Introduction to Programming

Week 13

Magnus Madsen

Week 13: Outline

- Exceptions: throwing and catching
- Checked vs. unchecked exceptions
- The exception hierarchy
- User-defined exceptions
- Exception chaining
- Try-Catch-Finally and Try-With-Resources
- File I/O basics

Prologue

Quote of the Week

"As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

— Maurice Wilkes

Epigram of the Week

"If you lie to the computer, it will get you."

— Perry Farrar

When Things Go Wrong

Ariane 5

On June 4, 1996, the Ariane 5 rocket exploded mid-launch due to a software bug.

How Can Programs Go Wrong?

The program is used incorrectly:

- User wants to open a file, but the file does not exist.
- User wants to connect to a domain, but the domain does not exist.
- User wants to perform an action, but does not have permission.

How Can Programs Go Wrong?

The program is used incorrectly:

- User wants to open a file, but the file does not exist.
- User wants to connect to a domain, but the domain does not exist.
- User wants to perform an action, but does not have permission.

The program depends on unreliable resources:

- The network is down.
- The file system is full.

How Can Programs Go Wrong?

The program is used incorrectly:

- User wants to open a file, but the file does not exist.
- User wants to connect to a domain, but the domain does not exist.
- User wants to perform an action, but does not have permission.

The program depends on unreliable resources:

- The network is down.
- The file system is full.

The program violates a protocol or performs a nonsensical operation:

- The program tries to index outside the bounds of an array.
- The program tries to divide by zero.
- The program tries to invoke a method on a `null` object.

How Can We Handle Errors?

The program is used incorrectly:

- Explain to the user what is wrong and ask for a corrective action!
- **Recoverable**, we can ask the user for help.

How Can We Handle Errors?

The program is used incorrectly:

- Explain to the user what is wrong and ask for a corrective action!
- **Recoverable**, we can ask the user for help.

The program depends on unreliable resources:

- If the network is down, we can try again later.
- If the file system is full, we can ask for corrective action.
- Often **recoverable**. Perhaps the problem will fix itself or the user can help.

How Can We Handle Errors?

The program is used incorrectly:

- Explain to the user what is wrong and ask for a corrective action!
- **Recoverable**, we can ask the user for help.

The program depends on unreliable resources:

- If the network is down, we can try again later.
- If the file system is full, we can ask for corrective action.
- Often **recoverable**. Perhaps the problem will fix itself or the user can help.

The program violates a protocol or performs a nonsensical operation:

- Programming bugs: we, the developers, have made a mistake.
- Almost always **unrecoverable**, the program itself must be corrected.
- Not much to do, other than to try to save data, and exit the program.

How can we signal errors within a program?

Store an error code in a global variable

- Commonly used in imperative languages such as C, Pascal, and Fortran.

How can we signal errors within a program?

Store an error code in a global variable

- Commonly used in imperative languages such as C, Pascal, and Fortran.

Return an error code, if any. Otherwise return a success value

- Commonly used in imperative languages such as C, Pascal, and Fortran.
- We have to distinguish between error codes and successful return values.

How can we signal errors within a program?

Store an error code in a global variable

- Commonly used in imperative languages such as C, Pascal, and Fortran.

Return an error code, if any. Otherwise return a success value

- Commonly used in imperative languages such as C, Pascal, and Fortran.
- We have to distinguish between error codes and successful return values.

Return a data structure which encodes success or failure

- Commonly used in functional languages such as Scala, Haskell, and OCaml.

How can we signal errors within a program?

Store an error code in a global variable

- Commonly used in imperative languages such as C, Pascal, and Fortran.

Return an error code, if any. Otherwise return a success value

- Commonly used in imperative languages such as C, Pascal, and Fortran.
- We have to distinguish between error codes and successful return values.

Return a data structure which encodes success or failure

- Commonly used in functional languages such as Scala, Haskell, and OCaml.

Redirect control-flow with an exception

- Commonly used in object-oriented languages such as Java, C#, Python, and C++.

Exceptions

What Is an Exception?

An **exception** is a **non-local control-flow** mechanism that interrupts normal execution and jumps to a handler to deal with **abnormal situations**.

Key Concepts:

- We use exceptions when we encounter **unexpected situations** or **error conditions**.
- We **throw** and **catch** exceptions. A Java exception is an object (an instance of a class).
- We **throw** an exception when we want to abort execution in a method.
- We **catch** an exception when we want to handle it and continue execution.
- If an exception is never caught, it reaches `main`, and the program aborts.

The Call Stack

Recall: We have seen that execution of a program relies on the **call stack**:

```
public class Main {
    public static void main(String[] args) { f(); }
    static void f() { g(); }
    static void g() { h(); }
    static void h() { throw new RuntimeException(); }
}
```

When an exception is thrown, and not caught, it prints the **stack trace**:

```
Exception in thread "main" java.lang.RuntimeException
  at Main.h(Main.java:5)
  at Main.g(Main.java:4)
  at Main.f(Main.java:3)
  at Main.main(Main.java:2)
```

Example: Throw

We can use exceptions to **abort execution** when given illegal input:

```
public class BankAccount {
    private int balance;

    public void withdraw(int amount) {
        if (amount <= 0) {
            String msg = "Amount must be positive.";
            throw new IllegalArgumentException(msg);
        }

        if (amount > balance) {
            String msg = "Amount must be less than or equal to the balance.";
            throw new IllegalArgumentException(msg);
        }

        balance = balance - amount;
    }
}
```

Example: Try-Catch

We can catch the exceptions elsewhere and ask for new input:

```
public static void main(String[] args) {
    BankAccount account = new BankAccount(100);
    while (true) {
        System.out.println("Enter an amount to withdraw:");
        int amount = StdIn.readInt();
        try {
            account.withdraw(amount);
        } catch (IllegalArgumentException e) {
            System.out.println("Oopsie... Not enough funds!");
        }
    }
}
```

Remark: No stack trace is printed because the exception is caught and handled.

Finding an Exception Handler

When an exception is thrown, the runtime searches for a suitable **exception handler** by **unwinding the call stack**.

The search proceeds as follows:

- The runtime examines the current stack frame for a matching handler.
- If a handler is found, execution **resumes at that point**.
- If no handler is found, the stack frame is **popped** and the search continues down the stack.
- If the search reaches `main`, the program **terminates** and prints a stack trace.

Q:

What does the following program fragment print?

```
public static void main(String[] args) {
    try {
        System.out.print("Hello ");
        throw new IllegalArgumentException("Oops!");
        System.out.print("World");
    } catch (IllegalArgumentException e) {
        System.out.print("Moon");
    }
    System.out.println("Goodbye");
}
```

Magnus' Phone Analogy

Exceptions are for **non-local** control.

Checked and Unchecked Exceptions

In Java, exceptions come in two variants[1]:

- A **checked** exception **must be caught or re-thrown.**
  - A subclass of `Exception` is a checked exception.
  - Enforced by the Java compiler.
- An **unchecked** exception may be caught or re-thrown.
  - A subclass of `RuntimeException` is an unchecked exception.

We have seen `IllegalArgumentException` as an example of an **unchecked** exception.

[1] Here we ignore Java's `Error` class.

Rationale for Checked and Unchecked Exceptions

Checked exceptions should be used for **recoverable error conditions**:

- Examples: file not found, network failure, database unavailable.
- The compiler enforces explicit handling of these exceptions.

Unchecked exceptions should be used for **programming defects**:

- Examples: null pointer dereference, array index out of bounds, division by zero.
- These indicate bugs that should be corrected in the source code.

Example: Checked Exceptions (1/2)

We can write a Java method to check if a file contains a specific `needle` string:

```
static boolean fileContains(String needle, String file) {
    String text = Files.readString(Path.of(file));
    return text.contains(needle);
}
```

Here we use `Files.readString` which may throw an **`IOException`**.

Example: Checked Exceptions (1/2)

We can write a Java method to check if a file contains a specific `needle` string:

```
static boolean fileContains(String needle, String file) {
    String text = Files.readString(Path.of(file));
    return text.contains(needle);
}
```

Here we use `Files.readString` which may throw an **`IOException`**.

Since **`IOException`** is a **checked** exception (a subclass of `Exception`), Java complains:

```
unreported exception IOException; must be caught or declared to be thrown
```

Example: Checked Exceptions (2/2)

(A) We can catch the exception:

```
static boolean fileContains(String needle, String file) {
    try {
        String text = Files.readString(Path.of(file));
        return text.contains(needle);
    } catch (IOException e) {
        return false;
    }
}
```

or **(B) We can declare that `fileContains` throws it:**

```
static boolean fileContains(String needle, String file) throws IOException {
    String text = Files.readString(Path.of(file));
    return text.contains(needle);
}
```

Q:

Which should be checked or unchecked?

- `NegativePriceException`
- `InvalidPinException`
- `OutOfStockException`
- `NegativeAltitudeException`
- `OutOfMemoryException`
- `RunwayOccupiedException`
- `InvalidISBNException`
- `NegativeAgeException`
- `InsufficientFundsException`

Rethrowing Exceptions

We can **catch** an exception, perform some action, and then **rethrow** it:

```
public void processPayment(double amount) throws PaymentException {
    try {
        creditCard.charge(amount);
    } catch (PaymentException e) {
        logger.error("Payment failed: " + e.getMessage());
        throw e; // Rethrow the same exception
    }
}
```

We can also **catch one exception** and **throw a different one**:

```
try {
    Files.readString(Path.of(file));
} catch (IOException e) {
    throw new IllegalArgumentException("Invalid file: " + file);
}
```

Q:

What can you say about this program fragment?

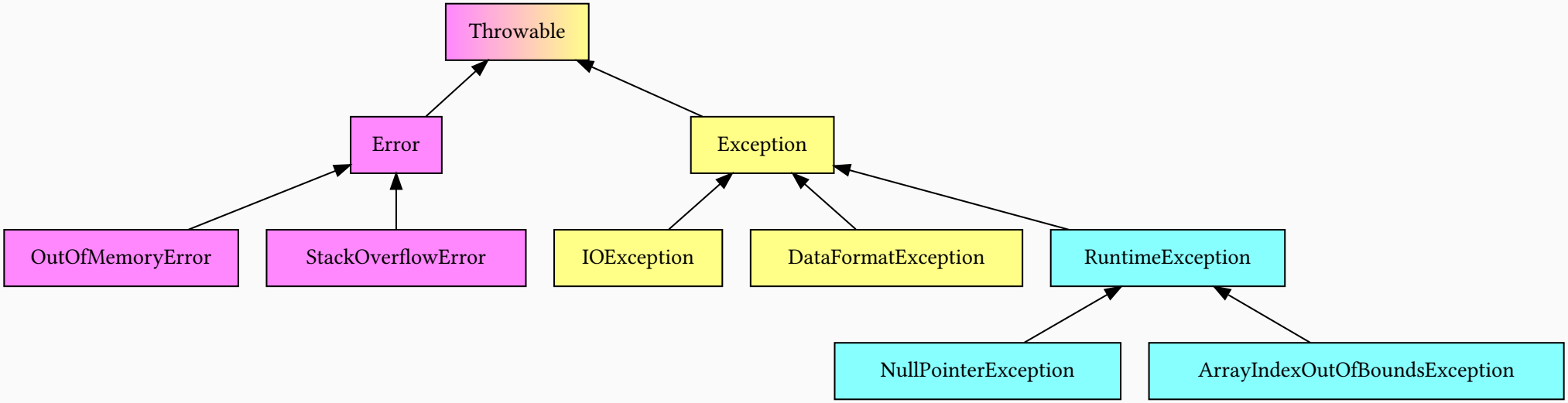
```
public static void main(String[] args) {
    if (f()) { g() }
}

static boolean f() {
    try {
        g();
        return true;
    } catch (IOException e) {
        throw IllegalArgumentException("...");
    }
}

static void g() throws IOException {
    Files.readString(Path.of("data.txt"));
}
```

The Exception Hierarchy

The Exception Hierarchy



Common Unchecked Exceptions

`ArithmeticException`

Thrown for exceptional arithmetic conditions, such as integer division by zero or modulo by zero.

`ArrayIndexOutOfBoundsException`

Thrown when accessing or modifying an array element with an invalid index (negative or ≥ length).

`NullPointerException`

Thrown when using a `null` reference, such as calling methods, accessing fields, or indexing arrays on null.

`IllegalStateException`

Thrown when a method is invoked at an illegal or inappropriate time.

```
Scanner scanner = new Scanner(System.in);
scanner.close();
scanner.nextInt(); // Scanner is closed
```

`NumberFormatException`

Thrown when converting a string to a numeric type with invalid format.

```
String text = "not a number";
int value = Integer.parseInt(text);
```

All of these are **unchecked** exceptions.

Overview: Checked and Unchecked Exceptions

Type	Extends	Examples
Checked	`Exception`	`IOException`, `SQLException`, `FileNotFoundException`, `ClassNotFoundException`
Unchecked	`RuntimeException`	`NullPointerException`, `IllegalArgumentException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`

- Checked exceptions must be caught or declared to be thrown.
- Unchecked exceptions may be left unhandled (but may crash the program).

Try-Catch Uses Subtyping

Exception handlers leverage **polymorphism**: a catch clause matches the declared type **and all its subtypes**.

For example:

```
try {
    int result = 10 / 0;
    System.out.println(result);
} catch (RuntimeException e) {
    System.out.println("Caught!");
}
```

User-Defined Exceptions

User-Defined Exceptions (1/2)

We can **define our own exceptions** by extending the `RuntimeException` class or the `Exception` class.

For example:

```
public class NegativeAmountException extends RuntimeException {
    public NegativeAmountException() {
        super("Negative amount entered.");
    }
}
```

User-Defined Exceptions (2/2)

Exceptions are ordinary classes that can have fields and methods:

```
public class InsufficientFundsException extends RuntimeException {
    private int deficit;

    public InsufficientFundsException(int deficit) {
        super("Insufficient funds. Deficit: " + deficit);
        this.deficit = deficit;
    }

    public int getDeficit() {
        return deficit;
    }
}
```

Multiple Catch Clauses

We can catch more than one exception:

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    BankAccount account = new BankAccount(100);
    while (true) {
        System.out.println("Enter an amount to withdraw:");
        try {
            int amount = scanner.nextInt();
            account.withdraw(amount);
        } catch (NegativeAmountException e) {
            System.out.println("Nice try! No negative amounts!");
        } catch (InsufficientFundsException e) {
            System.out.println("Oops! You need $" + e.getDeficit() + " more!");
        } catch (InputMismatchException e) {
            System.out.println("That's not a number, silly!");
            scanner.nextLine(); // Clear invalid input
        }
    }
}
```

Exception Chaining

Exception Chaining (1/3)

Sometimes an exception is the result of a series of unfortunate events.

We can use **exception chaining** to connect events to their root cause.

An exception constructor typically has a `cause` parameter, which we can use to specify the exception that caused it.

For example:

```
public class RiderException extends Exception {
    public RiderException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Exception Chaining (2/3)

```
public static void main(String[] args) throws Exception {
    try {
        new Rider(...).charge();
    } catch (RiderException e) {
        throw new Exception("rider lost", e);
    }
}

public  void charge() throws RiderException {
    try {
        horse.gallop();
    } catch (HorseException e) {
        throw new RiderException("horse lost", e);
    }
}
...
```

We can use exception chains to trace the root cause of an issue.

For want of a naile
the shoe is lost,
for want of a shoe
the horse is lost,
for want of a horse
the rider is lost.

Exception Chaining (3/3)

When an exception occurs, we can see what other exception caused it.

```
Exception in thread "main" java.lang.Exception: rider lost
  at WantOfANail.main(WantOfANail.java:6)
Caused by: RiderException: horse lost
  at Rider.charge(WantOfANail.java:22)
  at WantOfANail.main(WantOfANail.java:4)
Caused by: HorseException: shoe lost
  at Horse.gallop(WantOfANail.java:38)
  at Rider.charge(WantOfANail.java:20)
  ... 1 more
Caused by: ShoeException: want of nail
  at Shoe.checkNails(WantOfANail.java:56)
  at Shoe.secure(WantOfANail.java:51)
  at Horse.gallop(WantOfANail.java:36)
  ... 2 more
```

Try-Catch-Finally

Motivation: Finally

You turn on the stove to make some food,
But something on TV changes your mood,
You watch, you forget,
And now with regret,
Wish you'd turned it off—you should!

Motivation: Finally

You turn on the stove to make some food,
But something on TV changes your mood,
You watch, you forget,
And now with regret,
Wish you'd turned it off—you should!

Motivation: Finally

You turn on the stove to make some food,
But something on TV changes your mood,
You watch, you forget,
And now with regret,
Wish you'd turned it off—you should!

! Exceptions abort computation, but sometimes **we need to finish what we started**.

What is Finally?

A `finally` block contains code that **always executes**, regardless of whether an exception was thrown or caught.

Key Concepts:

- The `finally` block runs after the `try` and any `catch` blocks.
- It executes even if an exception is thrown and not caught.
- It executes even if the `try` or `catch` block contains a `return` statement.
- It is used to clean up resources (close files, release locks, etc.).

Example: Turning Off the Stove (1/2)

When we use exceptions, we risk leaving the stove on:

```
public static void main(String[] args) {
    Stove stove = new Stove();
    try {
        stove.turnOn();
        makeDinner();
        stove.turnOff();
    } catch (BoredException e) {
        System.out.println("Got bored: " + e.getMessage());
    }
}

public static void makeDinner() throws BoredException {
    System.out.println("Preparing ingredients...");
    throw new BoredException("This is taking forever!");
}
```

Example: Turning Off the Stove (2/2)

We can ensure we turn off the stove by adding a `finally` block.

```
public static void main(String[] args) {
    Stove stove = new Stove();
    try {
        stove.turnOn();
        makeDinner();
    } catch (BoredException e) {
        System.out.println("Got bored: " + e.getMessage());
    } finally {
        stove.turnOff();
    }
}
```

What does this program fragment print?

Q:

```
try {
    try {
        System.out.println("A");
        throw new RuntimeException();
    } catch (RuntimeException e) {
        System.out.println("B");
        throw e;
    } finally {
        System.out.println("C");
    }
    System.out.println("D");
} finally {
    System.out.println("E");
}
System.out.println("F");
```

Try-with-Resources

Try-with-Resources provides a cleaner and safer way to manage resources that need cleanup.

Why use try-with-resources instead of `finally`?

- Less verbose and requires less boilerplate code.
- Automatically closes resources, reducing the chance of errors.
- Handles multiple resources elegantly.

Example: Try-With-Resources

Try-with-resources can simplify turning off the stove.

Using Finally:

```
Stove stove = new Stove();
try {
    stove.turnOn();
    makeDinner();
} catch (BoredException e) {
    ...
} finally {
    stove.turnOff();
}
```

Using try-with-resources:

```
try (Stove stove = new Stove()) {
    stove.turnOn();
    makeDinner();
} catch (BoredException e) {
    ...
}
```

Example: Try-With-Resources

```
public static void copyFile(String src, String dst) throws IOException {
    try (FileInputStream input = new FileInputStream(src);
         FileOutputStream output = new FileOutputStream(dst)) {

        byte[] buffer = new byte[1024];
        int bytesRead;
        while ((bytesRead = input.read(buffer)) != -1) {
            output.write(buffer, 0, bytesRead);
        }
    }
    // Both input and output are automatically closed
}
```

ℹ Resources are closed in the reverse order in which they were opened.

Q: What is the difference between `throw` and `throws`?

File I/O

File I/O allows programs to work with **persistent data** that survives beyond program execution.

Common use cases:

- Configuration files and user preferences
- Data storage (CSV, JSON, XML)
- Logs and reports
- Saving application state

Programs typically work with two types of files:

Text Files:

- Human-readable characters
- Line-based structure
- Examples: `.txt`, `.java`, `.csv`, `.json`
- Can be opened in a text editor

Binary Files:

- Raw bytes of data
- Machine-readable format
- Examples: `.jpg`, `.class`, `.pdf`, `.zip`
- Appear as gibberish in a text editor

Note: In this course, we will focus exclusively on working with text files.

Example: Reading an Entire File as a String

We can use `Files.readString()` to read an entire file into a single String:

```
try {
    Path path = Path.of("data.txt");
    String content = Files.readString(path);
    System.out.println(content);
} catch (IOException e) {
    System.out.println("Could not read file: " + e.getMessage());
}
```

Example: Reading an Entire File as a List

We can use `Files.readAllLines()` to read a file into a List of lines:

```
try {
    Path path = Path.of("data.txt");
    List<String> lines = Files.readAllLines(path);
    for (String line : lines) {
        System.out.println(line);
    }
} catch (IOException e) {
    System.out.println("Could not read file: " + e.getMessage());
}
```

Example: Writing a String to a File

We can use `Files.writeString()` to write a String to a file:

```
try {
    Path path = Path.of("output.txt");
    String content = "Hello, World!\nThis is a test file.";
    Files.writeString(path, content);
} catch (IOException e) {
    System.out.println("Could not write file: " + e.getMessage());
}
```

Example: Writing a List to a File

We can use `Files.write()` to write a List of lines to a file:

```
try {
    Path path = Path.of("animals.txt");
    List<String> animals = new ArrayList<>();
    animals.add("Giraffe");
    animals.add("Platypus");
    Files.write(path, animals);
} catch (IOException e) {
    System.out.println("Could not write file: " + e.getMessage());
}
```

We have seen four simple methods for File I/O:

Reading Files:

- `Files.readString(path)`
  - Returns entire file as a String
- `Files.readAllLines(path)`
  - Returns file as List of lines

Writing Files:

- `Files.writeString(path, string)`
  - Writes a String to a file
- `Files.write(path, list)`
  - Writes a List of lines to a file

Remark: All these methods require handling `IOException`.

Limitation: These operations require loading the entire file into memory, which can be problematic for very large files.

Example: Processing a File Line-by-Line

We can use `BufferedReader` to process large files without loading everything into memory:

```
try {
    Path path = Path.of("server.log");
    BufferedReader reader = Files.newBufferedReader(path);
    String line;
    while ((line = reader.readLine()) != null) {
        if (line.contains("ERROR")) {
            System.out.println(line);
        }
    }
    reader.close();
} catch (IOException e) {
    System.out.println("Could not read file: " + e.getMessage());
}
```

Example: Using Try-With-Resources

We can use try-with-resources to automatically close the reader:

```
Path path = Path.of("server.log");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    String line;
    while ((line = reader.readLine()) != null) {
        if (line.contains("ERROR")) {
            System.out.println(line);
        }
    }
    // Reader is automatically closed
} catch (IOException e) {
    System.out.println("Could not read file: " + e.getMessage());
}
```

The **Files** class provides static methods for working with files, including reading, writing, and querying file properties.

Method	Description
`Files.exists(path)`	Check if file exists
`Files.readString(path)`	Read entire file as String
`Files.readAllLines(path)`	Read file as List of lines
`Files.writeString(path, string)`	Write String to file
`Files.write(path, list)`	Write List of lines to file
`Files.newBufferedReader(path)`	Create BufferedReader for line-by-line reading
`Files.newBufferedWriter(path)`	Create BufferedWriter for line-by-line writing

Epilogue

Error of the Week

What can you say about this program?

```
try {
    Files.readString(Path.of("data.txt"));
} catch (IOException e) {
    System.out.println("Oops! Something went wrong with I/O!");
} catch (FileNotFoundException e) {
    System.out.println("Where did that file go?");
}
```

Error of the Week

What can you say about this program?

```
try {
    Files.readString(Path.of("data.txt"));
} catch (IOException e) {
    System.out.println("Oops! Something went wrong with I/O!");
} catch (FileNotFoundException e) {
    System.out.println("Where did that file go?");
}
```

```
error: exception FileNotFoundException has already been caught
```

Live Programming

- BankAccount
  - with `deposit` and `withdraw`.
  - with `InsufficientFundsException` and `TooManyFundsException`
  - the `transfer` method

Sources for Images and Slides

- https://introcs.cs.princeton.edu/java/lectures/
- https://www.esa.int/ESA_Multimedia/Images/2009/09/Explosion_of_first_Ariane_5_flight_June_4_1996
- https://commons.wikimedia.org/wiki/File:Fire_inside_an_abandoned_convent_in_Massueville,_Quebec,_Canada.jpg
- https://commons.wikimedia.org/wiki/File:Fire_inside_an_abandoned_convent_in_Massueville,_Quebec,_Canada.jpg
- https://commons.wikimedia.org/wiki/File:Fire_inside_an_abandoned_convent_in_Massueville,_Quebec,_Canada.jpg