



## Exception Handling

Dr. Abdelkarim Erradi

CSE@QU

# Outline

- Exception handling using **try-catch-finally**
- Exception Types
- Throwing an Exception
- Custom Exceptions


# try-catch-finally block

# Definition

- An exception is an event that occurs during the execution of a program that **disrupts the normal flow of execution**.
- Examples
  - A program is going to read a file, but *the file is missing*
  - A program is reading an array, but the *out of bound* case occurs
  - A program is receiving a file, but the network connection fails
- With **exception handling**, a program can continue executing (rather than terminating) after dealing with the exception.

# Example of throwing an exception

```
1  // Fig. 8.1: Time1.java
2  // Time1 class declaration maintains the time in 24-hour format.
3
4  public class Time1
5  {
6      private int hour; // 0 - 23
7      private int minute; // 0 - 59
8      private int second; // 0 - 59
9
10     // set a new time value using universal time; throw an
11     // exception if the hour, minute or second is invalid
12     public void setTime( int h, int m, int s )
13     {
14         // validate hour, minute and second
15         if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
16             ( s >= 0 && s < 60 ) )
17         {
18             hour = h;
19             minute = m;
20             second = s;
21         } // end if
22     else
23         throw new IllegalArgumentException(
24             "hour, minute and/or second was out of range" );
25     } // end method setTime
```

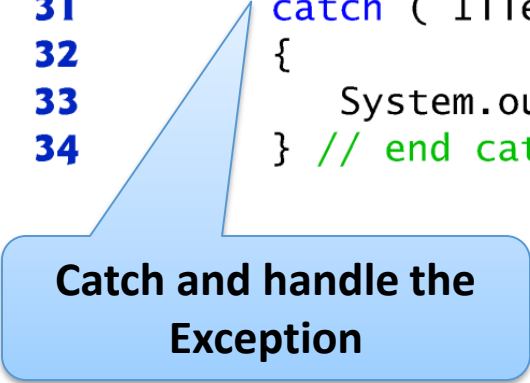


throws an  
IllegalArgumentException  
if the data validation fails in  
set method

# Example of catching and handling the Exception

---

```
25
26      // attempt to set time with invalid values
27      try
28      {
29          time.setTime( 99, 99, 99 ); // all values out of range
30      } // end try
31      catch ( IllegalArgumentException e )
32      {
33          System.out.printf( "Exception: %s\n\n", e.getMessage() );
34      } // end catch
```



**Catch and handle the  
Exception**

# Method `setTime()` and Exception Handling

- For incorrect values, `setTime` throws an **exception** of type `IllegalArgumentException` (lines 23–24)
  - The **throw** statement (line 23) creates a new object of type `IllegalArgumentException` and pass a custom error message
  - The **throw** statement immediately terminates `setTime()` method and the exception is returned to **the client code** that attempted to set the time
  - The client can use **try...catch** to handle the exception

# Some common exceptions ...

- `ArrayIndexOutOfBoundsException` occurs when an attempt is made to access an element past either end of an array
- A `NullPointerException` occurs when a null reference is used where an object is expected
- `ClassCastException` occurs when an attempt is made to cast an object that does not have an *is-a* relationship with the type specified in the cast operator.
- `IOException` may occur when reading or writing to files

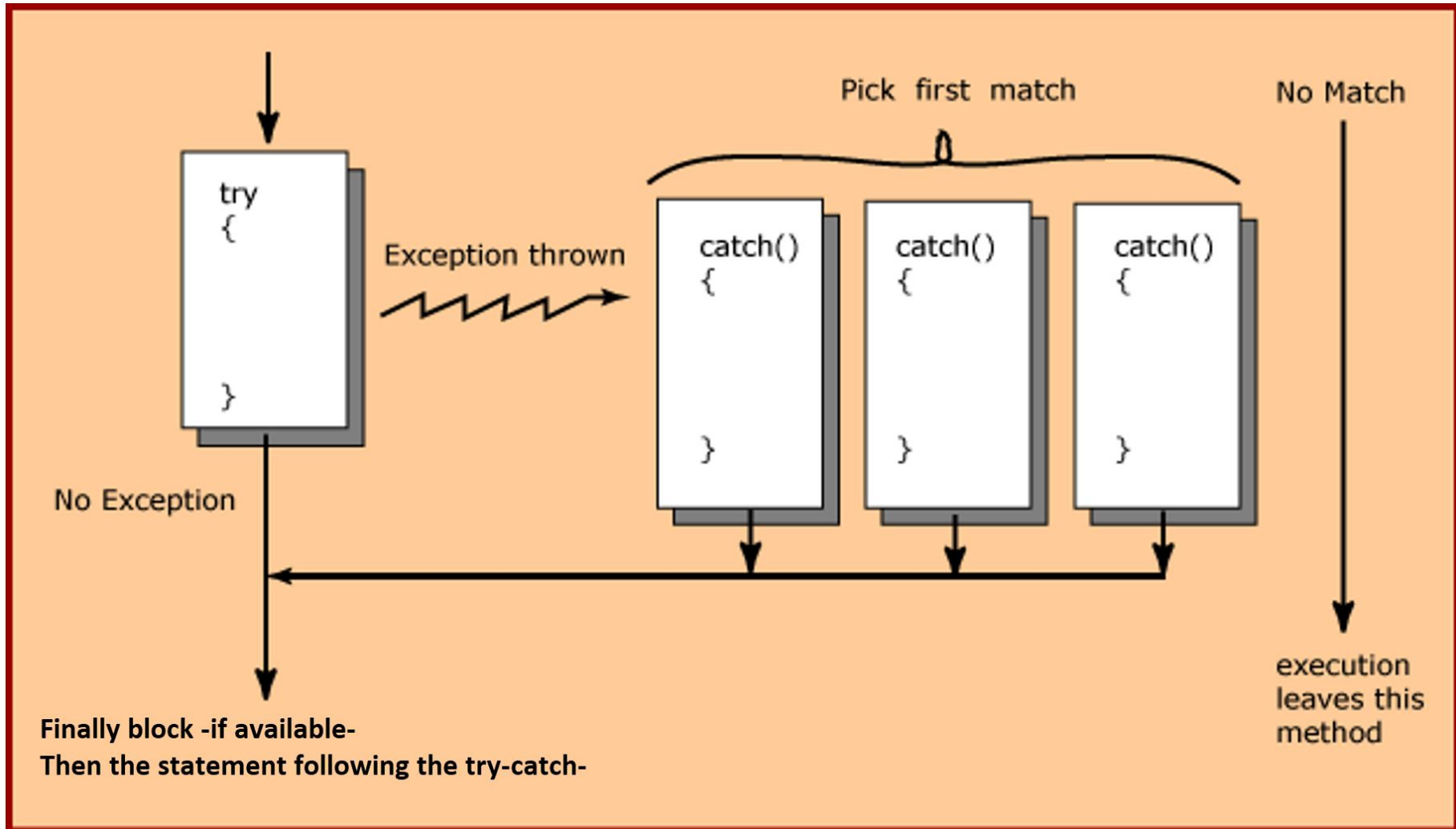


# Exception Handling using **try-catch-finally**

```
try {  
    //...something might have exception  
}  
  
catch (SomeExceptionClass e) {  
    //handle the exception here  
}  
  
finally {  
    //release resources  
    //Perform any actions here common to  
    whether or not an exception is thrown.  
}
```



# How try and catch Work



# How try and catch Work (1 of 2)

- If an exception occurs in a **try** block, the try block **terminates immediately** and program control transfers to the first catch block whose type matches the type of the exception that occurred
  - If there are remaining statements after the statement that causes the exception, those remaining statements won't be executed.
- After the exception is handled, any remaining catch blocks are ignored, and execution resumes at:
  - The **finally block**, if one is present
  - Or at the first line of code after the try...catch sequence
  - **Control does not return to the try block.**

# How try and catch Work (2 of 2)

- If no **catch**{ } block matches the exception, the **execution leaves this method**
  - The unhandled exception is passed to the caller
  - Uncaught exceptions keep propagating all the way to main method. If they are still not handled in the main then the Java runtime will display an error message then terminate the application
- If no exception occurs in the try block, the catch blocks are skipped and the execution continues at the **finally block**, if one is present, then the statement after the try...catch sequence
  - The **finally block** , if one is present, will always execute whether or not an exception occurs in the corresponding try block
  - The finally block is used to **release resources** acquired in the try block such as closing files and database/network connections

# When to Use Exceptions?

- Use **the event is truly exceptional and is an error**
- Do not use it to deal with simple and expected situations
- Example:

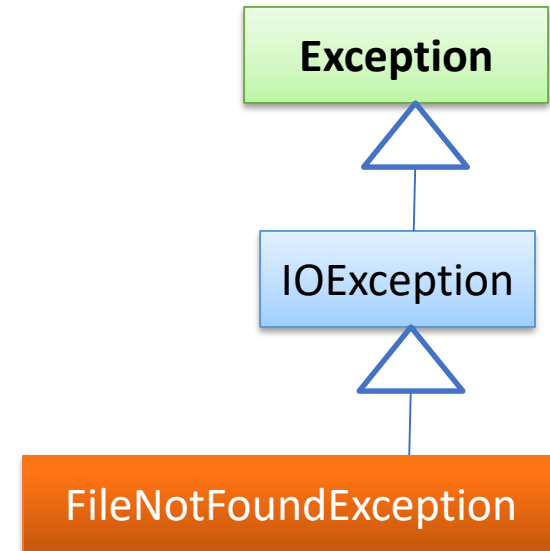
```
try {  
    System.out.println(refVar.toString());  
}  
catch (NullPointerException ex) {  
    System.out.println("refVar is null");  
}
```

Can be replaced by:

```
if (refVar != null)  
    System.out.println(refVar.toString());  
else  
    System.out.println("refVar is null");
```

# Try with multiple catch blocks

```
try {  
    //maybe read a file or something...  
}  
catch (FileNotFoundException e) {  
    System.out.println("FileNotFoundException: " +  
        e.getMessage());  
}  
catch (IOException e) {  
    System.out.println("Caught IOException: " +  
        e.getMessage());  
}
```



- *Why we catch `FileNotFoundException` first, and then `IOException`?*

=> The **most specific exception types should appear first** in the structure, followed by the more general exception types.

- Otherwise if we place the general exception types first then the catch blocks of specific exception types will never be used
- e.g., if we move **catch (IOException e)** first then **catch (FileNotFoundException e)** will never be used.

# Superclass/Subclass Exceptions

- A catch parameter of an exception **can also catch all the exception subtypes**
  - Enables catch to handle related exceptions with a concise notation
  - E.g., **catch (Exception e)** can catch all exceptions that are subtypes of Exception class
- Catching related exceptions in one catch block makes sense only if the handling behavior is the same for **all** subclasses
- You can also catch each subclass type individually if those exceptions require different processing
- If multiple catch blocks match a particular exception type, only the **first** matching catch block executes.

# Multi-Catch

```
try {  
    ...  
} catch (ClassCastException e) {  
    doSomethingClever(e);  
    throw e;  
} catch (InstantiationException |  
         NoSuchMethodException |  
         InvocationTargetException e) {  
    // Useful if you do generic actions  
    log(e);  
    throw e;  
}
```



Log the exception then  
rethrow it



# try-with-resources

- The **try-with-resources** is a try statement that declares a *resource* (i.e., an object that must be closed after the try block)
  - Any object that implements `java.lang.AutoCloseable` can be used as a resource.
  - “**try-with-resources**” will auto-close the resource (e.g., an open file) that was created in the try

```
try (Scanner inputFile = new Scanner(new FileInputStream(filePath))) {  
    String line;  
    while (inputFile.hasNext()) {  
        line = inputFile.nextLine();  
        fileLines.add(line);  
    }  
} catch (IOException e) {  
    System.out.println(e.getMessage());  
}
```

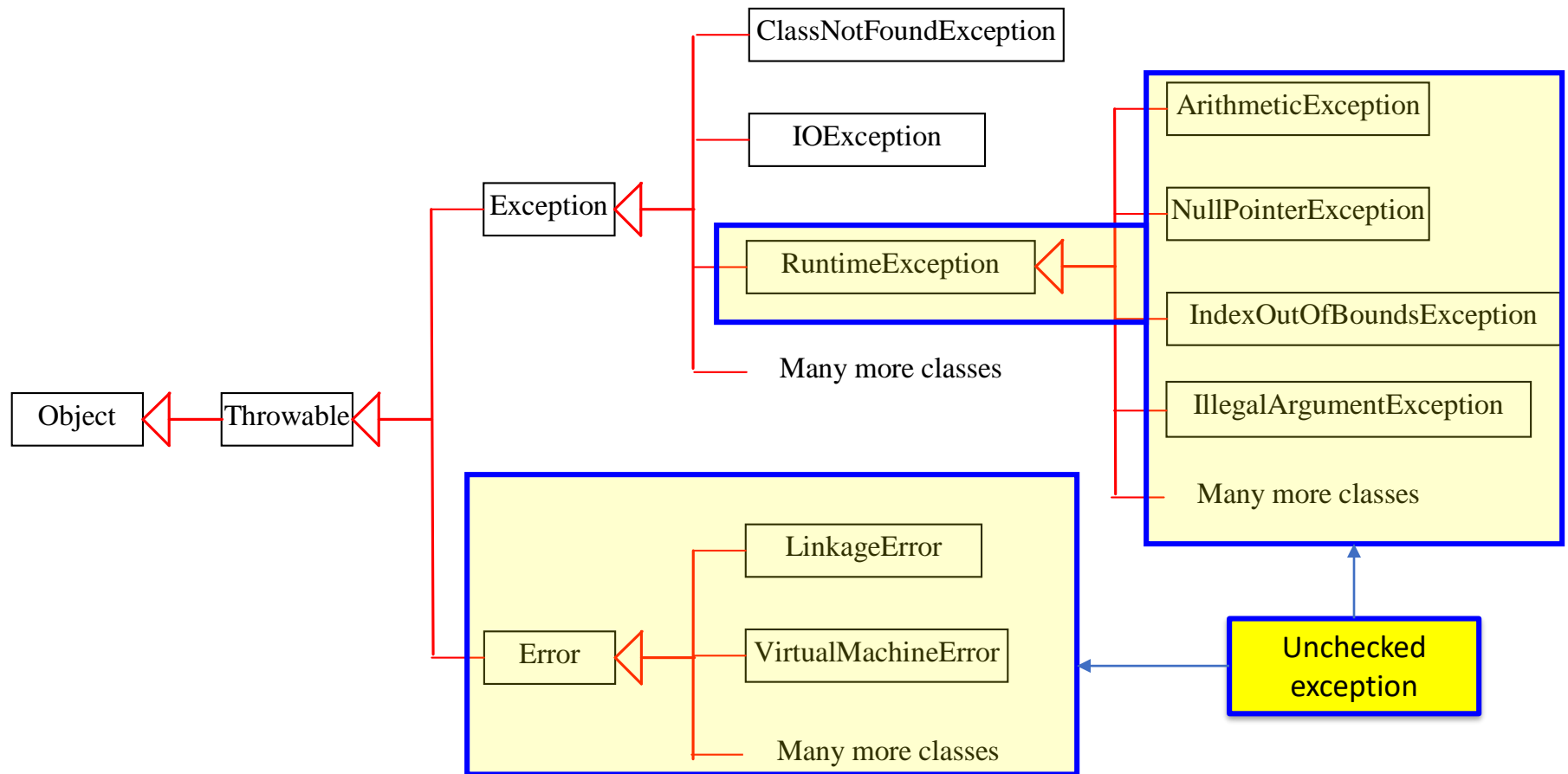
inputFile will be closed automatically after try block. No need to add a finally block

# Exception Types

# 2 kinds of exceptions

- ***Checked exception*** (*Java forces you to handle them*)
  - These are exceptional conditions that a **well-written application should anticipate and recover from**.
  - They occur usually when interacting with resources e.g. missing files, database connection problems, network connection errors e.g., `FileNotFoundException`
- ***Unchecked exceptions*** (*Java does NOT force you to handle them*), they include:
  - **Error exceptions** (that extend `Error` class): Exceptional conditions that are **external to the application and outside its control**. The application usually cannot anticipate or recover from them. e.g., **Out of memory exception**
  - **Runtime exceptions** (that extend `RuntimeException` class) : caused by programming errors, such as accessing a null object, bad casting, accessing an out-of-bounds array, and arithmetic errors.

# Unchecked vs. Checked Exceptions

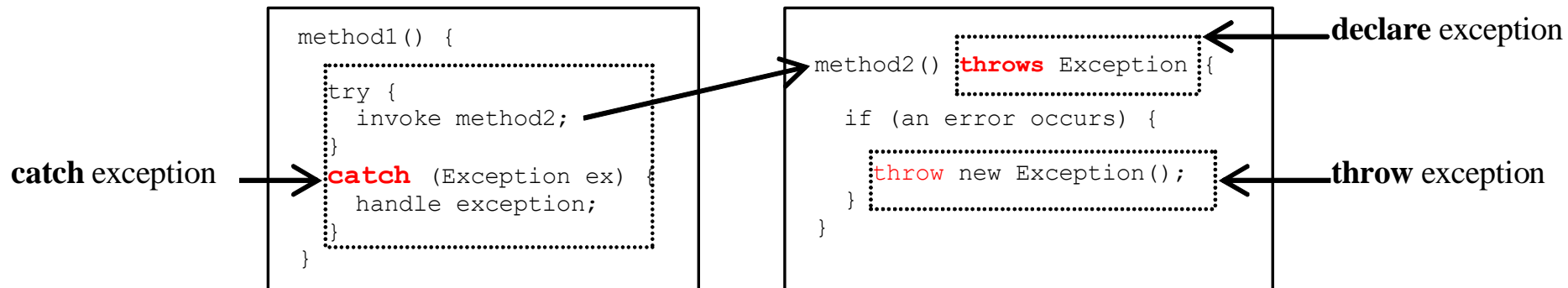


- Java does not mandate you to **catch and handle unchecked exceptions**
- All other exceptions are **checked exceptions**, meaning that the compiler forces the programmer to catch and handle them.

# Throwing an Exception

# Declaring, Throwing, and Catching Exceptions

- Throw exceptions to indicate a problem
- Use catch blocks to specify exception handlers



# Declaring Exceptions

- Specifies the exceptions a method may throw

E.g., `public void myMethod() throws IOException`

- Can declare a comma-separated list of the exceptions that the method will throw if various problems occur
  - May be thrown by statements in the method's body or by methods called from the body
- Method can throw exceptions listed in its throws clause or their subclasses.
  - e.g., `IOException` is a subclass of `Exception`
- Clients of a method with a throws clause are thus informed that the method may throw exceptions

# Throwing Exceptions

- A method can create an exception instance and throw it

```
public void setRadius(double radius)
    throws IllegalArgumentException {
    if (radius >= 0)
        radius = radius;
    else
        throw new IllegalArgumentException(
            "Radius cannot be negative");
}
```

- A method can throw multiple exceptions

```
public static void appendToFile(String filePath, String textToAppend)
    throws IOException, AlreadyExistsException {
```



# When a called method throws an exception the caller should either throw it or handle it

```
public static void appendToFile(String filePath, String textToAppend)
    throws AlreadyExistsException {
    if (isLineExists(filePath, textToAppend)) {
        throw new AlreadyExistsException("The line to be add already exists");
    }
    writeToFile(filePath, textToAppend, true);
}
```

Add throws declaration  
Surround with try/catch

```
...
public static void appendToFile(String filePath, String
textToAppend)
throws AlreadyExistsException, IOException {
    if (isLineExists(filePath, textToAppend)) {
        ...
    }
}
```

When a called method  
throws an exception.  
The caller should  
either throw it or  
handle it.

```
...
}
}
try {
    writeToFile(filePath, textToAppend, true);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
}
...
```

# Custom Exceptions

# Creating a New Exception Type

- Sometimes it's useful to create your own exception classes that are specific to the problems that can occur in your app
  - A new exception class **must extend** an existing exception class (such as `Exception`) to be able to use the exception-handling mechanism
  - Before creating a custom exception try to first use one of Java's built-in exception classes that might be suitable for the type of problems your methods need to indicate

```

public class InvalidLoginException extends Exception {
    //An example of throwing it could be:
    //throw new InvalidLoginException("Email and/or password are invalid");
    public InvalidLoginException(String message) {
        super(message);
    }

    public InvalidLoginException(String message, Throwable cause) {
        super(message, cause);
    }
}

```

---

```

public class AlreadyExistsException extends Exception {
    private String dataToAdd; //with getters and setters
    private String destination;
    public AlreadyExistsException(String message) {
        super(message);
    }
}

```

A custom Exception is a class that extends Exception. It can have extra attributes and methods.

```

public AlreadyExistsException(String message, String dataToAdd, String destination) {
    super(message);
    this.dataToAdd = dataToAdd;
    this.destination = destination;
}

public AlreadyExistsException(String message, Throwable cause) {
    super(message, cause);
}

```

# Summary

- Exceptions are a powerful mechanism for separating Error-Handling Code from "Regular" Code => **this simplifies the normal flow code**
- The ***try block*** identifies a block of code in which an exception can occur
- The ***catch block*** defines as an exception handler that can handle a particular type of exception
- The ***finally block*** is guaranteed to execute, and is the right place to release resources acquired in the try block such as closing files, database connections and network connections