### **Reading**

Please read Chapter 12 of book.

### **Exceptions**

Runtime errors (such as array index out of bounds or input mismatch) are exceptional situations.

In Java, an **exception** is an object.

It represents a condition that prevent the program from proceeding normally.

Error situations **throw** an exception.

A programmer can added code **handle** the exception to continue execution.

If an exception is not handled, the program will terminate abnormally.

### **Example**

Consider this example:

package com.foothill;

import java.util.Scanner;

public class TestProgram {

public static void main(String[] args) {

Scanner input = new Scanner(System.in);

int num1, num2;

System.out.print("Enter 2 numbers: ");

num1 = input.nextInt();

num2 = input.nextInt();

System.out.println(num1+"/"+num2 +" is " + num1/num2);

}

}

If you run this program, you will see:

Enter 2 numbers: 7 0

Exception in thread "main" **java.lang.ArithmeticException**: / by zero

at com.foothill.TestProgram.main(TestProgram.java:12)

How can we fix this error without using exceptions?

Let's consider another example.

package com.foothill;

import java.util.Scanner;

public class TestProgram {

public static int readValue() {

Scanner input = new Scanner(System.in);

return input.nextInt();

}

public static void main(String[] args) {

int num1;

System.out.print("Enter a number: ");

num1 = readValue();

}

}

If you run this program, you will see:

Enter a number: abc

Exception in thread "main" **java.util.InputMismatchException**

at java.base/java.util.Scanner.throwFor(Scanner.java:939)

at java.base/java.util.Scanner.next(Scanner.java:1594)

at java.base/java.util.Scanner.nextInt(Scanner.java:2258)

at java.base/java.util.Scanner.nextInt(Scanner.java:2212)

at com.foothill.TestProgram.readValue(TestProgram.java:7)

at com.foothill.TestProgram.main(TestProgram.java:13)

You could fix this by changing readValue to:

public static int readValue() {

Scanner input = new Scanner(System.in);

if (!input.hasNextInt())

System.out.println("Bad data.");

System.exit(1);

}

return = input.nextInt();

}

Now the run will look as follows:

Enter a number: abc

Bad data.

But there is a problem.

The method should not get to terminate the program. The caller should decide if to terminate the program.

Alternatively we could (1) return another boolean from readValue() to say if the value is good and (2) add logic in main() to check this value. This adds major complication to our code.

Need a simple way to notify the caller that an exception has occurred.

### **Using Exceptions**

Java allows a method to throw an exception that can be caught and handled by the caller.

package com.foothill;

import java.util.Scanner;

**import java.util.InputMismatchException;**

public class TestProgram {

public static int readValue() {

Scanner input = new Scanner(System.in);

if (!input.hasNextInt())

**throw new** **InputMismatchException("Bad data.");**

return input.nextInt();

}

public static void main(String[] args) {

int num;

System.out.print("Enter a number: ");

**try {**

num = readValue();

**}**

**catch (InputMismatchException ex) {**

**System.out.println("Need to enter a number.");**

**}**

**System.out.println("Continue here...");**

}

}

The value thrown is an exception object. This is called **throwing an exception**.

Exception object is created from an exception class, in this case java.lang.InputMismatchException.

Constructor InputMismatchException(str) is called to construct the exception object.

The **try block** contains the code that is executed under normal conditions.

The **throw statement** interrupts normal execution.

The **catch block** catches the exception and is executed to **handle the exception**.

Afterwards, statements after the catch block are executed.

Output of running the above will look as below:

Enter a number: abc

Need to enter a number.

Continue here...

The identifier, ex, in the catch block is the exception object thrown.

The throw statement can be use directly in the try block or within a method called by the try block.

### **Advantages of Using Exceptions**

* Lets the caller decide whether to handle the exception or terminate. A called method (especially library method like str.split()) does not know what to do in case of an error.
* Avoids propagating back of error status.
* Separates detecting of the error from the handling of the error.

Consider the updated code:

package com.foothill;

import java.util.Scanner;

import java.util.InputMismatchException;

public class TestProgram {

public static int readValue() {

Scanner input = new Scanner(System.in);

if (!input.hasNextInt()) {

**input.nextLine();**

throw new InputMismatchException("Bad data.");

}

return input.nextInt();

}

public static void main(String[] args) {

int num = 0;

**boolean stop = false;**

**do {**

try {

System.out.print("Enter a number: ");

num = readValue();

**stop = true;**

}

catch (InputMismatchException ex) {

System.out.println("Need to enter a number. Try again.");

}

**} while (!stop);**

**System.out.println("Got "+num+"!");**

}

}

When run, it shows:

Enter a number: abc

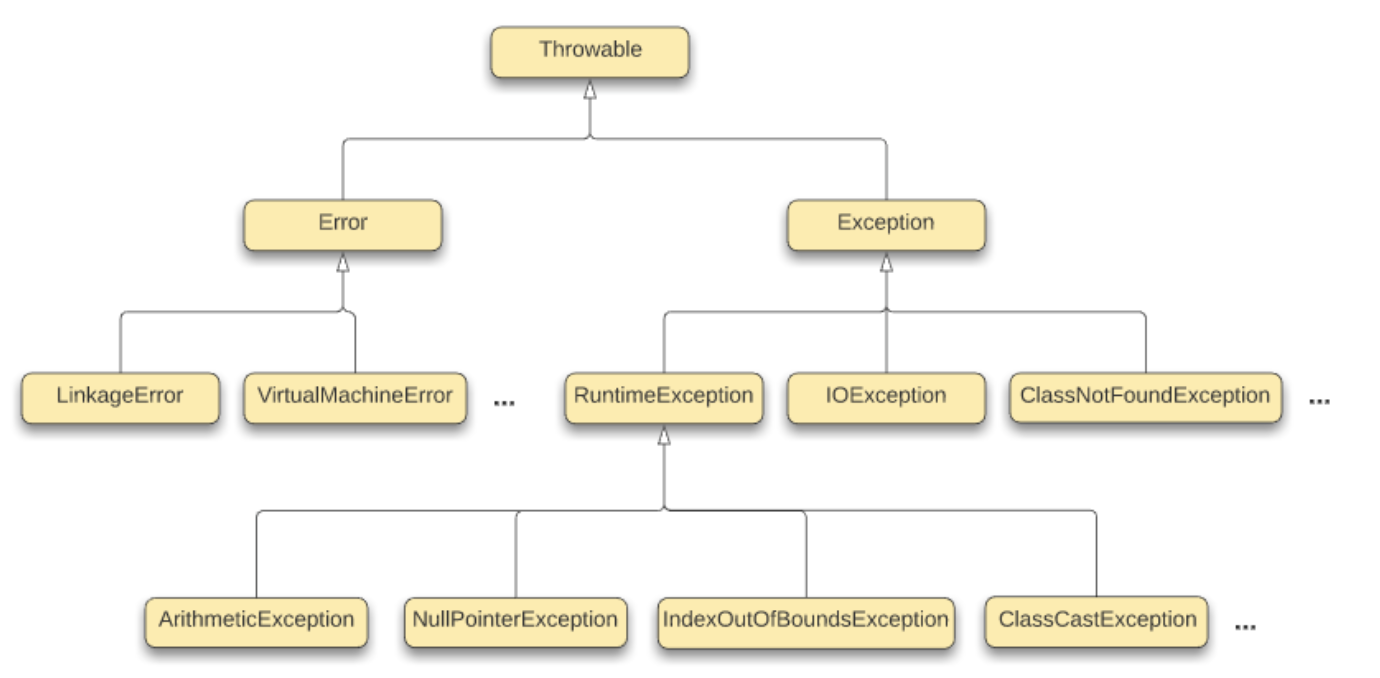
Need to enter a number. Try again.

Enter a number: 4

Got 4!

### **Existing Exception Types**

All exception objects have root class java.lang.Throwable.



There are 3 types of exception classes:

* System Errors
* Exceptions
* Runtime Exceptions

System Errors, represented by the Error class, are internal system errors thrown by the JVM. These are rare and not much can be done except terminating the program.

Exceptions, represented by the Exception class, are errors caused by your program and dependencies. These can be caught by your program. Examples are:

* ClassNotFoundException - You have seen this before. Where?
* IOException - Has subclasses FileNotFoundException, EOFException, etc.

Runtime Exceptions, represented by the RuntimeException class, are programming errors, such as array out-of-bounds, bad casting, etc. Examples are:

* ArithmeticException - Division by zero
* NullPointerException - Trying to use a reference variable that holds null
* IndexOutOfBoundsException - Array index out of range
* ClassCastException - Bad casting

**Class Exercise**: Write programs to cause these exceptions.

RuntimeException and Error are called **unchecked** exceptions.

Rest are called **checked** exceptions.

The compiler will forces you to deal with all checked exceptions in your program - in try/catch or declaring in a method header.

### **Exception Handling Model**

Three operations:

* Declaring an Exception
* Throwing an Exception
* Catching an Exception

void readData() **throws IOException, CustomException** {

...

**throw new IOException();**

...

}

void processInput() {

**try {**

readData();

**}**

**catch (IOException ex1) {**

System.out.println(...);

**}**

**catch (CustomException ex2) {**

**...**

**}**

**finally {**

**...**

**}**

}

A method must state the types of checked exceptions it might throw. Use the **throws** keyword followed by list of exception types (separated by commas) in the method header to declare exceptions.

To **throw** an exception, create an instance of the exception object. Then use the throw keyword followed by the exception object. Note the difference between throw vs. throws.

To catch exceptions, use a **try/catch block**. The try block is executed. If no exceptions occur, catch blocks is skipped. If an exception does occur in the try block, the rest of statements are skipped; first matching catch block is executed.

If no matching catch block is found, method is exited and the exception is passed to the calling method. In this way the exception propagates back through the chain of calling methods. If no matching catch block found, program terminates.

Exception in thread "main" java.util.InputMismatchException

at java.base/java.util.Scanner.throwFor(Scanner.java:939)

at java.base/java.util.Scanner.next(Scanner.java:1594)

at java.base/java.util.Scanner.nextInt(Scanner.java:2258)

at java.base/java.util.Scanner.nextInt(Scanner.java:2212)

at com.foothill.TestProgram.readValue(TestProgram.java:7)

at com.foothill.TestProgram.main(TestProgram.java:13)

The **finally** clause always executes (whether the exception occurs or not).

### **Exception Details**

Exception object provides methods to get information about the exception.

* getMessage() - Returns the message string
* toString() - Returns "<name of Exception> : <message string>"
* printStackTrace() - Prints call stack to console

Consider the program:

package com.foothill;

public class TestProgram {

public static void main(String[] args) {

try {

System.out.println(1/0);

}

catch (Exception ex) {

System.out.println("getMessage() gives: "+ex.getMessage());

System.out.println("toString() gives: "+ex.toString());

System.out.println("printStackTrace() gives:");

ex.printStackTrace();

}

}

}

It outputs:

getMessage() gives: **/ by zero**

toString() gives: **java.lang.ArithmeticException: / by zero**

printStackTrace() gives:

**java.lang.ArithmeticException: / by zero**

**at com.foothill.TestProgram.main(TestProgram.java:6)**

### **Defining Exceptions**

You can define your exception class by extending the Exception class.

public class InvalidRadiusException extends Exception {

private double radius;

// Constructor

public InvalidRadiusException(double r) {

super("Bad radius value " + r);

radius = r;

}

}

Then it can be used in the Circle class:

public class Circle {

private double radius;

// Constructors

public Circle() **throws InvalidRadiusException** {

this(1.0);

}

public Circle(double r) **throws InvalidRadiusException** {

**setRadius(r);**

}

// Methods

...

public void setRadius(double r) {

if (r >= 0) radius = r;

else **throw new InvalidRadiusException(r);**

}

...

}

public class MyProgram {

public static void main(Strings[] args) {

**try {**

Circle circle1 = new Circle(25);

Circle circle2 = new Circle(-5);

Circle circle3 = new Circle(100);

**}**

**catch (InvalidRadiusException ex) {**

**System.out.println(ex);**

**}**

}

}