ESS 201 - Programming II (Java)
Term 1, 2019-20

Exception Handling

T K Srikanth
International Institute of Information Technology, Bangalore

```
public class Test {
    public static float findRatio(int x, int y) {
    return x/y;
    }

    public static void main(String[] args) {
    int a, b;
    // read in values of a,b as integers each time from input
        ...
    System.out.println(findRatio(a, b));
    }
}
```

What can go wrong in this code?? And what corrective action could we take?

Possible points of failure

1. User could enter a 0 for b (divide by zero)
2. User could enter a non-integer as input for either value
3. If reading from a file, could have problems opening file or reach end-of-file

How do you structure code so that all such situations are detected, and the system recovers gracefully (to the extent it can) from these errors.

For example, in this example, in each of the above errors, could ask to user to correct the input and try again...

```
public class Test {
        public static float findRatio(int x, int y) {
        If (y != 0)
                return x/y;
        else
                // what should we return???
                return ….
        }
```

Check for incorrect input here

```
public static void main(String[] args) {
        int a, b;
        // read in values of a,b as integer
        ….

        // check for 0 input here:
        if( b != 0)
                System.out.println(findRatio(a, b));
        else
                System.out.println(“bad input - try
        again”);
}
}
```

… Or here. Which is better??

Other errors

What is the best way to catch other errors:

- Other bad input (e.g. String instead of int)
- Wrong file name (if input from a file)

Safe coding practice requires we check for every possible error and “manage” it well - graceful recovery if possible

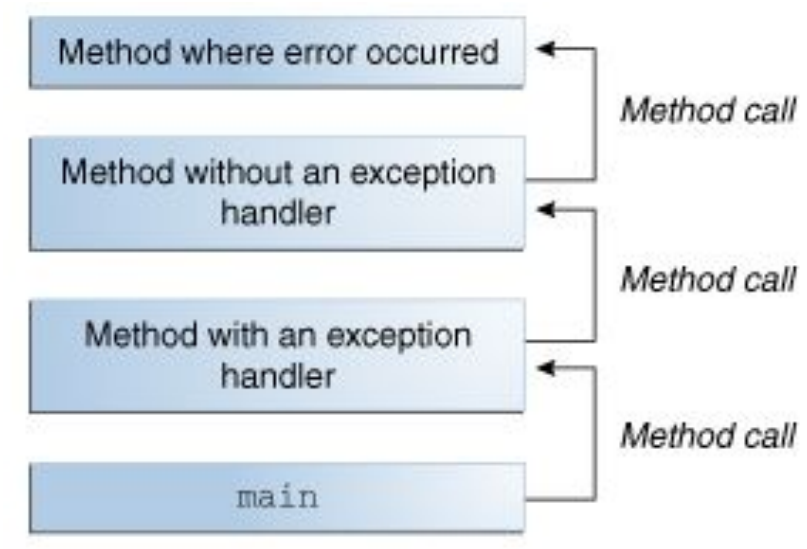
But adding such error-checking code throughout the program can make the code very difficult to follow, and can itself introduce logic errors - creates too many branches in the code. Can make loops, returns etc more complicated, as well as make it difficult to cleanup consistently.

Exception Handling

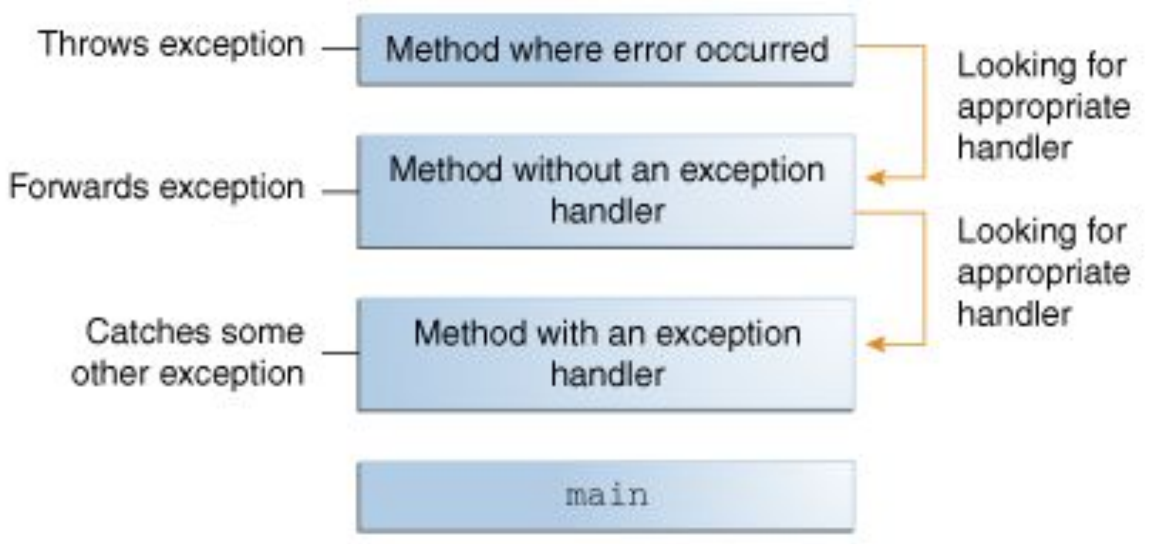
A framework for managing errors/exceptions

An Exception object encapsulates various information about an error condition. Created when the situation is detected, and “returned” to the calling method by a parallel path - called “throw”ing an exception

Call stack


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

Exception handling - stack unwinding


Throws exception
Method where error occurred
Looking for appropriate handler
Forwards exception
Method without an exception handler
Looking for appropriate handler
Catches some other exception
Method with an exception handler
main

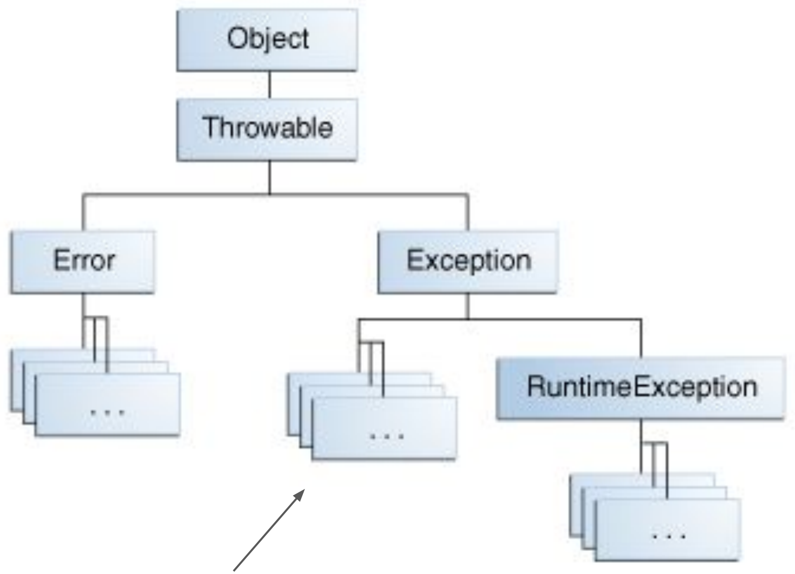
Exceptions

Base class: Exception

Example sub-types:

1. User could enter a 0 for b (divide by zero) ⇒ ArithmeticException
2. User could enter a non-integer as input for either value ⇒ InputMismatchException
3. If reading from a file, could have problems opening file or reach end-of-file ⇒ IOException

Checked vs Unchecked exceptions



Checked exceptions

try-catch

Implement a “catch” block

```
void method1() {
        try {
                // calls a method that throws checked exception
        } catch (Exception e) {
                // do something with the exception
        }
}
```

catch blocks

```
try {
   … calls methods that might throw exceptions
} catch (IndexOutOfBoundsException e) {
    System.err.println("out of bounds " + e.getMessage());
} catch (IOException e) {
    System.err.println("io error: " + e.getMessage());
}
```

If an exception is thrown from inside the try block, the **first catch block** that matches the thrown execution is executed.

catch-or-specify

Code that calls a method that might thrown a checked exception should either:

- Have a try block that catches that exception, and/or
- Throw that same exception itself

```
void method1() throws IOException {
    // call a method that throws IOException

    }
```

finally block

The finally block is executed when the try block exits or when the catch block exits

```
BufferedReader br = new BufferedReader(new FileReader(path));
try {
    return br.readLine();
} catch (IOException e) {
  System.err.println("io error: " + e.getMessage());
} finally {
  if (br != null) br.close();
}
```

Good practice. Ensures that the file is always closed.