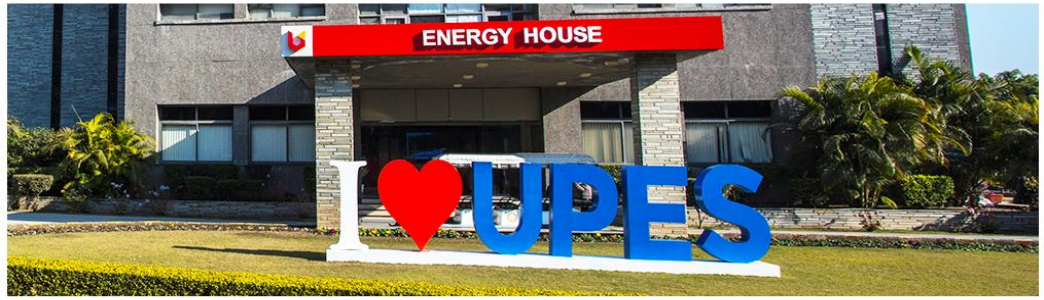


UPES

UNIVERSITY WITH A PURPOSE

Object Oriented Programming

UPES
UNIVERSITY WITH A PURPOSE

Exceptions in Java

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

Exceptions in Java

- Program statements that you want to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown.
- Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java runtime system.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

Exceptions in Java

- This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors }
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1 }
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2 }
// ...
finally {
// block of code to be executed after try block ends }
```

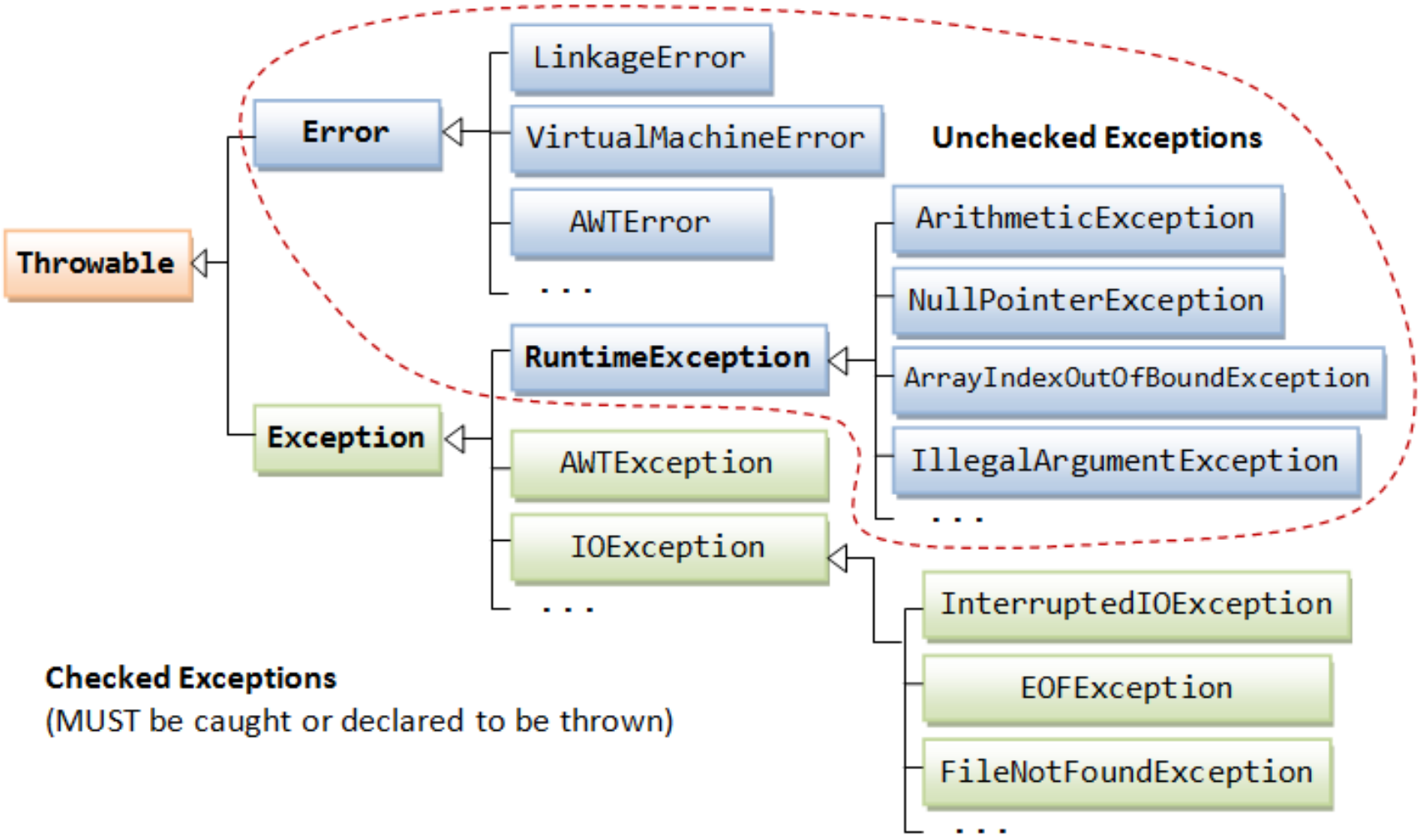
Here, `ExceptionType` is the type of exception that has occurred.

UPES
UNIVERSITY WITH A PURPOSE

Exception Types

- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.
- There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.
- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Exception Types



Source: https://www.ntu.edu.sg/home/ehchua/programming/java/J5a_ExceptionAssert.html

UPES
UNIVERSITY WITH A PURPOSE

Exception Types

- The base class for all `Exception` objects is `java.lang.Throwable`, together with its two subclasses `java.lang.Exception` and `java.lang.Error`.
- The Error class describes internal system errors (e.g., VirtualMachineError, LinkageError) that rarely occur. If such an error occurs, there is little that you can do and the program will be terminated by the Java runtime.
- The Exception class describes the error caused by your program (e.g. FileNotFoundException, IOException).
- These errors could be caught and handled by your program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).

Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This program includes an expression that intentionally causes a divide-by-zero error:

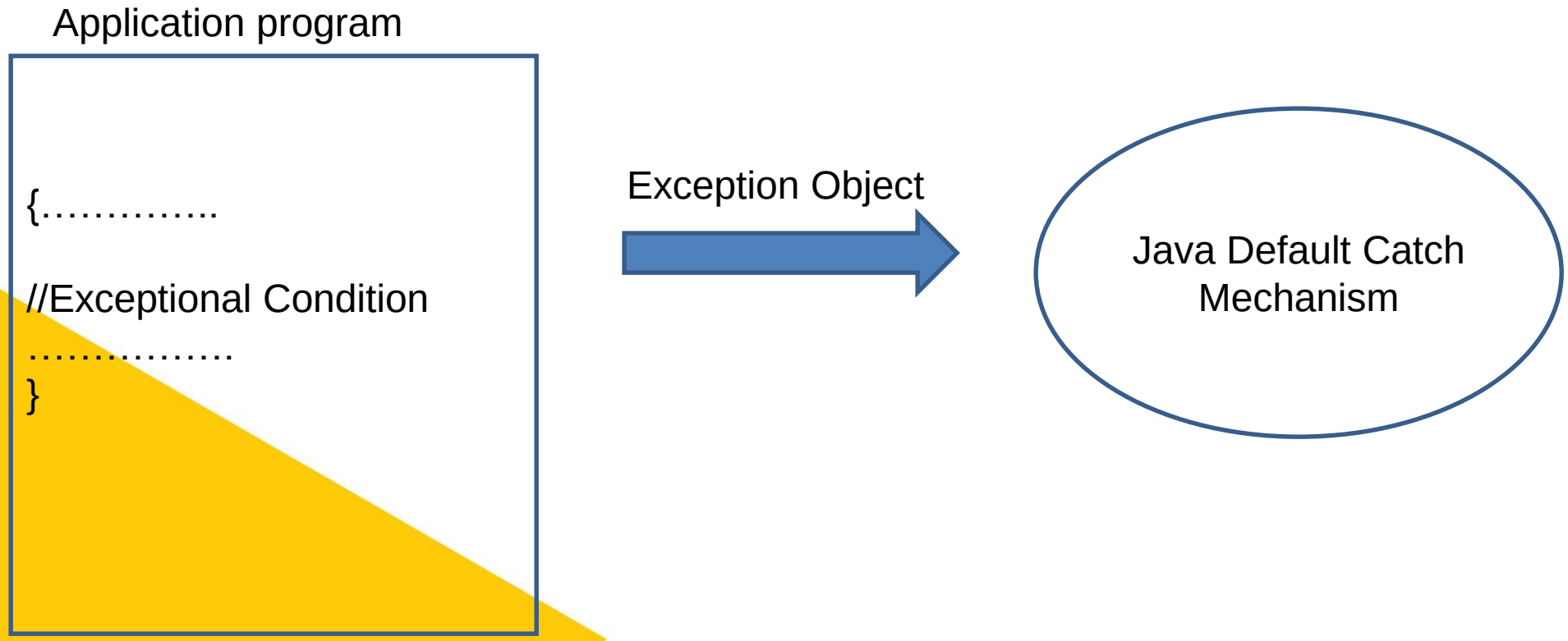
```
class Exc0 {
public static void main(String args[]) {
int d = 0;
int a = 42 / d; }  }
```

The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the exception generated when this example is executed:

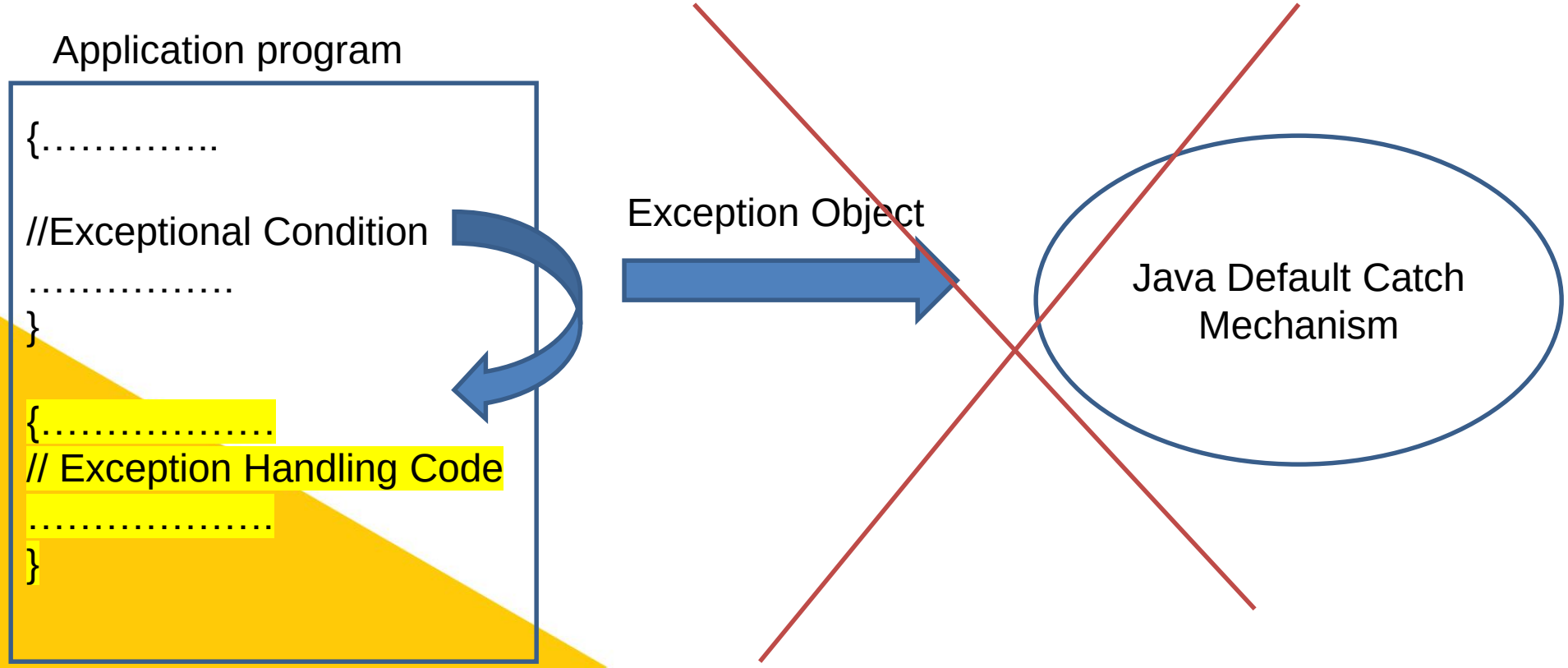
```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace. Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened.

Uncaught Exceptions



Catching Exceptions



Benefits:

1. Program will continue and will not end.
2. We can give specific message
3. We can handle certain exceptions which may not be handled by Java by default.

UPES
UNIVERSITY WITH A PURPOSE

Handling Exceptions

There may be 4 situations for exception handling:

1. Default throw and default catch mechanism
2. Default throw and user defined catch mechanism
3. User defined throw and default catch mechanism
4. User defined throw and user defined catch mechanism

Note: Java exceptions are raised with the **throw** keyword and within the **catch** block

Throwable Class

- The `Throwable` class provides a `String` variable that can be set by the subclasses to provide a detailed message to give the information of the exception.
- All classes of `Throwable` define a one-parameter constructor that takes the String as the detailed message.
- The class `Throwable` provides a `getmessage()` method to retrieve the exception.

Using try and catch

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
}
catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Displaying a Description of an Exception

- **Throwable** overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception.
- You can display this description in a **println( )** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}
```

- When this version is substituted in the program, and the program is run, each divide-byzero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

Multiple catch Clauses

```
// Demonstrate multiple catch statements.
class MultipleCatches {
public static void main(String args[]) {
try {
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e); }
System.out.println("After try/catch blocks."); } }
```

This program will cause a division-by-zero exception if it is started with no command line arguments, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

References

Schildt, H. (2014). *Java: the complete reference*. McGraw-Hill Education Group.