Lecture - Exceptions & Error Handling

Introduction to exception handling

Sometimes the best outcome can be when nothing unusual happens.

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error.

> **Definition: An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.**

However, the case where exceptional things happen must also be prepared for. Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur.

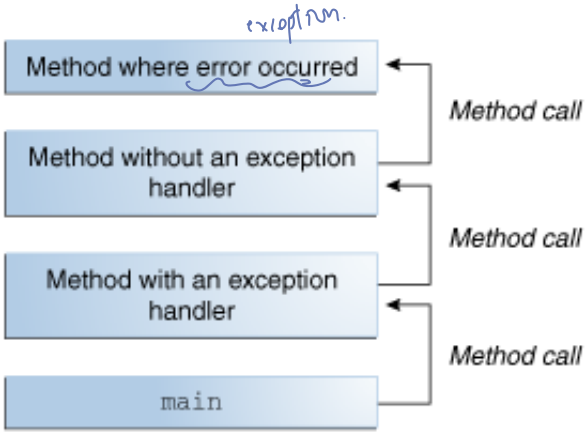
Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens. This is called ***throwing an exception***.

In another place in the program, the programmer must provide code that deals with the exceptional case. This is called ***handling the exception***.

How Exceptions work?

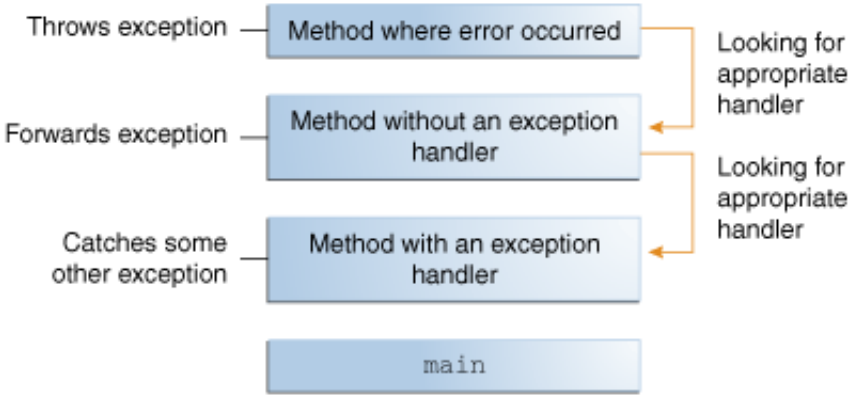
When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack as shown in the figure.



The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.



try-throw-catch

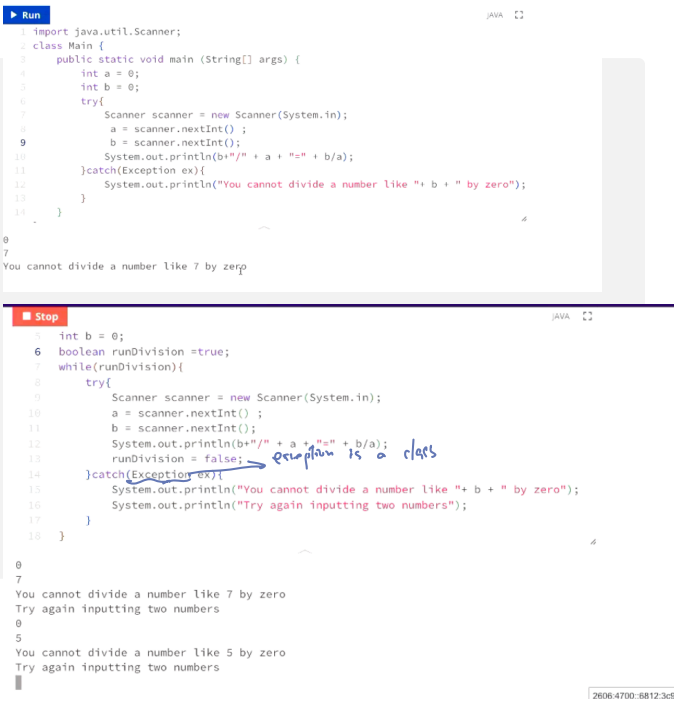
The basic way of handling exceptions in Java consists of the `try-catch` mechanism.

`try`

The `try` block contains the code for the basic algorithm. It tells what to do when everything goes smoothly.

It is called a `try` block because it "tries" to execute the case where all goes as planned. It can also contain code that throws an exception if something unusual happens, and is caught by the `catch` block.

```
class Main {
    public static void main (String[] args) {
        // . . .
        try{
            // Something that may fail
        }catch(Exception e){
            String message = e.getMessage();
            System.out.println("Exception: " + message);
            System.exit(1);
        }
        // ...
    }
}
```



`catch`

The code inside the try block is executed as normal. If something goes wrong in any of the standard library methods inside the block, an exception will be generated. When this happens, the execution of the try block stops, and the catch block is executed. The variable `e` contains information about the error that occurred.

This `e` is called the **catch block parameter**. It does two things:

- It specifies the type of thrown exception object that the catch block can catch (e.g., an Exception class object above).
- It provides a name (for the thrown object that is caught) on which it can operate in the catch block. Note: The identifier `e` is often used by convention, but any non-keyword identifier can be used.

The catch block has only one parameter. A catch block looks like a method definition that has a parameter of type `Exception` class, but it is not really a method definition.

In java.lang file it's automatic import
But, if the specific exception is not in the java.lang
you need to import yourself.

a specific error message.

ArithmeticException.

```
import java.util.Scanner;
import java.utoInputMismatchException
class Main {
    public static void main (String[] args) {
        int a = 0;
        int b = 0;


            try{
                Scanner scanner = new Scanner(System.in);
                a = scanner.nextInt() ;
                b = scanner.nextInt();
                System.out.println(b+"/" + a + "=" + b/a);
            }catch(ArithmeticException ex){
                System.out.println("You cannot divide a number like "+ b + " by zer

            }catch(InputMismatchException ex){
                System.out.println("You have input something other than an integer"
            }catch(Exception ex){ => all exception parents class
                System.out.println("Some unknown exception occurred" + ex.getMessag
            }

        System.out.println("Terminating program");
```

exceptions have hierarchy

Only one exception will be process each time.

→ Be careful to put the most general exception at the bottom

```
class Main {
    public static void main (String[] args) {
        // . . .
        try{
            Calculator calc = new Calculator();
            calc.divide(5,0);
        }catch(Exception e){
            String message = e.getMessage();
            System.out.println("Exception: " + message);
            System.exit(1);
        }
        // ...
    }
}

class Calculator{

    public int divide( int number,int divisor){
        return number/divisor;
    }
}
```

=) Exception : / by zero.

```
        Calculator calc = new Calculator();
        calc.divide(5,0);

    // ...
    }
}

class Calculator{

    public int divide( int number,int divisor) throws ArithmeticException{
        return number/divisor;
    }
}
```

not handle exception in this method.

```
    }catch(ArithmeticException e){
        String message = e.getMessage();
        System.out.println("Exception: " + message);
        System.exit(1);
    }
    // ...
}
}

class Calculator{

    public int divide( int number,int divisor) throws ArithmeticException{
        return number/divisor;
    }
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Calculator.divide(Main.java:15)
        at Main.main(Main.java:6)
```

The execution of the catch block is called catching the exception, or handling the exception; the catch block is called an exception handler

In some cases, it is possible to recover from an exception. For example, if it was caused by a bad choice by the user then it is possible to ask for another choice. In some cases (but rarely in production software), the best action is to tidy everything up and exit gracefully, as done above.

Exercise: Put some code in the try block above that you know will compile but fail at runtime. Perhaps base it on trouble you had in Assignment 1. If you had no troubles (you wish!) then you can try reading beyond the end of System.in with a Scanner.

`throw`

Often, you will want to report a problem, even if none of the library routines throws an exception. This is done by the throw command.

```
class Main {
    public static void main (String[] args) {
        // . . .
        try{
            //. . .
            throw new Exception("StringArgument");
            //. . .
        }catch(Exception e){
            String message = e.getMessage();
            System.out.println(message);
            System.exit(1);
        }
        ///. . .
    }
}
```

When the throw statement occurs, the `try` block stops.

A throw statement is similar to a method call:

```
throw new ExceptionClassName(SomeString);
```

In the above example, the object of class `ExceptionClassName` is created using a string as its argument. This object, which is an argument to the throw operator, is the exception object thrown.

Instead of calling a method, a `throw` statement calls a `catch` block.

Whenever an exception is thrown, it should ultimately be handled (or caught) by some `catch` block.

Recall that the catch block only takes one parameter, typically an `Exception`. The above code only passes a string to the Exception constructor. That is enough to print an error message and exit, but

not enough to allow the code to try to recover. Using polymorphism, you can create a class *derived* from `Exception`, to contain as much additional information as you like.

`try-throw-catch` mechanism

When a try block is executed, two things can happen:

1. No exception is thrown in the `try` block.
   – The code in the try block is executed to the end of the block
   – The `catch` block is skipped
   – The execution continues with the code placed after the catch block
2. An exception is thrown in the `try` block and caught in the catch block
   – The rest of the code in the `try` block is skipped
   – Control is transferred to a following `catch` block (in simple cases)
   – The thrown object is plugged in for the catch block parameter
   – The code in the `catch` block is executed
   – The code that follows that `catch` block is executed (if any)

Using the `getMessage` Method

Every exception has a `String` instance variable that contains some message. This string typically identifies the reason for the exception.

In the previous example, "StringArgument" is an argument to the `Exception` constructor.

This is the string used for the value of the string instance variable of exception `e`. Therefore, the method call `e.getMessage()` returns this string.

Using the `printStackTrace` Method

The `printStackTrace()` method in Java is used to handle exceptions and errors. It is a method of Java's throwable class which prints the throwable along with other details like the line number and class name where the exception occurred. It is very useful in diagnosing exceptions. For example, if one out of five methods in your code cause an exception, `printStackTrace()` will pinpoint the exact line in which the method raised the exception.

Exception classes

There are more exception classes than just the single class `Exception`.

- There are more exception classes in the standard Java libraries
- New exception classes can be defined like any other class

All predefined exception classes have the following properties:

- There is a constructor that takes a single argument of type `String`.
- The class has an accessor method `getMessage` that can recover the string given as an argument to the constructor when the exception object was created.

All programmer-defined exception classes must be derived from the class `Exception` its descendants.

Exception Classes from Standard Packages

The predefined exception class `Exception` is the root class for all exceptions. Every exception class is a descendent class of the class `Exception`.

Although the `Exception` class can be used directly in a class or program, it is most often used to define a
derived class.

The class Exception is in the java.lang package, and so requires no import statement.

Numerous predefined exception classes are included in the standard packages that come with Java. For example:

- IOException
- NoSuchMethodException
- FileNotFoundException

Many exception classes must be imported in order to use them.

```
import java.io.IOException;
```

Defining Exception Classes

A `throw` statement can throw an exception object of any exception class.

Instead of using a predefined class, exception classes can be programmer-defined
– These can be tailored to carry the precise kinds of information needed in the catch block

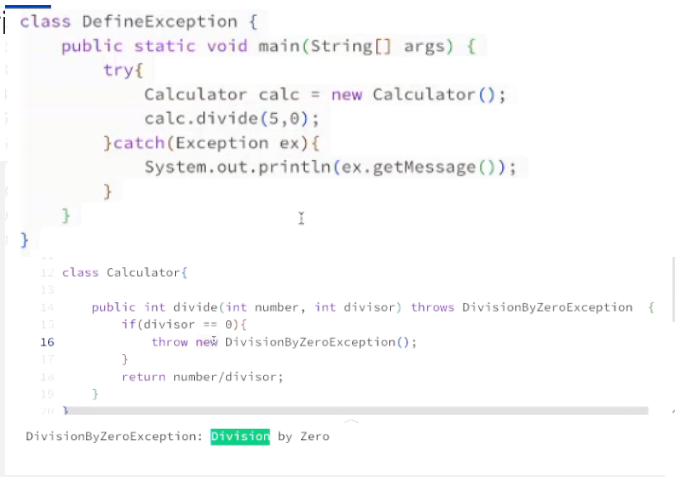
– A different type of exception can be defined to identify each different exceptional situation

Every exception class to be defined must be a derived class of some already defined exception class
– It can be a derived class of any exception class in the standard Java libraries, or of any programmer defined exception class

Constructors are the most important members to define in an exception class
– They must behave appropriately with respect to the variables and methods inherited from the base class
– Often, there are no other members, except those inheri

The following exception class performs these basic tasks

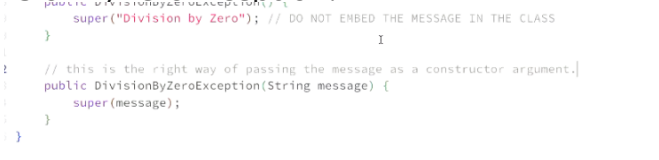
```
class DefineException {
    public static void main(String[] args) {

    }
}

class DivisionByZeroException extends Exception {
    public DivisionByZeroException() {
        super("Division by Zero");
    }

    public DivisionByZeroException(String message) {
        super(message);
    }
}
```



bad way to do this

the right way to write

Exercise: Write Main.divide method that will take two numbers a and b, and return a/b if b is non-zero. If b is zero, it should throw (and catch) a DivisionByZeroException.



Other message types

An exception class constructor can be defined that takes an argument of any other type. It would store its value in an instance variable. It would need to define accessor methods for this instance variable.

The following exception class takes an int message.

```
class Main {
    public static void main(String[] args) {

    }
}

class BadNumberException extends Exception {
    private int badNumber;

    // Accessor
```

multiple exception throws by methods.

```
class DefineException {
    public static void main(String[] args) {
        try{
        Calculator calc = new Calculator();
        calc.divide();
        }catch(InputMismatchException ex){
            System.out.println(ex.getMessage());
        }catch(ArithmeticException ex){
            System.out.println(ex.getMessage());
        }catch(Exception ex){
            System.out.println(ex.getMessage());
        }
```

```
class Calculator{

    public int divide() throws InputMismatchException, ArithmeticException{
        Scanner scanner = new Scanner(System.in);
        int number = scanner.nextInt();
        int divisor = scanner.nextInt();
        return number/divisor;
    }
}

class DivisionByZeroException extends Exception {
    public DivisionByZeroException() {
        super("Division by Zero"); // DO NOT EMBED THE MESSAGE IN THE CLASS
```

```
class DefineException {
    public static void main(String[] args) {
        try{
        Calculator calc = new Calculator();
        calc.divide();
        }catch(InputMismatchException| ArithmeticException ex){
            System.out.println(ex.getStackTrace());
        }
    }
}

class Calculator{
```

get where the series of line that cause exception.

you can also write in this way but the Exception class should be write in another catch block.

```
    public int getBadNumber () {
        return badNumber;
    }

    // new constructor taking integer
    public BadNumberException (int number) {
        super("Bad number");
        badNumber = number;
    }

    // Standard constructors
    public BadNumberException () {
        super ("Bad number");
        badNumber = -1;
    }

    public BadNumberException (String message) {
        super(message);
        badNumber = -1;
    }

}
```

Exception object characteristics

The two most important things about an exception object are its type (i.e., exception class) and the message it carries.
– The message is sent along with the exception object as an instance variable.
– This message can be recovered with the accessor method `getMessage`, so that the catch block can use the message.

Programmer-Defined Exception Class Guidelines

A programmer-defined exception class must be a derived class of an already existing exception class.

The exception class should allow for the fact that the method `getMessage` is inherited. For all predefined exception classes, `getMessage` returns the string that is passed to its constructor as an argument, or it will return a default string if no argument is used with the constructor.

At least two constructors should be defined:

- A constructor that takes a string argument and begins with a call to super, which takes the string argument.
- A no-argument constructor that includes a call to super with a default string as the argument.

Often more constructors will be provided, to pass additional information.

Multiple catch blocks

A try block can potentially throw any number of exception values, and they can be of differing types.

In any one execution of a try block, at most one exception can be thrown (since a throw statement ends the execution of the try block). However, different types of exception values can be thrown on different executions of the try block.

A catch block can only catch values of the exception class type given in the catch block heading.

Different types of exceptions can be caught by placing **more than one** `catch` block after a `try` block.

Any number of catch blocks can be included, but they must be placed in the correct order.

When an exception is thrown in a try block, the catch blocks are examined in order. The first one that matches the type of the exception thrown is the one that is executed.

Catch the more specific exception first (i.e., catch a descendant class before an ancestor).

Exercise: Fix the following code.

```
class MultipleTryCatch {
    public static void main (String[] args) {
        try {
            double a = 1.0/0.0;
        }catch (Exception e) {
            System.out.println("Try again");
        }catch (ArithmeticException e) {
            System.out.println("Well done");
        }
    }
}
```

Because an ArithmeticException is a descendant of Exception, all ArithmeticException will be caught by the first catch block before ever reaching the second block. The catch block for ArithmeticException will never be used. (Ed refuses even to compile this code.)

To correct the code, simply reverse the two blocks.

Throwing exceptions from methods

The greatest value of exceptions is when we don't know enough of the context to be able to handle the situation. For example, if we divide by 0, should we replace the answer by a very large number and continue, or should we tell the calling method to choose a different set of parameters and call again?

Because of this, it is common to want to `throw` an exception in a method, but not `catch` it in the same method.

In such cases, the program using the method should enclose the method invocation in a `try` block, and `catch` the exception in a catch block that follows. The method that throws the exception would not surround the `throw` by `try` and `catch` blocks.

However, the method that throws without catching has to include a *throws clause* in its header.

```
class Main {
    static private int dangerMethod () throws Exception {
        throw new Exception ("example");
    }

    static public void caller () {
        int a;
        try {
            a = dangerMethod ();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    static public void caller1 () throws Exception {
        int a = dangerMethod ();
    }

    static public void main (String[] args) {
        try {
            caller();
        } catch (Exception e) {
            System.out.println ("Main caught from caller");
        }

        try {
            caller1();
        } catch (Exception e) {
            System.out.println ("Main caught from caller1");
        }
    }
}
```

The reason is that, if a method can throw an exception but does not catch it, it must provide a warning to the callers. The process of including an exception class in a throws clause is called *declaring the exception*. In the above example, any method that calls `dangerMethod` (in this case `caller` and `caller1`) must deal with the exception. It can either catch it itself as caller does, or declare that it may throw it as caller1 does.

Exercise: Run the code and trace where the exceptions flow.

Exercise: Remove the throws clause. What happens?

Exercise: Try removing different `try`/`catch` pairs and see how the behaviour changes.

If a method throws an exception and does not catch it, then the method invocation ends immediately. (Well, Almost immediately; we'll meet the `finally` block in the next slide.)

If a method can throw more than one type of exception, then separate the exception types by commas:

```
public void aMethod() throws AnException, AnotherException
```

The catch or declare rule: An exception must be handled somewhere

An exception can be handled in a catch block.

Alternatively, declaring the exception shifts the handling responsibility to the method that invoked the exception-throwing method. The invoking method must then handle the exception, unless it too uses the same technique to "pass the buck".

One of these two must be done. Ultimately, every exception that is thrown should eventually be caught by a catch block in some method that does not just declare the exception class in a throws clause.

The two techniques can be mixed. Some exceptions may be caught, and others may be declared in a `throws` clause. However, these techniques must be used consistently with a given exception.

- If an exception is not declared, then it must be handled within the method.
- If an exception is declared, then the responsibility for handling it is shifted to some other calling method.

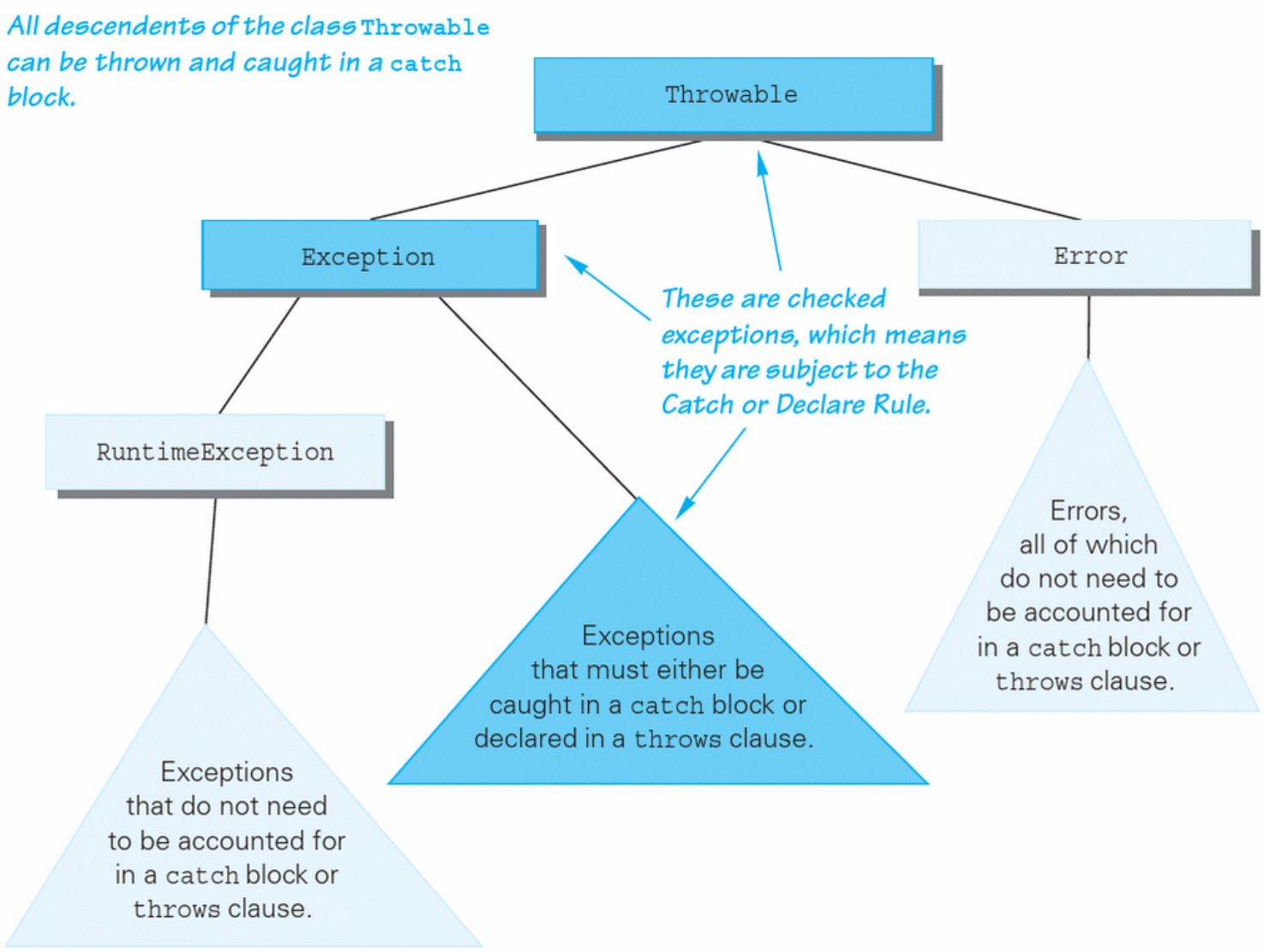
Note that if a method definition encloses an invocation of a second method, and the second method can throw an exception and does not catch it, then the first method must catch or declare it.

Advanced topics (not examinable)

Hierarchy of throwable objects

The try/throw/catch framework can apply to objects other than exceptions.

Display 9.11 Hierarchy of `Throwable` Objects



Most importantly, not all exceptions are subject to the "catch or declare" (also called catch & specify) rule. Ones that are called *checked* exceptions.

The compiler checks to see if they are accounted for with either a catch block or a throws clause.

The classes `Throwable`, `Exception`, and all descendants of the class `Exception` (other than `RuntimeException` and its descendants) are checked exceptions.

Descendants of `RuntimeException` are *unchecked* exceptions.

The class `Error` and all its descendant classes are called *error classes*. They are *not* subject to the "catch or declare" rule.

Exercise: The following is the example from the previous slide. Replace Exception with `ArithmeticException`, and remove the `throws` clauses. Note that it now compiles, runs, and crashes out with an error. Can you guess what base type `ArithmeticException` may be derived from?

```
class ExceptionDemo {
    static private int dangerMethod () throws Exception {
        throw new Exception ("example");
    }

    static public void caller () {
        int anInt;
        try {
            anInt = dangerMethod ();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    static public void caller1 () throws Exception {
        int anotherInt = dangerMethod ();
    }

    static public void main (String[] args) {
        try {
            caller();
        } catch (Exception e) {
            System.out.println ("Main caught from caller");
        }

        try {
            caller1();
        } catch (Exception e) {
            System.out.println ("Main caught from caller1");
        }
    }
}
```

When a method in a derived class is overridden, it should have the same exception classes listed in its throws clause that it had in the base class, or it should have a *subset* of them. That is, a derived class may not add any exceptions to the throws clause, but it can delete some.

- This is because an object of a derived class must be able to be used any way its parent class can. It is the same as the reason that derived class cannot convert a method from public to private.

Examples

- When a program contains an assertion check, and the assertion check fails, an object of the class `AssertionError` is thrown. This causes the program to end with an error message.

  The class `AssertionError` is derived from the class `Error`, and therefore is an unchecked Throwable. In order to prevent the program from ending, it could be handled, but this is not required.
- The Scanner class can throw the `InputMismatchException`. The `nextInt` method of the `Scanner` class can be used to read `int` values from the keyboard.

  However, if a user enters something other than a well-formed `int` value, an `InputMismatchException` will be thrown. Unless this exception is caught, the program will end with an error message. If the exception is caught, the `catch` block can give code for some alternative action, such as asking the user to reenter the input.

  The InputMismatchException is in the standard Java package `java.util`
  A program that refers to it must use an `import` statement, such as the following:
  `import java.util.InputMismatchException;`
  It is a descendent class of RuntimeException. Therefore, it is an unchecked exception and does not
  have to be caught in a catch block or declared in a throws clause.
- An `ArrayIndexOutOfBoundsException` is thrown whenever a program attempts to use an array index that is out of bounds (below 0 or above the length minus 1). This normally causes the program to end.
  Like all other descendants of the class `RuntimeException`, it is an unchecked exception. There is no requirement to handle it.
  When this exception is thrown, it is nearly always an indication that the program contains an error. Instead of attempting to handle the exception, the program should simply be fixed. (A possible exception would be in a class like ArrayList in which it may signal that the underlying array needs to grow. However, it is typically better to check for that explicitly.)

What happens if an exception is never caught?

If every method up to and including the main method simply includes a `throws` clause for an exception, that exception may be thrown but never caught

In a GUI program (i.e., a program with a windowing interface), nothing happens - but the user may be left in an unexplained situation, and the program may be no longer be reliable.

In non-GUI programs, this causes the program to terminate with an error message giving the name of the exception class.

Every well-written program should eventually catch every exception by a `catch` block in some method.

When to use exceptions

There is overhead in using exceptions, even if the exception is not thrown. The `try` itself slightly increases run time.

Exceptions should be reserved for situations where a method encounters an unusual or unexpected case that cannot be handled easily in some other way. (In other languages such as Python, exceptions are used more freely.)

How exceptions are handled depends on how a method is called.

Event driven programming

Exception handling is an example of a programming methodology known as *event-driven programming*. When using event-driven programming, objects are defined so that they send events to other objects that handle the events. An event is also an object. Sending an event is called *firing an event*.

In exception handling, the event objects are the exception objects. They are fired (thrown) by an object when the object invokes a method that throws the exception. An exception event is sent to a catch block, where it is handled.

Another important type of event driven programming is writing GUI applications. There, events are typically triggered by a user action, such as a mouse movement, mouse click, or key press. The handlers for these events can last a long time, such as recalculating a spreadsheet.

Nested `try`-`catch` blocks

It is possible to place a `try` block and its following `catch` blocks inside a larger `try` block, or inside a larger `catch` block.

- If a set of `try`-`catch` blocks are placed inside a larger `catch` block, different names must be used for the `catch` block parameters in the inner and outer blocks, just like any other set of nested blocks
- If a set of `try`-`catch` blocks are placed inside a larger `try` block, and an exception is thrown in the inner `try` block that is not caught, then the exception is thrown to the outer `try` block for processing, and may be caught in one of its `catch` blocks

The `finally` block

Consider the following code. It contains (a toy representation of) a *lock* which says that an object of this class can only perform a function if no other object is performing it. They are useful for multi-threaded programming. For this example, it is just something that we need to make sure we release at the end of `dangerousMethod`.

```
class Main {
    static int lock;

    static void dangerousMethod () throws Exception {
        lock++;     // lock a resource to prevent access
        try {
            throw new Exception ("not caught");
        } catch (ExceptionA e) {
            // catch only one type of exception
        }
        lock--;     // free up lock
    }

    static public void main (String[] args) {
        try {
            dangerousMethod ();
        } catch (Exception e) {

        }
        System.out.println ("Lock is " + lock);
    }
}

class ExceptionA extends Exception {

}
```

When you run it, you will notice that it ends with Lock==1. That is, throwing the exception caused the lock not to be released.

This represents a common problem: the there is some "tidying up" that has to be done whether or not exception is thrown from this method.

Java provides a way of doing this clean-up: a "finally" block that can come *after* all the "catch" blocks. The code in the catch block is run either after the try block completes successfully, or after one of the catch blocks runs, or **after an uncaught exception is thrown and before the method exits**.

Once again, if the try-catch-finally blocks are inside a method definition, there are three possibilities when the code is run:

- The try block runs to the end, no exception is thrown, and the finally block is executed
- An exception is thrown in the try block, caught in one of the catch blocks, and the finally block is executed
- An exception is thrown in the try block, there is no matching catch block in the method, the finally block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method

Exercise: Replace the `lock--;` in dangerousMethod above by the block `finally { lock--; }`. How does the output change?

Exception controlled loops

Sometimes it is better to simply loop through an action again when an exception is thrown, as follows:

```
boolean done = false;
while (! done)
{
    try
    {
        // CodeThatMayThrowAnException
        done = true;
    }
    catch (SomeExceptionClass e)
    {
        // SomeMoreCode
    }
}
```

Exercise: Read and understand the following code. Try it with inputs "forty", "=1", "10.5", "10,5", "10/5", "10". Before you try each, guess whether or not `nextInt()` will succeed.

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

class InputDemo {
    public static void main (String[] args) {
        Scanner keyboard = new Scanner(System.in);
        int number = 0;     // initialize to keep the compiler happy
        boolean done = false;
        while (!done) {
            try {
                System.out.println("Enter a whole number:");
                number = keyboard.nextInt();    // may throw
                done = true;                    // only if nextInt succeeded
            } catch (InputMismatchException e) {
                keyboard.nextLine();
                System.out.println("That was not a correctly written whole number.");
                System.out.println("Try again.");
            }
        }
        System.out.println("You entered " + number);
    }
}
```

Program exit values

You will notice that `System.exit()` takes an argument. This argument is passed to the operating system when the program finishes. If the program is being run from a script, the script will interpret an exit value of 0 as "success", or `true`, and a non-zero value as "failure" or `false`. If the program

exits because of an error, it should pass a non-zero argument to `exit`. The text book doesn't follow this convention, but you should.

ExceptionDemo

This simple program has not handled any exceptions yet. Please identify all potential exceptions and use the "try...catch" statement to capture them.

```
import java.util.Scanner;

public class ExceptionDemo {

  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);

    int[] arr = {1, 2, 3, 4, 5, 6, 0};
    System.out.println("Array index: ");
    int index = keyboard.nextInt();

    int value = arr[index];
    System.out.printf("arr[%d] = %d\n", index, value);

    keyboard.close();
  }
}
```

Extension: Define a new customized exception class and throw it instead of the predefined exception class when something wrong happens (e.g., the array index is out of bound).

Extension: Update the program to allow user to repeatedly provide a valid array index until they are successfully.

Additional Readings

1. WALTER, S. Absolute Java, Global Edition. [Harlow]: Pearson, 2016. (Chapter 9)
2. SCHILDT, H. Java: The Complete Reference, 12th Edition: McGraw-Hill, 2022 (Chapter 10)
3. Java Exceptions (accessible on 14-02-2024) Oracle's Java Documentation. Available at: https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html.