# Lets Code!

Exceptions

# Exceptions, Assertions, Logging

# What we'll cover

# What we'll cover

Throwable Class

# What we'll cover

Throwable Class

Basic Exception Handling

# What we'll cover

Throwable Class

Basic Exception Handling

Better Exception Handling

# What we'll cover

Throwable Class

Basic Exception Handling

Better Exception Handling

Finally Keyword

# What we'll cover

Throwable Class

Basic Exception Handling

Better Exception Handling

Finally Keyword

Assertions

# What we'll cover

Throwable Class

Basic Exception Handling

Better Exception Handling

Finally Keyword

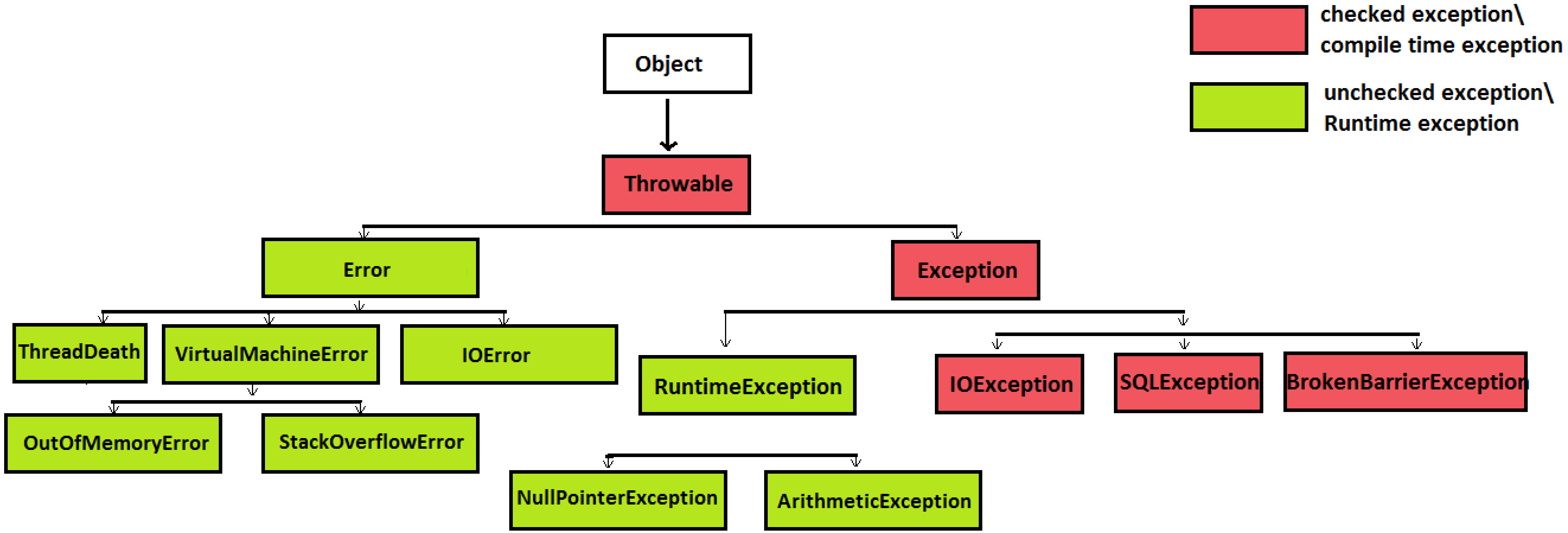
Assertions

Logging

# Throwable Class

- Throwable Hierarchy
- Errors
- Checked Exceptions
- Unchecked Exceptions

# Throwable Hierarchy



# Errors

- The `Error` hierarchy describes internal errors and resource exhaustion situations.
- Do not advertise internal java errors; Any code can potentially throw an `Error`.
- `IOError`, `StackOverflowError`, and `OutOfMemoryError` are a few of the commonly encountered `Errors`
  - `IOError` - serious I/O error has occurred
  - `StackOverflowError` - application recurses too deeply.
  - `OutOfMemoryError` - JVM cannot allocate an object because it is out of memory

# Unchecked Exceptions

- Any exception which derives from `Error` or `RuntimeException` class.
- An Exception subclassing `RuntimeException` is considered to be the programmer's fault.
    - `ArrayIndexOutOfBoundException` can be avoided by testing array index against array bounds
    - `NullPointerException` can be avoided by testing for null values.

# Checked Exceptions

- An Exception subclassing `IOException` is *potentially* not the programmer's fault.
  - `FileNotFoundException` can be thrown when trying to read from a remote file that a person incidentally removes.
  - `SQLException` can be thrown as a result of a faulty network connection.

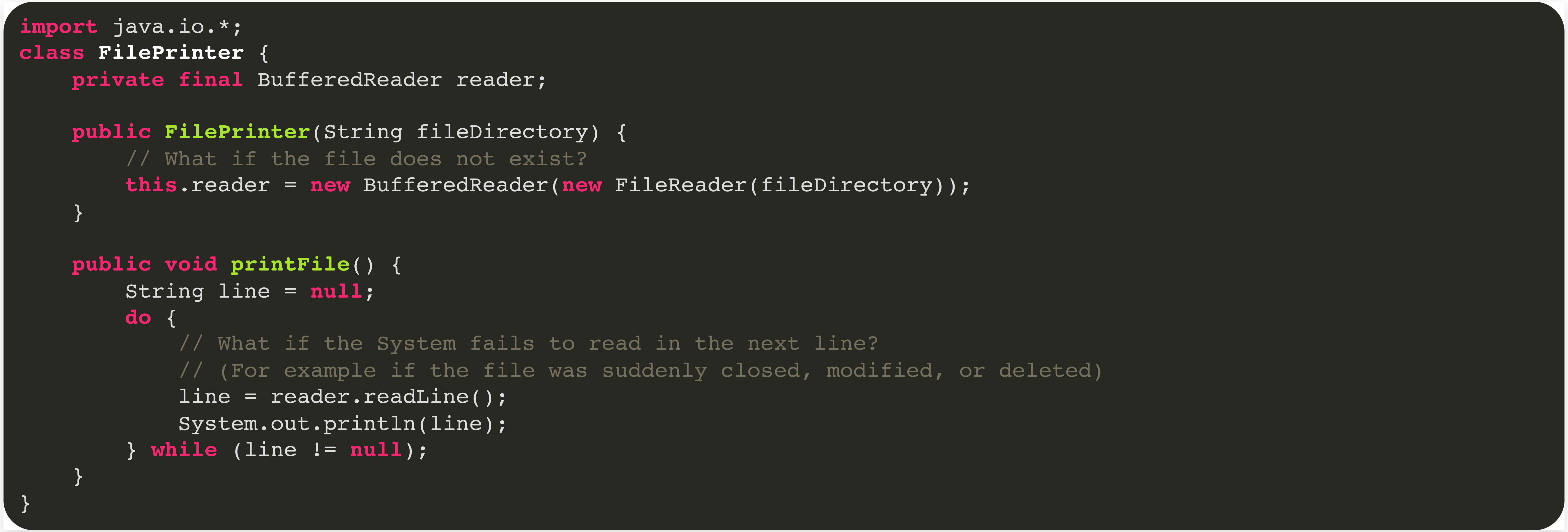
# Basic Exception Handling

- What is Exception Handling?
- Unhandled Exception Example
- Handled Exception Example

# What is Exception Handling?

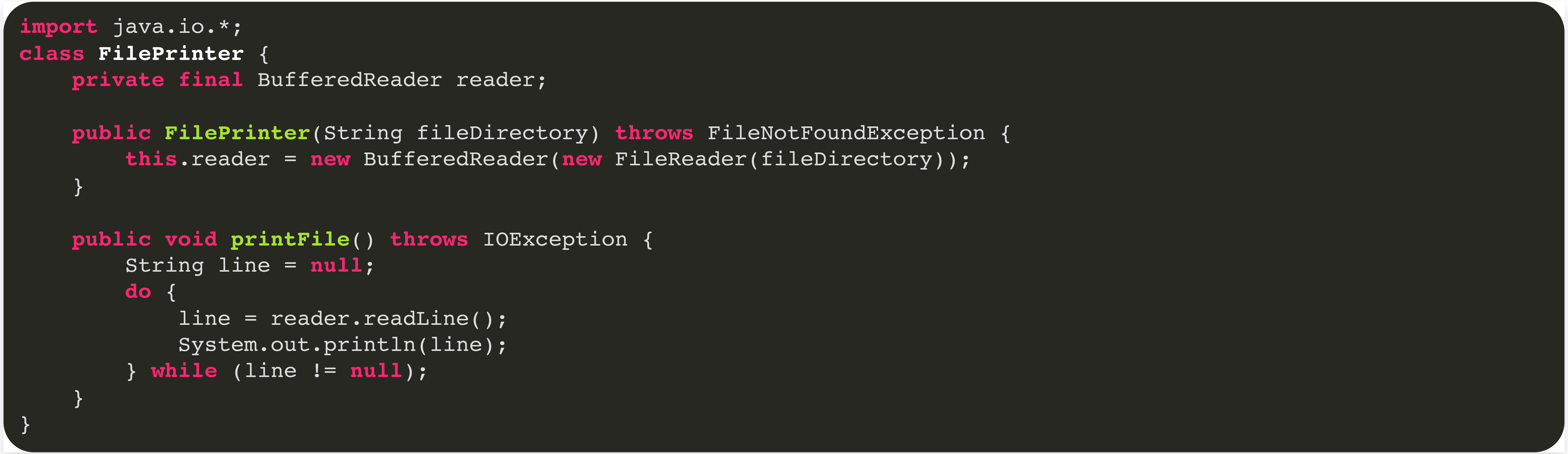
- For exceptional situations, Java uses a form of error trapping called, *exception handling*.
- Exception handling enables the author of the code to record and handle errors in their program.

# ~~Unchecked~~ Unhandled Exception Example; Compile Error



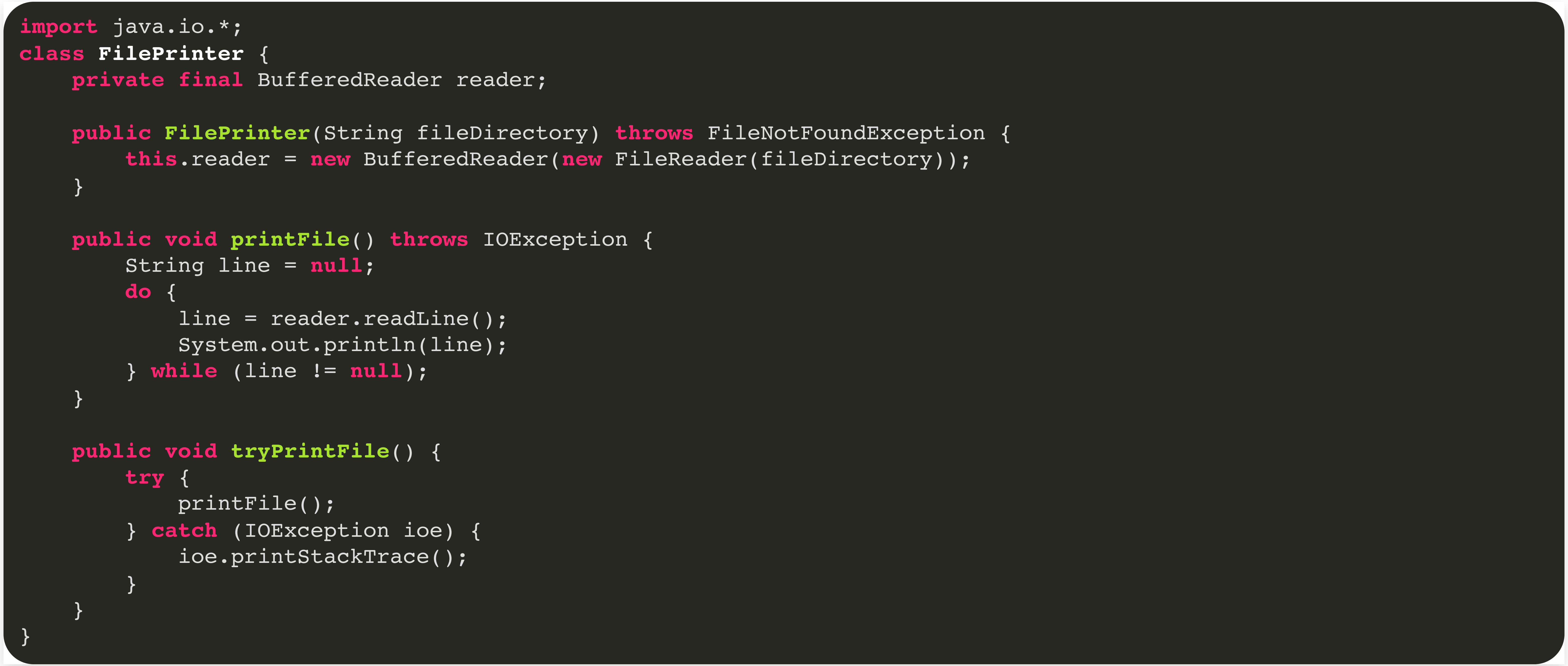
```java
import java.io.*;
class FilePrinter {
    private final BufferedReader reader;

    public FilePrinter(String fileDirectory) {
        // What if the file does not exist?
        this.reader = new BufferedReader(new FileReader(fileDirectory));
    }

    public void printFile() {
        String line = null;
        do {
            // What if the System fails to read in the next line?
            // (For example if the file was suddenly closed, modified, or deleted)
            line = reader.readLine();
            System.out.println(line);
        } while (line != null);
    }
}
```

# Exception Handling; Signature Throw Clause



```java
import java.io.*;
class FilePrinter {
    private final BufferedReader reader;

    public FilePrinter(String fileDirectory) throws FileNotFoundException {
        this.reader = new BufferedReader(new FileReader(fileDirectory));
    }

    public void printFile() throws IOException {
        String line = null;
        do {
            line = reader.readLine();
            System.out.println(line);
        } while (line != null);
    }
}
```

# Exception Handling; Try / Catch



```java
import java.io.*;
class FilePrinter {
    private final BufferedReader reader;

    public FilePrinter(String fileDirectory) throws FileNotFoundException {
        this.reader = new BufferedReader(new FileReader(fileDirectory));
    }

    public void printFile() throws IOException {
        String line = null;
        do {
            line = reader.readLine();
            System.out.println(line);
        } while (line != null);
    }

    public void tryPrintFile() {
        try {
            printFile();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
```

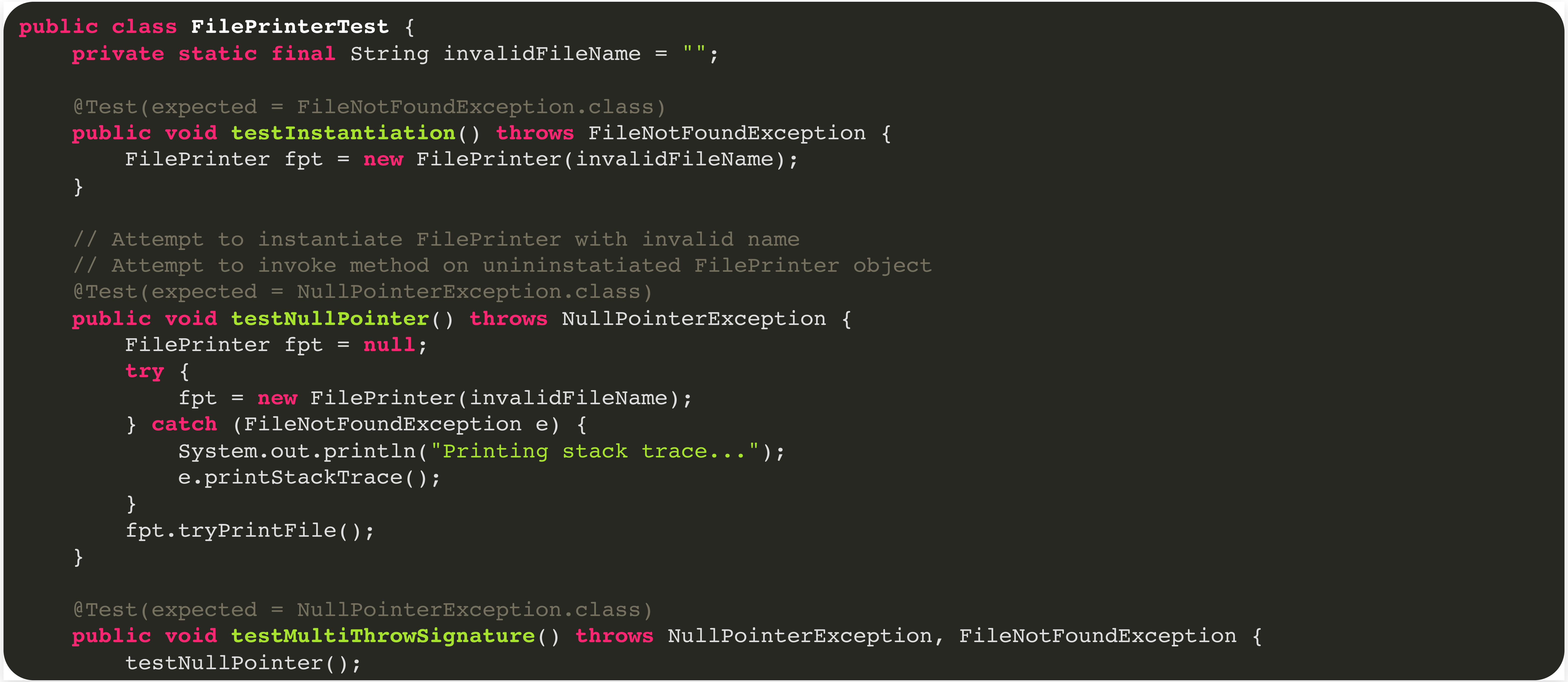
# Better Exception Handling

- Multi-Exception Handling
- Dynamic Exception Handling
- Uniform Exception Handling (Good)
- Uniform Exception Handling (Bad)
- How to throw an Exception
- Recursion ~~and~~ Exception Handling

# Multi-Exception Handling

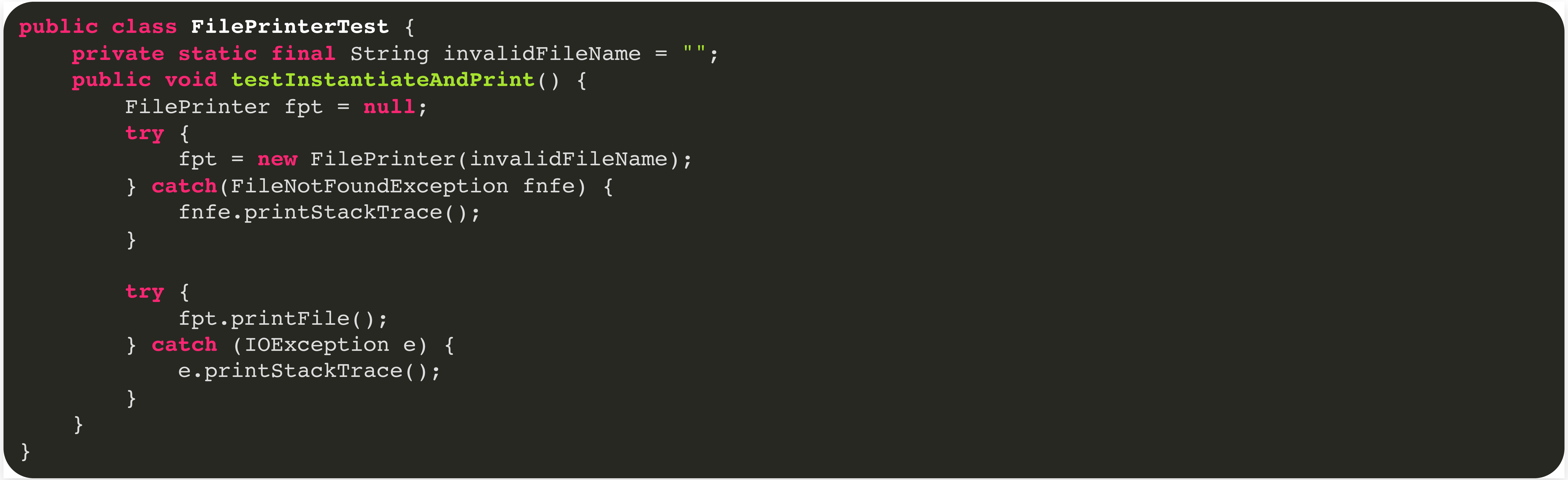
- Consider the case where multiple exceptions may be thrown.
- For example, in our `FilePrinter` class, the
  - constructor throws a `FileNotFoundException`
  - `printFile()` throws an `IOException`
- What if we wanted to create a `FilePrinter` object, then print its contents?

# Multi-Exception Handling Examples



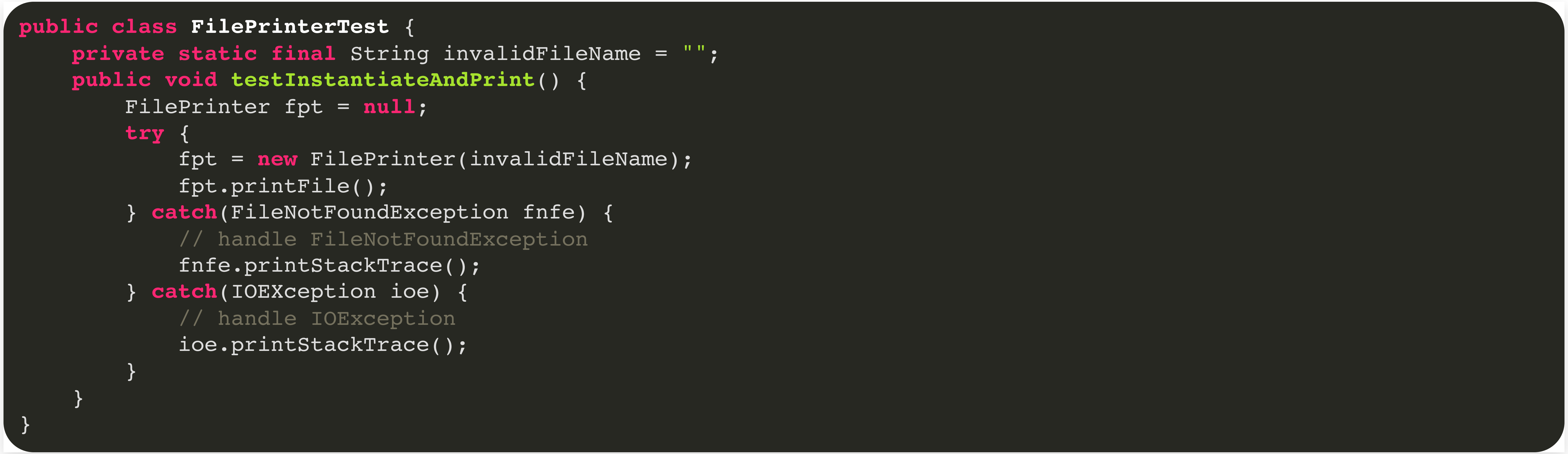
```java
public class FilePrinterTest {
    private static final String invalidFileName = "";

    @Test(expected = FileNotFoundException.class)
    public void testInstantiation() throws FileNotFoundException {
        FilePrinter fpt = new FilePrinter(invalidFileName);
    }

    // Attempt to instantiate FilePrinter with invalid name
    // Attempt to invoke method on unininstatiated FilePrinter object
    @Test(expected = NullPointerException.class)
    public void testNullPointer() throws NullPointerException {
        FilePrinter fpt = null;
        try {
            fpt = new FilePrinter(invalidFileName);
        } catch (FileNotFoundException e) {
            System.out.println("Printing stack trace...");
            e.printStackTrace();
        }
        fpt.tryPrintFile();
    }

    @Test(expected = NullPointerException.class)
    public void testMultiThrowSignature() throws NullPointerException, FileNotFoundException {
        testNullPointer();
```

# Dynamic Exception Handling; Expanded



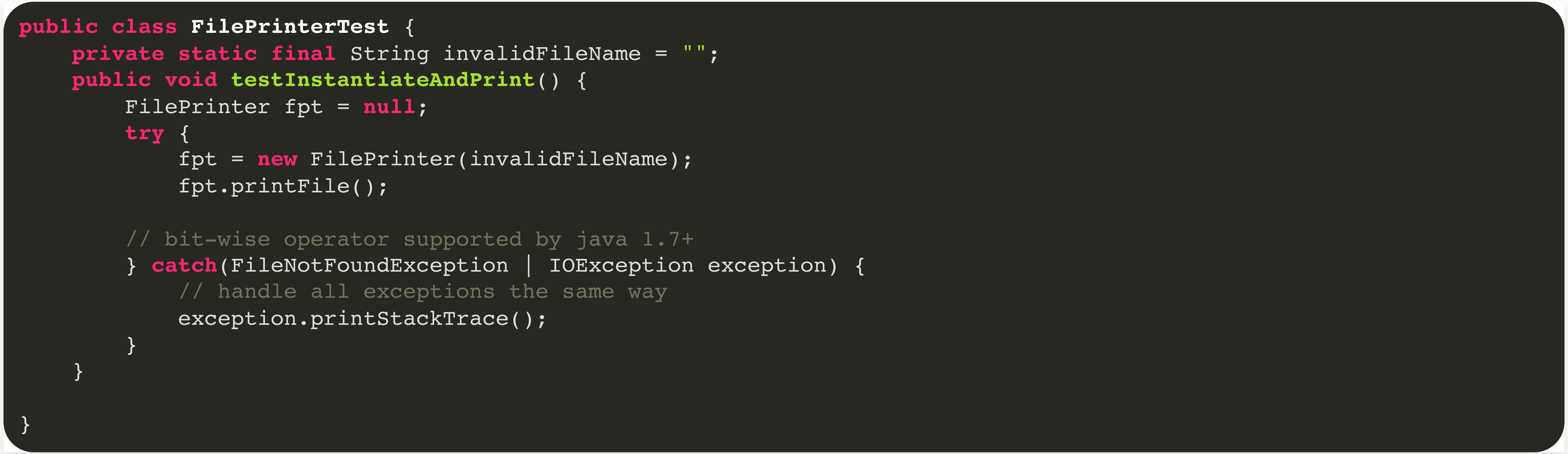
```java
public class FilePrinterTest {
    private static final String invalidFileName = "";
    public void testInstantiateAndPrint() {
        FilePrinter fpt = null;
        try {
            fpt = new FilePrinter(invalidFileName);
        } catch(FileNotFoundException fnfe) {
            fnfe.printStackTrace();
        }

        try {
            fpt.printFile();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

# Dynamic Exception Handling; Compressed



```java
public class FilePrinterTest {
    private static final String invalidFileName = "";
    public void testInstantiateAndPrint() {
        FilePrinter fpt = null;
        try {
            fpt = new FilePrinter(invalidFileName);
            fpt.printFile();
        } catch(FileNotFoundException fnfe) {
            // handle FileNotFoundException
            fnfe.printStackTrace();
        } catch(IOException ioe) {
            // handle IOException
            ioe.printStackTrace();
        }
    }
}
```

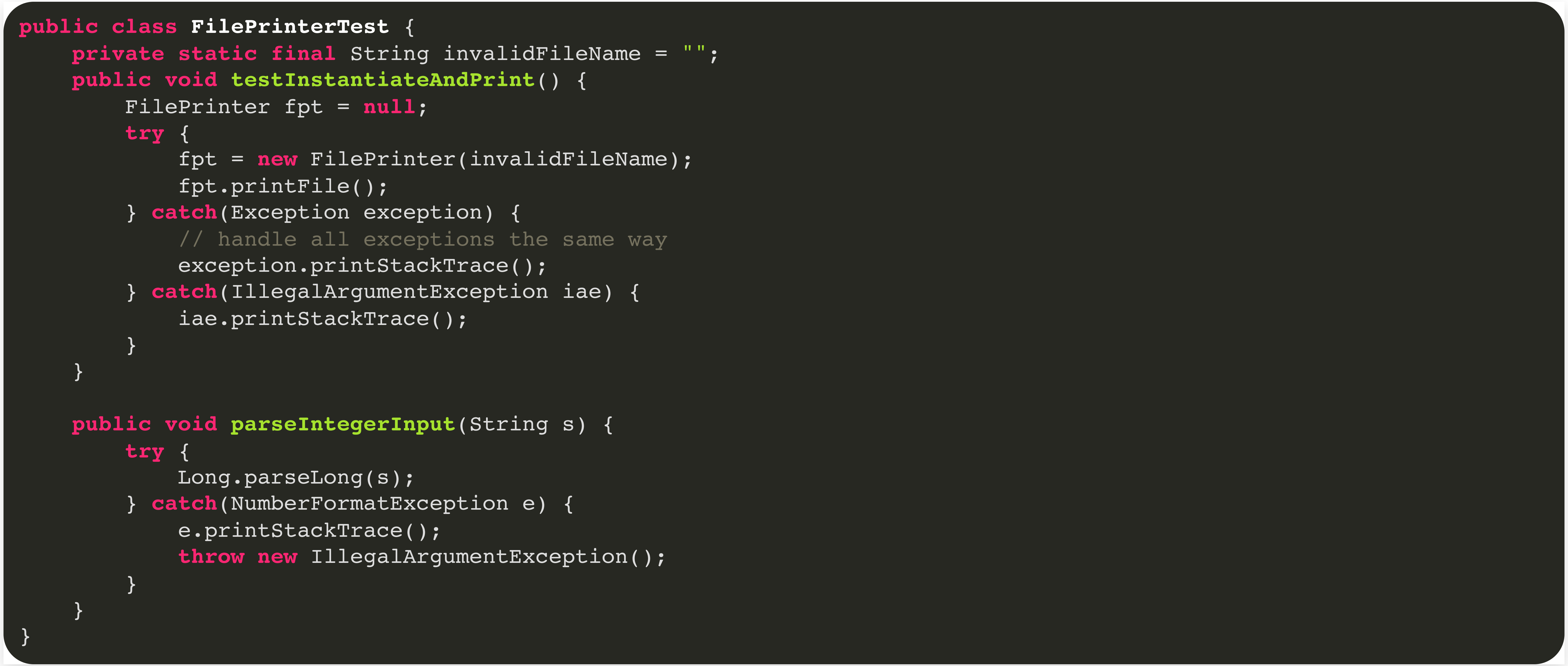
# Uniform Handling Of Exceptions (Good)



```java
public class FilePrinterTest {
    private static final String invalidFileName = "";
    public void testInstantiateAndPrint() {
        FilePrinter fpt = null;
        try {
            fpt = new FilePrinter(invalidFileName);
            fpt.printFile();

        // bit-wise operator supported by java 1.7+
        } catch(FileNotFoundException | IOException exception) {
            // handle all exceptions the same way
            exception.printStackTrace();
        }
    }

}
```

- Each expected exception in this class is explicitly named.
- The handling of each of them is uniform.

# Uniform Handling Of Exceptions (Bad)



```java
public class FilePrinterTest {
    private static final String invalidFileName = "";
    public void testInstantiateAndPrint() {
        FilePrinter fpt = null;
        try {
            fpt = new FilePrinter(invalidFileName);
            fpt.printFile();
        } catch(Exception exception) {
            // handle all exceptions the same way
            exception.printStackTrace();
        } catch(IllegalArgumentException iae) {
            iae.printStackTrace();
        }
    }

    public void parseIntegerInput(String s) {
        try {
            Long.parseLong(s);
        } catch(NumberFormatException e) {
            e.printStackTrace();
            throw new IllegalArgumentException();
        }
    }
}
```

# Recursion ~~and~~ Exception Handling

- DON'T DO IT!
- Recursion and Exception Handling do not go together
- Exceptions keep track of all pending method calls
- By nature, recursion pends method calls **`n`** levels deep, where **`n`** is the recursive depth of the method call.
- Combining recursion and exception handling can result in very strange `StackTraces`

# Finally Keyword

- Purpose
- Conditions under which `finally` block is executed
- Syntax
- Decoupling `finally` clause from `try/catch` clauses

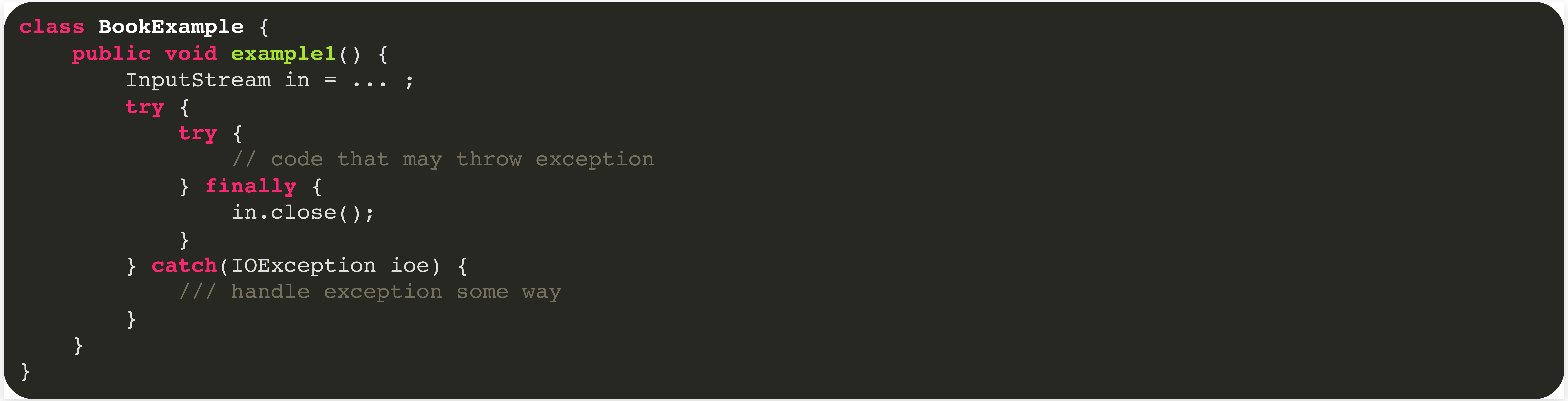
# Purpose

- When code throws an exception, it stops processing the remaining code in the scope, then exits the method.
- If the method has aqcuired some local resource, then this can become an issue; The program will cease execution, and hold the resource indefinitely.
- The finally clause executes whether or not an exception was code.

# Conditions under which `finally` block is executed

1. If no exception are thrown.
2. If exception outside `try` block is thrown.
3. If an exception is thrown in a `catch` clause.
4. The program skips to the `finally` clause, if the `catch` clause does not throw an exception.

# Decoupling `finally` clause from `try/catch` clauses



```
class BookExample {
    public void example1() {
        InputStream in = ... ;
        try {
            try {
                // code that may throw exception
            } finally {
                in.close();
            }
        } catch(IOException ioe) {
            /// handle exception some way
        }
    }
}
```

# Assertions

- Assertions are commonly used idiom of defensive programming.
- Java has a keyword **`assert`**, which takes two forms:

  1. **`assert condition;`**
  2. **`assert condition : expression;`**

- **`assert`** evaluates a condition, then throws an **`AssertionError`** if it is false. The second argument *expression* is a message String.

# Toggling Assert Statements

- By default, assertions are disabled; If an assert statement is passed `false`, no exception is thrown.
- Assertions can be enabled by running the program with the `-ea` option.
- `java -ea MyProject` enables for entire project
- `java -ea:MyClass -ea:com.codedifferently.MyProject` enables for `MyClass`

# When To Use

- Assertion failures are intended to be fatal, unrecoverable errors
- Assertion checks are turned on only during development and testing
- As an additional check against uncanny method returns.

# Logging

- It's common to use `System.out.println` to check against troublesome code.
- Once the issue is resolved, these statements are usually removed, or commented out.
- Later, if the issue persists, the print statements are re-inserted.
- The Logging API is designed to overcome this issue.

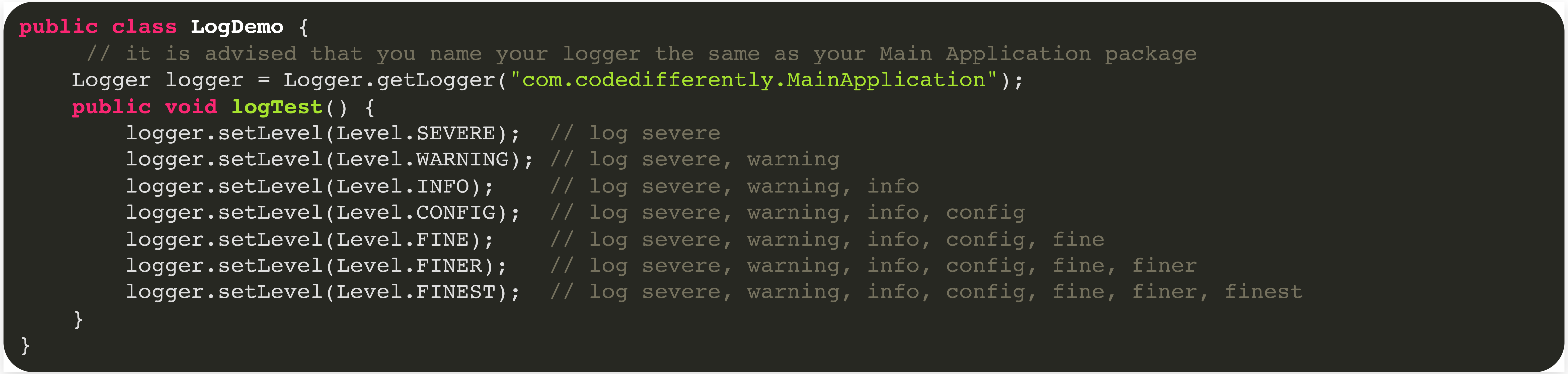
# Principal advantages of Logging API

- It's easy to (un)suppress all log records, or just those below a certain level.
- Suppressed logs are inexpensive; The penalty for leaving them in your code is minimal.
- Log records can be directed to different handlers; Console display, writing to file / database, etc.
- Log records can be formatted; For example, plaint ext, or XML
- Logging configuration is controlled by configuration file; Applications can replace this mechanism

# The 7 Logging Levels

- By default, loggers will log all messages of `INFO` or higher to console.
- `SEVERE`
- `WARNING`
- `INFO`
- `CONFIG`
- `FINE`
- `FINER`
- `FINEST`

# Syntax



```
public class LogDemo {
     // it is advised that you name your logger the same as your Main Application package
    Logger logger = Logger.getLogger("com.codedifferently.MainApplication");
    public void logTest() {
        logger.setLevel(Level.SEVERE);  // log severe
        logger.setLevel(Level.WARNING); // log severe, warning
        logger.setLevel(Level.INFO);    // log severe, warning, info
        logger.setLevel(Level.CONFIG);  // log severe, warning, info, config
        logger.setLevel(Level.FINE);    // log severe, warning, info, config, fine
        logger.setLevel(Level.FINER);   // log severe, warning, info, config, fine, finer
        logger.setLevel(Level.FINEST);  // log severe, warning, info, config, fine, finer, finest
    }
}
```