# Exceptions

# Exceptions represent errors in Java programs

Named *“exceptions"* because they are expected to occur rarely[1]

In Java, the term refers to **two things**:

- The **objects** that represent errors that occur at runtime
- The **classes** defining these objects, which represent kinds of errors

When an error is detected, an exception is **thrown**

Implicitly (by Java or some library) or explicitly (by your program)

Thrown exceptions are expected to be **handled**

[1] John B. Goodenough. Exception Handling: Issues and a Proposed Notation. Commun. ACM 18(12): 683-696 (1975)

```
void m1() {
    ...
    try {
        ... // does things
        throw new SomeException(“Something went wrong.”);
        ... // does more things, if the exception is not thrown
    } catch (SomeException e) {
        log(e);
        ... // do something else
    }
    ...
}
```

```
void m1() {
    ...
    try {
        ... // does things
        throw new SomeException(“Something went wrong.”);
        ... // does more things, if the exception is not thrown
    } catch (SomeException e) {
        log(e);
        ... // do something else
    }
    ...
}
```

```
void m1() {
    ...
    try {
        ... // does things
        throw new SomeException(“Something went wrong.”);
        ... // does more things, if the exception is not thrown
    } catch (SomeException e) {
        log(e);
        ... // do something else
    }
    ...
}
```

```
void m1() {
    ...
    try {
        ... // does things
        throw new SomeException(“Something went wrong.”);
        ... // does more things, if the exception is not thrown
    } catch (SomeException e) {
        log(e);
        ... // do something else
    }
    ...
}
```

```
void m1() {
    ...
    try {
        ... // does things
        throw new SomeException(“Something went wrong.”);
        ... // does more things, if the exception is not thrown
    } catch (SomeException e) {
        log(e);
        ... // do something else
    }
    ...
}
```

# Exception type hierarchy

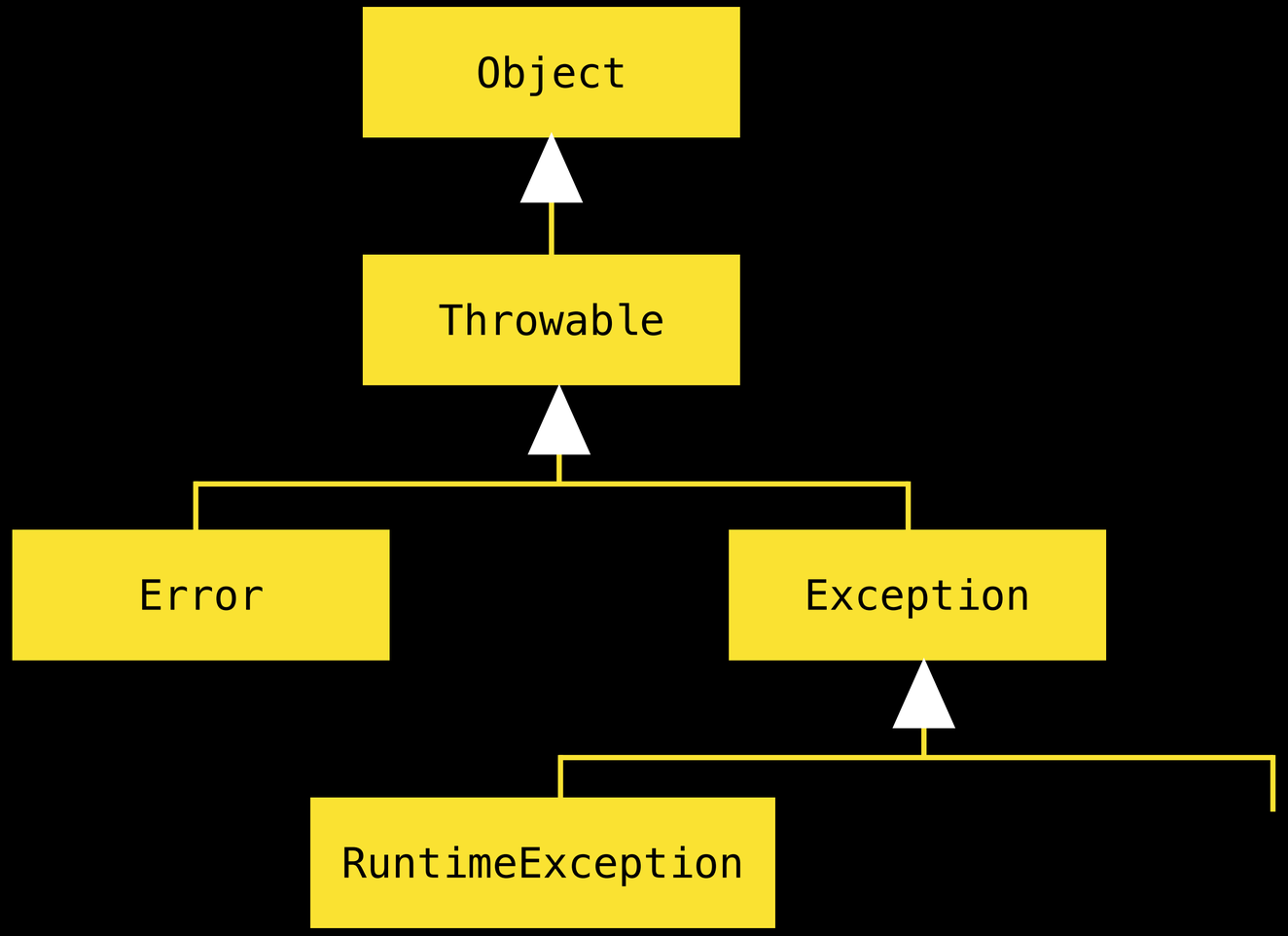
Types of objects that may represent erroneous situations

In most practical scenarios, you'll be using

`Exception`, or

`RuntimeException`

or one of their subtypes



# Exception type hierarchy

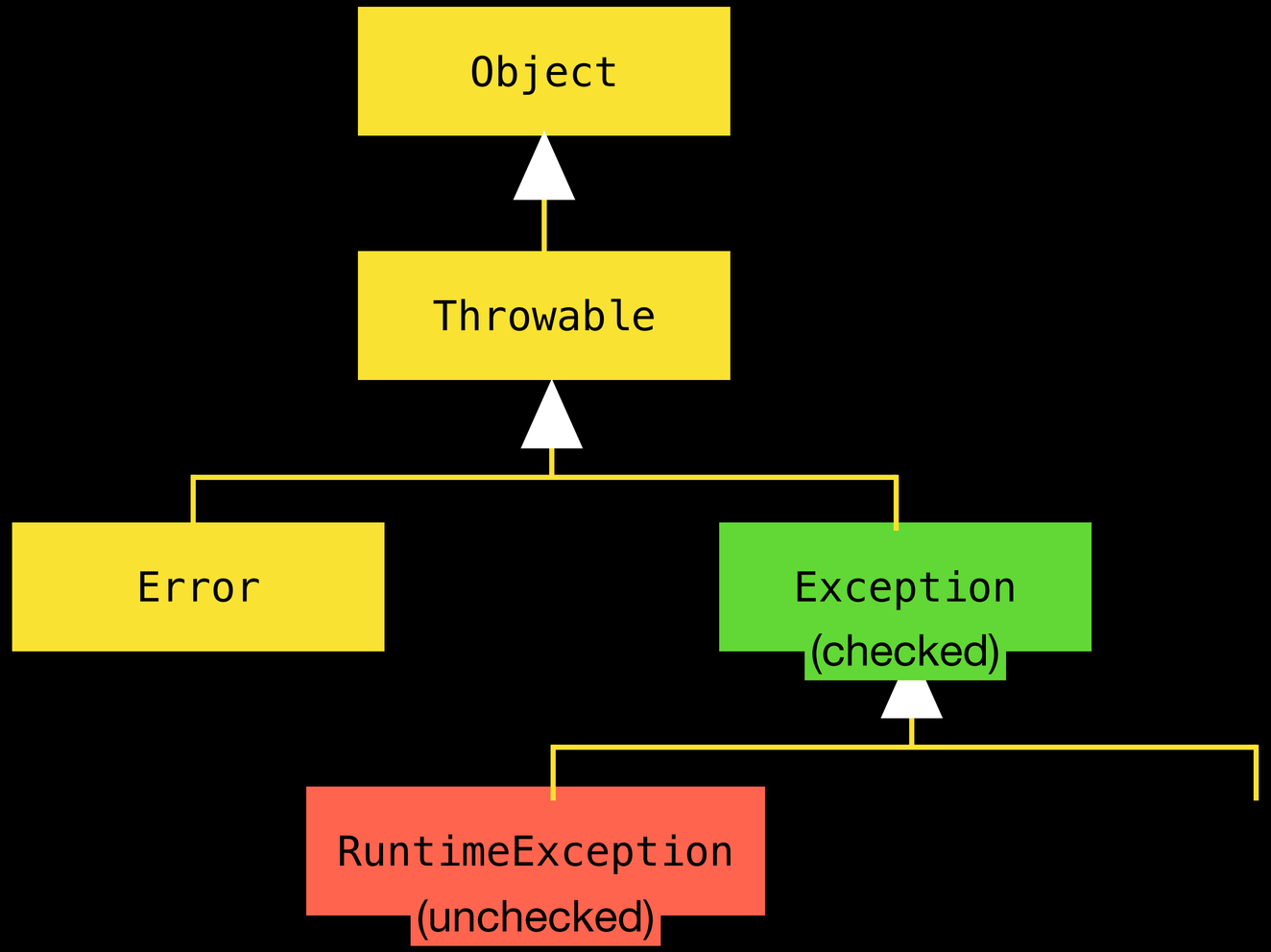
Types of objects that may represent erroneous situations

In most practical scenarios, you'll be using

`Exception`, or

`RuntimeException`

or one of their subtypes



Code that may throw checked exceptions must either

- indicate this explicitly, or
- handle them

# Using checked exceptions

```
public static void main(String[] args) {
    String[] files = { "file1.txt", "file2.txt", "file3.txt" };
    String[] results;

    try {
        // Call the function and handle exceptions here
        results = readTextFiles(files);

        // Print contents of each file
        for (int i = 0; i < results.length; i++) {
            System.out.println("Contents of " + files[i] + ":");
            System.out.println(results[i]);
            System.out.println("------------");
        }

    } catch (IOException e) {
        // Handle any file-related exceptions here
        System.err.println("Error reading files: " + e.getMessage());
    }
}
```

# Using checked exceptions

```
public static void main(String[] args) {
    String[] files = { "file1.txt", "file2.txt", "file3.txt" };
    String[] results;

    try {
        // Call the function and handle exceptions here
        results = readTextFiles(files); // may throw IOException

        // Print contents of each file
        for (int i = 0; i < results.length; i++) {
            System.out.println("Contents of " + files[i] + ":");
            System.out.println(results[i]);
            System.out.println("----------");
        }

    } catch (IOException e) {
        // Handle any file-related exceptions here
        System.err.println("Error reading files: " + e.getMessage());
    }
}
```

```
public static String[] readTextFiles(String[] fileNames) throws IOException {
   List<String> contents = new ArrayList<>();

    for (String fileName : fileNames) {
        StringBuilder fileContent = new StringBuilder();

        BufferedReader reader = new BufferedReader(new FileReader(fileName));
        String line;
        while ((line = reader.readLine()) != null) {
            fileContent.append(line).append("\n");
        }
        reader.close();
        contents.add(fileContent.toString());
    }

   return contents.toArray(new String[0]);
}
```

# Some high-level kinds of exceptions

Programming errors, i.e., bugs

- Unchecked exceptions

Problems while accessing external elements, e.g., network, files, GPS

- Checked exceptions

Application-specific problems, e.g., a compiler cannot parse an incorrectly-constructed program

- Checked exceptions, or just regular function results

# Some examples from Java

## (Checked or Unchecked?)

`FileNotFoundException`

# Some examples from Java

## (Checked or Unchecked?)

`FileNotFoundException`

`IndexOutOfBoundsException`

# Some examples from Java

## (Checked or Unchecked?)

`FileNotFoundException`

`IndexOutOfBoundsException`

`ArithmeticException`

# Some examples from Java

## (Checked or Unchecked?)

`FileNotFoundException`

`IndexOutOfBoundsException`

`ArithmeticException`

`SQLException`

# Some examples from Java

## (Checked or Unchecked?)

`FileNotFoundException`

`IndexOutOfBoundsException`

`ArithmeticException`

`SQLException`

`OutOfMemoryError`

# Some examples from Java

## (Checked or Unchecked?)

`FileNotFoundException`

`IndexOutOfBoundsException`

`ArithmeticException`

`SQLException`

`OutOfMemoryError` => Not checked, but not an exception

- According to the JLS*, an application is **not expected to recover**
- Really hairy stuff, e.g., `VirtualMachineError` or `StackOverflowError`

*https://docs.oracle.com/javase/specs/jls/se8/html/jls-11.html

# Which exception types to use?

## A rule of thumb

Try to use **specific exception types**, e.g., if a file is absent, throw `FileNotFoundException`, not `IOException` (its supertype)

Apply the same principle when defining your own exception types, e.g., `InvalidMoveException`, instead of `GameException`

Exception types should represent **types of errors**

- **Don’t overdo it!** You don’t need one exception type for each possible invalid move
- The specific move can be data carried by the exception object

# Handling exceptions

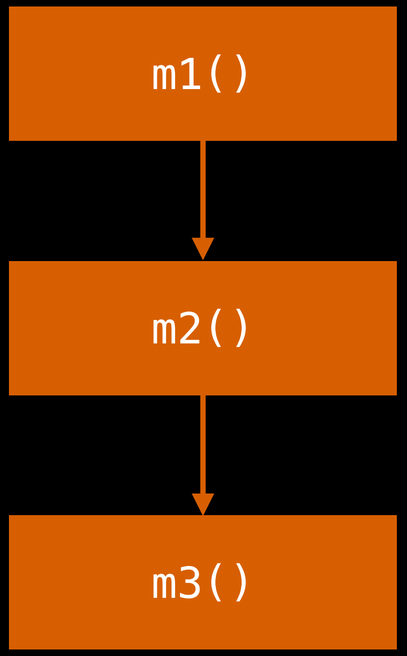
**AKA: recovering from errors or failing graciously**

Handling an exception == doing something with it that stops its propagation

# Handling exceptions

**AKA: recovering from errors or failing graciously**

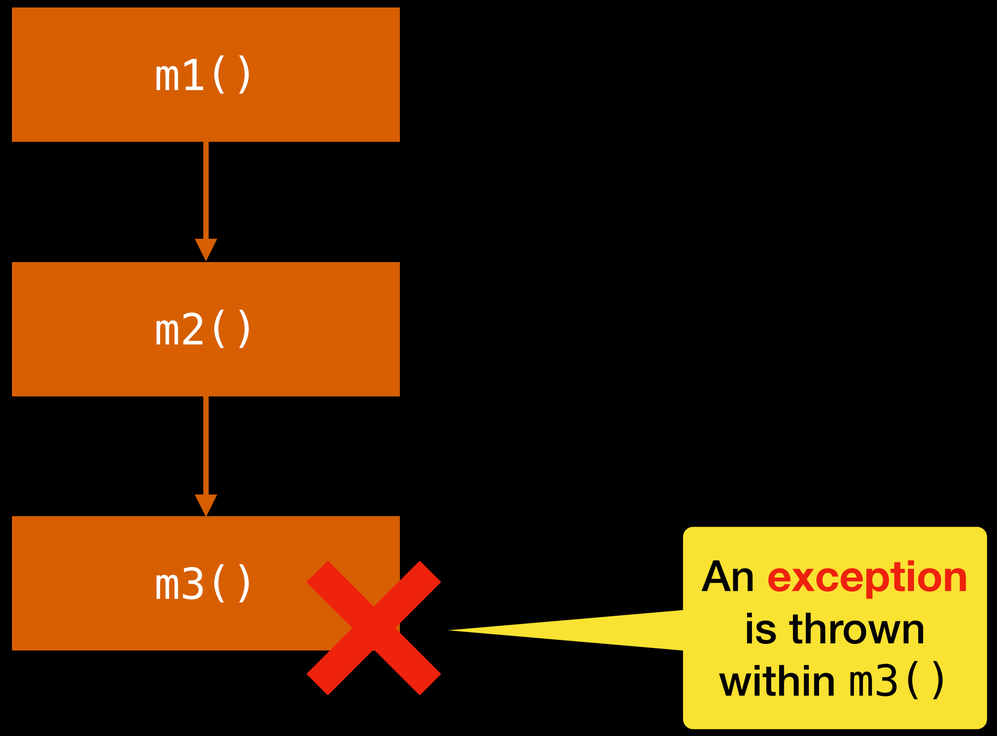
Handling an exception == doing something with it that stops its propagation



# Handling exceptions

**AKA: recovering from errors or failing graciously**

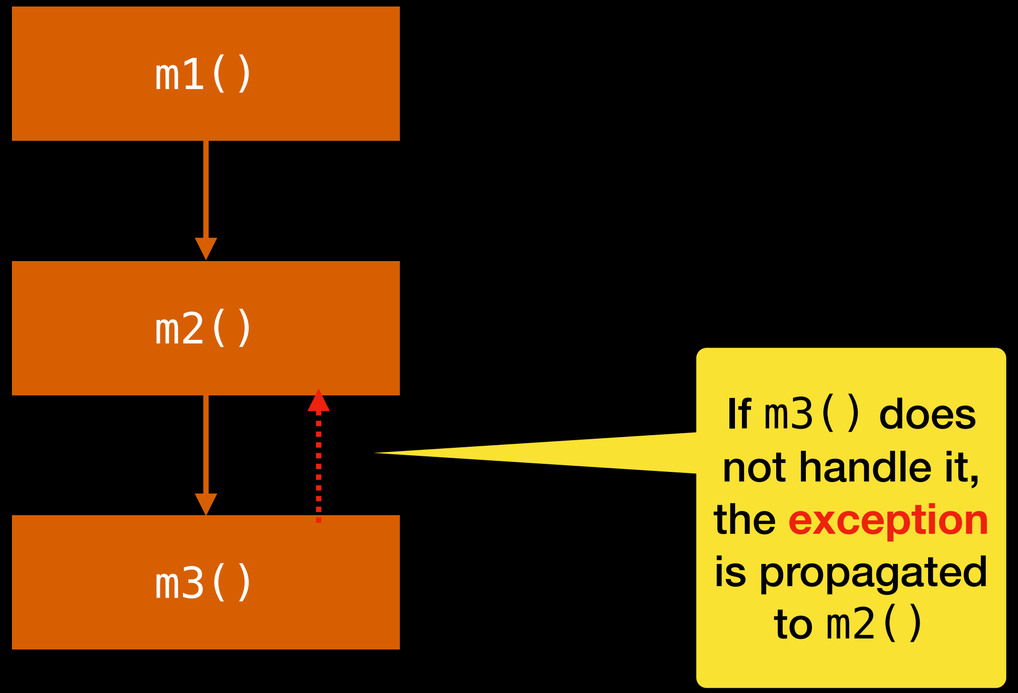
Handling an exception == doing something with it that stops its propagation



# Handling exceptions

**AKA: recovering from errors or failing graciously**

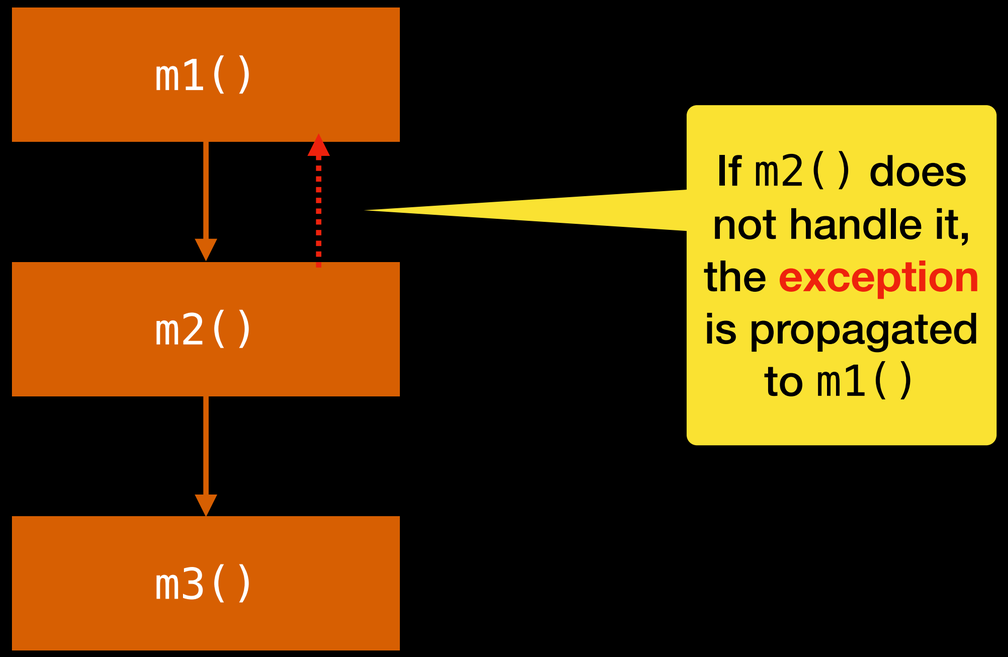
Handling an exception == doing something with it that stops its propagation



# Handling exceptions

**AKA: recovering from errors or failing graciously**

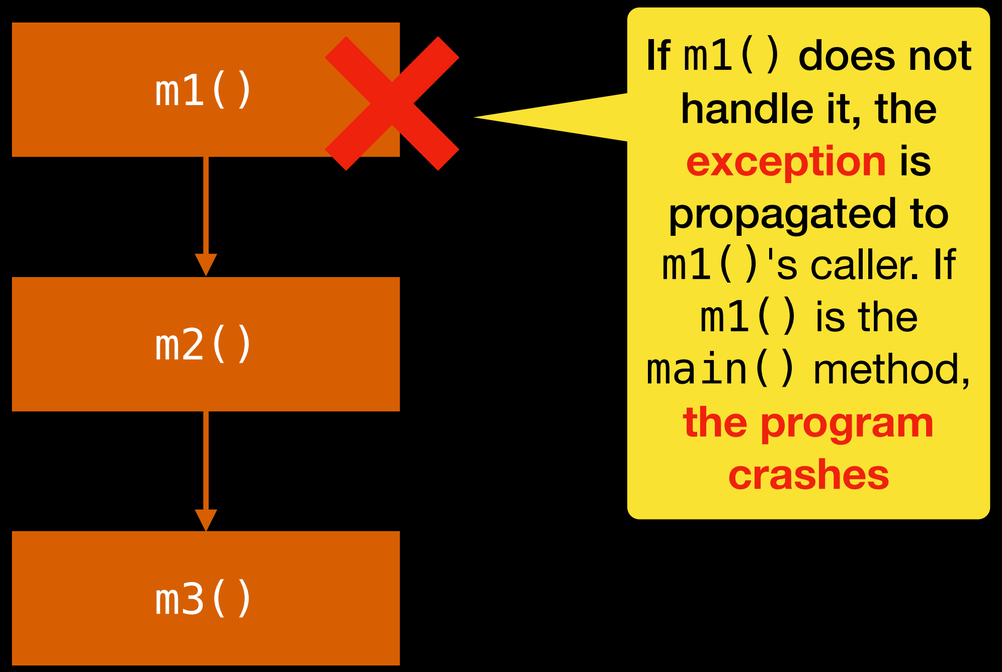
Handling an exception == doing something with it that stops its propagation



# Handling exceptions

## AKA: recovering from errors or failing graciously

Handling an exception == doing something with it that stops its propagation



# Handling exceptions

**AKA: recovering from errors or failing graciously**

Handling an exception == doing something with it that stops its propagation

In Java, `catch` clauses attached to `try` statements implement exception handlers

- Stop exception propagation, if types match
- Their code blocks do the actual handling

But **what should a `catch` block do** with a caught exception?

# What to do with an exception?

## As with most things in life, it depends

Some applications will **continue running** despite most errors

- Servers, compilers, operating systems
- User should be notified of the error, if applicable, and the error should be logged

In some cases, **re-throwing** is the answer

- Libraries and frameworks often do not handle errors
- Just signal them to client code in a clear and well-documented manner

# What to do with an exception?

## A bit more…

Does it **stop execution** from proceeding?

- If yes, the exception should at least be logged
- User should be notified of the error, if applicable

If some **alternative action** can be taken, take it

- Exs.: Invoking another service, running in local mode, using approximate location, instead of the GPS, returning a default value
- Again, logging the error is usually a good idea

# Things not to do, in general

**Empty** `catch` block

- Unless you know it is ok to ignore the exception

**Overly general** `catch` blocks, e.g., `catch(Exception)` or `catch(Throwable)`

- Unless you want the program to never stop, e.g., because it is a server or the outer loop of a game
- In this case, it makes sense to have this low in the stack, near `main()`

Handle the exception with **`e.printStackTrace()`**

- Unless you are only debugging the program
- Even then, it is better to avoid, as you may forget it there

Universally:

Ensure that acquired **resources are released**, even in the presence of errors

Speaking of resources, let’s go back to a previous example

```
public static String[] readTextFiles(String[] fileNames) throws IOException {
   List<String> contents = new ArrayList<>();

    for (String fileName : fileNames) {
        StringBuilder fileContent = new StringBuilder();

        BufferedReader reader = new BufferedReader(new FileReader(fileName));
        String line;
        while ((line = reader.readLine()) != null) {
            fileContent.append(line).append("\n");
        }
        reader.close();
        contents.add(fileContent.toString());
    }

   return contents.toArray(new String[0]);
}
```

```
public static String[] readTextFiles(String[] fileNames) throws IOException {
   List<String> contents = new ArrayList<>();

    for (String fileName : fileNames) {
        StringBuilder fileContent = new StringBuilder();

        BufferedReader reader = new BufferedReader(new FileReader(fileName));
        String line;
        while ((line = reader.readLine()) != null) {
            fileContent.append(line).append("\n");
        }
        reader.close(); // Not called if exception occurs!
        contents.add(fileContent.toString());
    }

   return contents.toArray(new String[0]);
}
```

```
public static String[] readTextFiles(String[] fileNames) throws IOException {
        List<String> contents = new ArrayList<>();
        for (String fileName : fileNames) {
             BufferedReader reader = null;
             try {
               StringBuilder fileContent = new StringBuilder();

               reader = new BufferedReader(new FileReader(fileName));
               String line;
               while ((line = reader.readLine()) != null) {
                   fileContent.append(line).append("\n");
               }
               contents.add(fileContent.toString());
             } finally {
               if (reader!=null)  { reader.close(); } // always invoked
             }
         }
    return contents.toArray(new String[0]);
}
```

```
public static String[] readTextFiles(String[] fileNames) throws IOException {
       List<String> contents = new ArrayList<>();
       for (String fileName : fileNames) {
         BufferedReader reader = null;
         try {
            StringBuilder fileContent = new StringBuilder();

            reader = new BufferedReader(new FileReader(fileName));
            String line;
            while ((line = reader.readLine()) != null) {
                fileContent.append(line).append("\n");
            }
            contents.add(fileContent.toString());
         } finally {
             try {
                 if (reader != null) { reader.close(); }
             } catch (IOException e) { ... /* log the exception and stop */ }
         }

       }
   return contents.toArray(new String[0]);
}
```

Show file `SocketReaderToFile.java`

Show file

`ImprovedSocketReaderToFile.java`