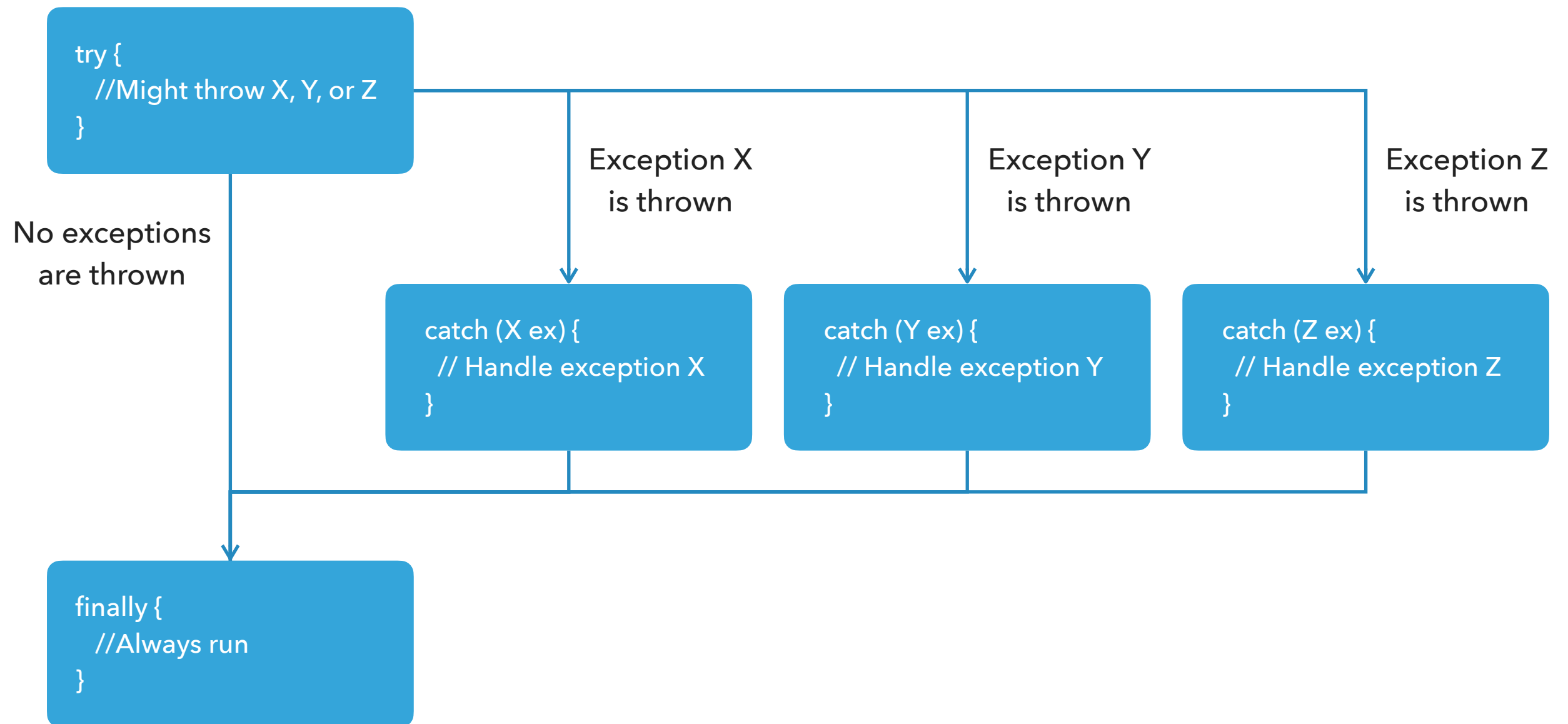
EXCEPTION LIFECYCLE

1. After an exception object is created, it is **handed off** to the runtime system (*thrown*)
2. The runtime system attempts to **find** a handler for the exception by backtracking the ordered list of methods that had been **called** (aka the call stack)
3. If a handler is **found**, the exception is *caught*:
 - 3.1. It is handled, or possibly re-thrown
4. If the handler is **not found** (the runtime backtracks all the way to the `main()` method):
 - 4.1. The exception stack trace is printed to the standard error channel
 - 4.2. Application aborts execution

HANDLING EXCEPTIONS

- ▶ Java supports special try-catch-finally control structure that allows to handle and recover from exceptions
- ▶ Consists of 3 blocks:
 - ▶ try block
 - ▶ Must be present when invoking unsafe method and declared only once
 - ▶ Specifies that executed code block will be handled if exception occurs
 - ▶ catch block
 - ▶ Must be present and can be declared as many times as required
 - ▶ Specifies how to handle concrete exception or group of exceptions
 - ▶ finally block
 - ▶ Optional and declared only once (Must be present if catch block is absent)
 - ▶ Always executes action after try and catch blocks (if any declared)

TRY-CATCH-FINALLY: FLOW CHART



TRY-CATCH-FINALLY: CODE EXAMPLE

Unsafe method is the one with
throws declaration in signature

```
try {  
    //Invoke unsafe method  
} catch (Exception1 ex) {  
    //Handle Exception1 type  
} catch (Exception2 ex) {  
    //Handle exception2 type  
} finally {  
    //This part always executed  
    //after try and catch blocks  
}
```

Handling all exceptions
declared in the signature

REFERENCES

- ▶ <https://docs.oracle.com/javase/tutorial/essential/exceptions/>
- ▶ <https://beginnersbook.com/2013/04/try-catch-in-java/>
- ▶ https://www.tutorialspoint.com/java/java_exceptions.htm
- ▶ <https://stackify.com/best-practices-exceptions-java/>