UNIVERSITETI METROPOLITAN TIRANA

Course: Object Oriented Programming

File I/O and Exception Handling

Building Resilient Systems

Evis Plaku

Two essential mechanisms for writing production code that survives reality's chaos

- Programs must persist data beyond runtime to application shutdown
- Unpredictable failures happen: missing files, network errors, or invalid input
- Need systematic ways to detect, handle and recover from errors gracefully

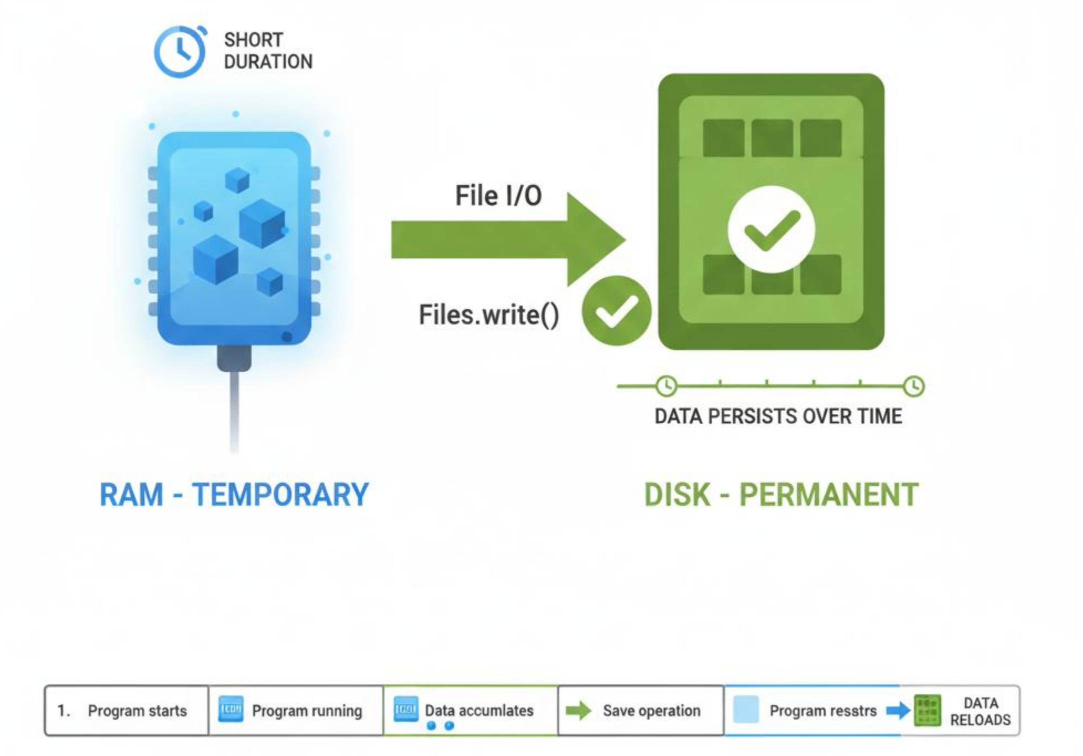
DATA IN RAM
(TEMPORARY)
ERROR
PROGRAM RUNNING
PROGRAM ENDS
ALL DATA GONE
DATA ON DISK
(SOLID STORAGE)
DISK STORAGE

In-memory data vanishes when the application stops running

- Variables live in RAM; temporarily cleared when program ends

- Real systems must **store data permanently** to disk or database

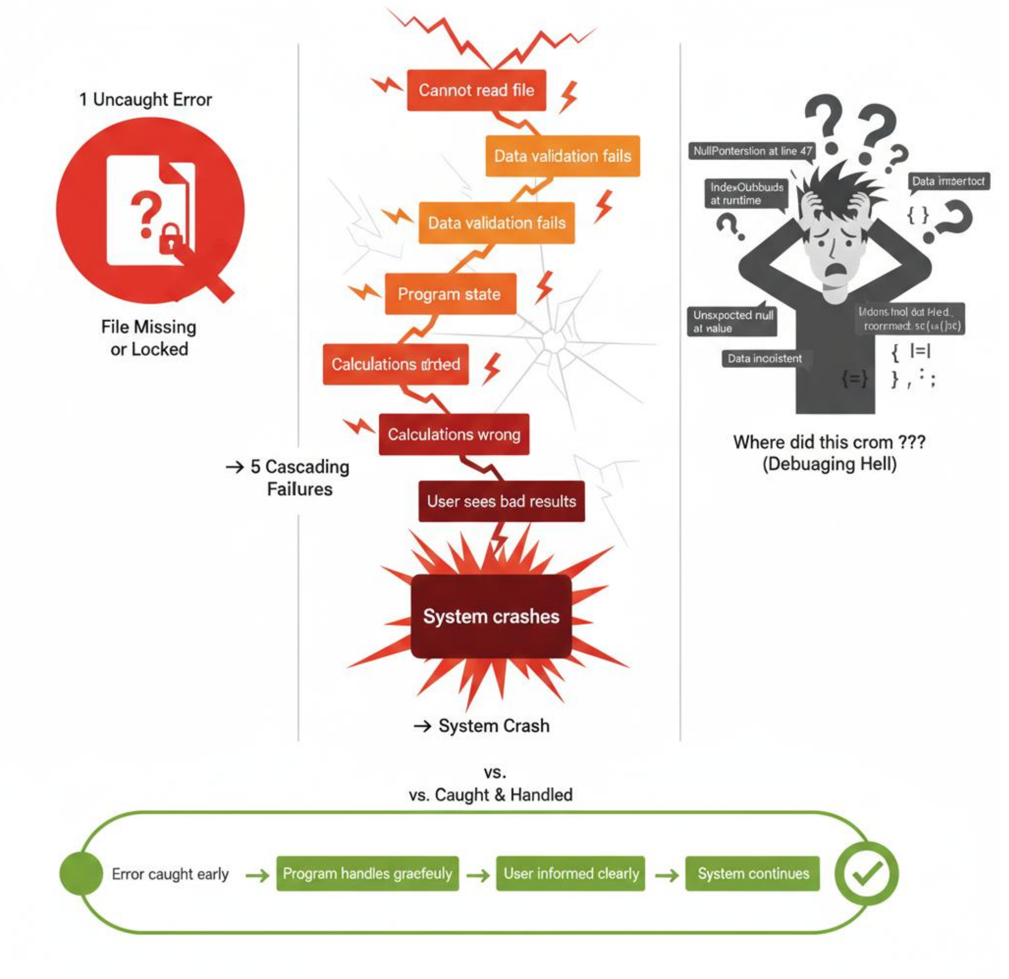
- File I/O enables programs to read and write **persistent** storage



THE ERROR REALITY

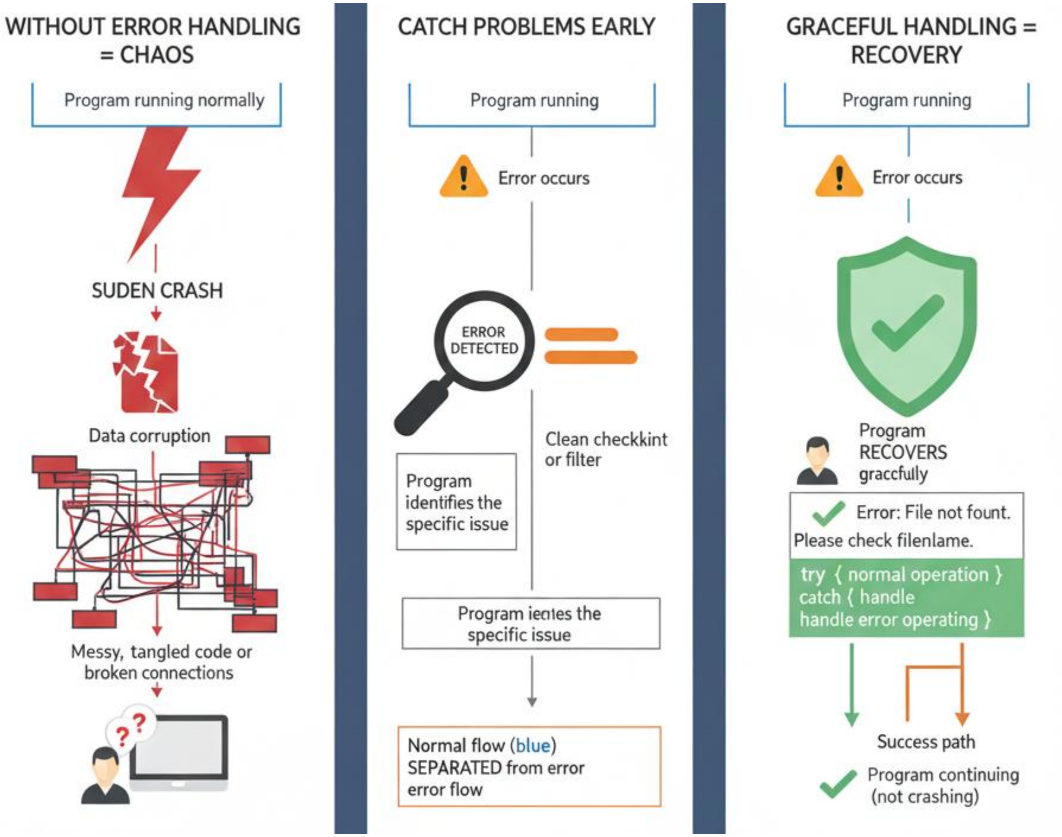
Code fails for reasons beyond your control

- Files might not exist or be locked by other programs
- Assumptions about data often prove wrong at runtime
- Ignoring errors creates cascading failures hard to debug

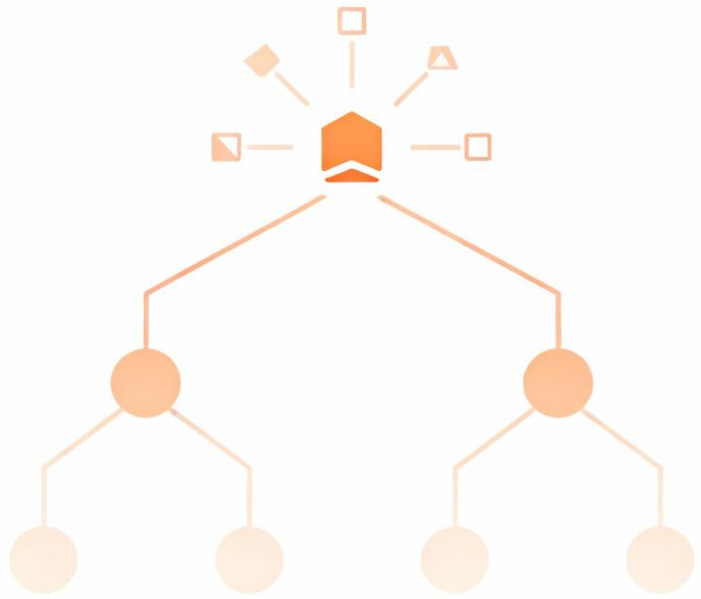


Exceptions are controlled interruptions allowing graceful handling of unexpected situations

- Program can detect problems before they cause data corruption
- Code can attempt **recovery** or **fail cleanly** with clear messages
- Separates normal flow from error


WITHOUT ERROR HANDLING = CHAOS
Program running normally
SUDEN CRASH
Data corruption
Messy, tangled code or broken connections
CATCH PROBLEMS EARLY
Program running
Error occurs
ERROR DETECTED
Clean checkkint or filter
Program identifies the specific issue
Program iectes the specific issue
Normal flow (blue) SEPARATED from error error flow
GRACEFUL HANDLING = RECOVERY
Program running
Error occurs
Program RECOVERS graccfully
Success path
Program continuing (not crashing)

```
✓ Error: File not fount.
Please check filenlame.
try { normal operation }
catch { handle
handle error operating }
```

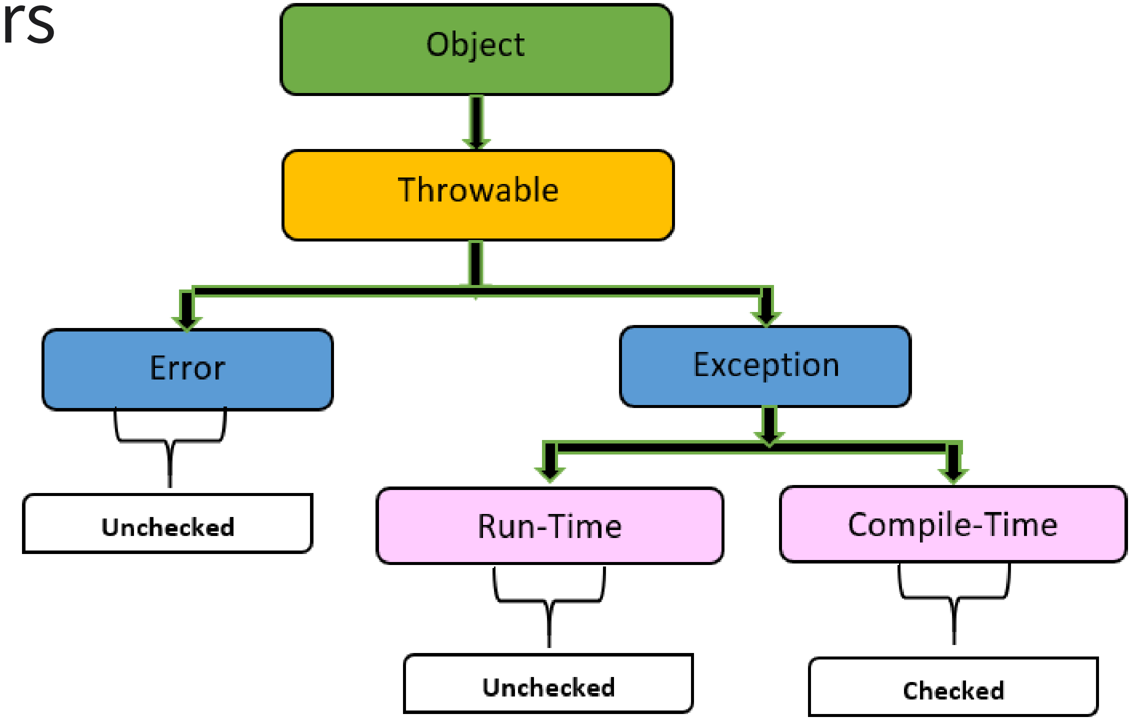


Core Concepts

Exceptions

Java exceptions form a hierarchy from general Throwable down to specific error types

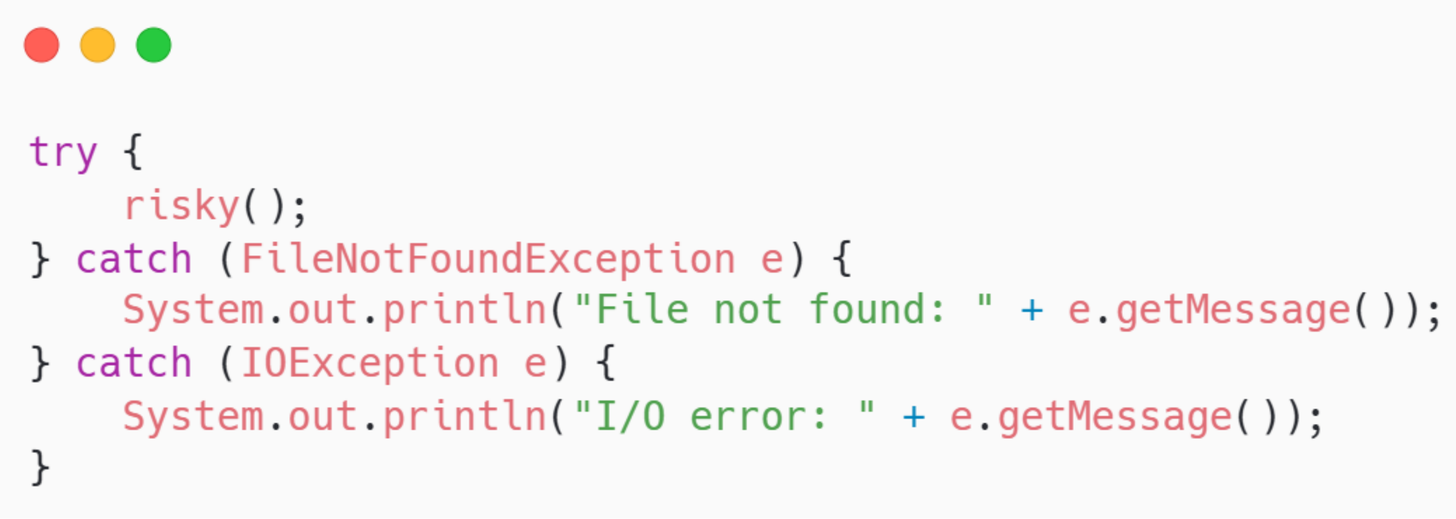
- Throwable is root of all exceptions errors are most severe system level
- Exception is for recoverable errors checked at compile time
- Specific exceptions like `IOException` `FileNotFoundException` extend Exception



TRY – CATCH BLOCKS

Try-catch wraps code that might fail and provides recovery handlers

- Try block contains risky code that might throw an exception
- Catch block executes if exception occurs matching the declared type



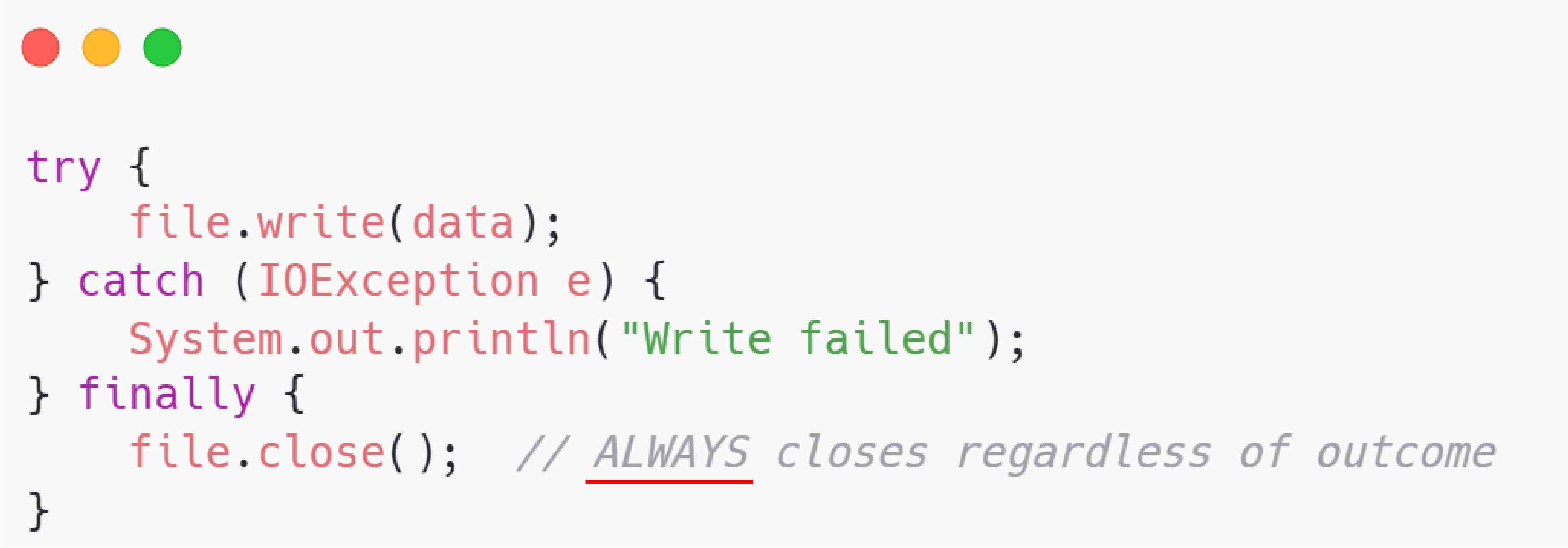
```java
try {
    risky();
} catch (FileNotFoundException e) {
    System.out.println("File not found: " + e.getMessage());
} catch (IOException e) {
    System.out.println("I/O error: " + e.getMessage());
}
```

- Multiple catch blocks handle different exception types differently

FINALLY BLOCK

“Finally” always executes whether exception occurred or not guaranteeing cleanup

- Finally block runs after try completes or catch handles exception
- Used for cleanup: closing files streams connections releasing resources



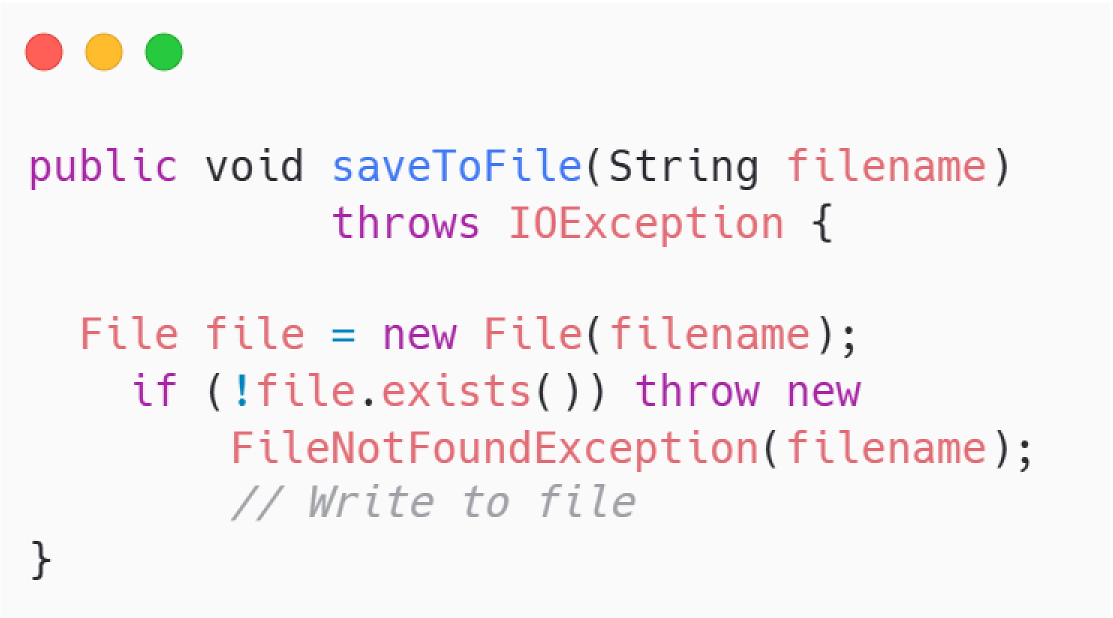
```
try {
    file.write(data);
} catch (IOException e) {
    System.out.println("Write failed");
} finally {
    file.close();  // ALWAYS closes regardless of outcome
}
```

- Most reliable place to guarantee resource release

THROWS DECLARATION

Methods can declare exceptions they throw passing responsibility to caller

- Throws keyword lists exceptions
- Caller must handle those exceptions or declare throws itself
- Propagates exception handling responsibility up the call stack

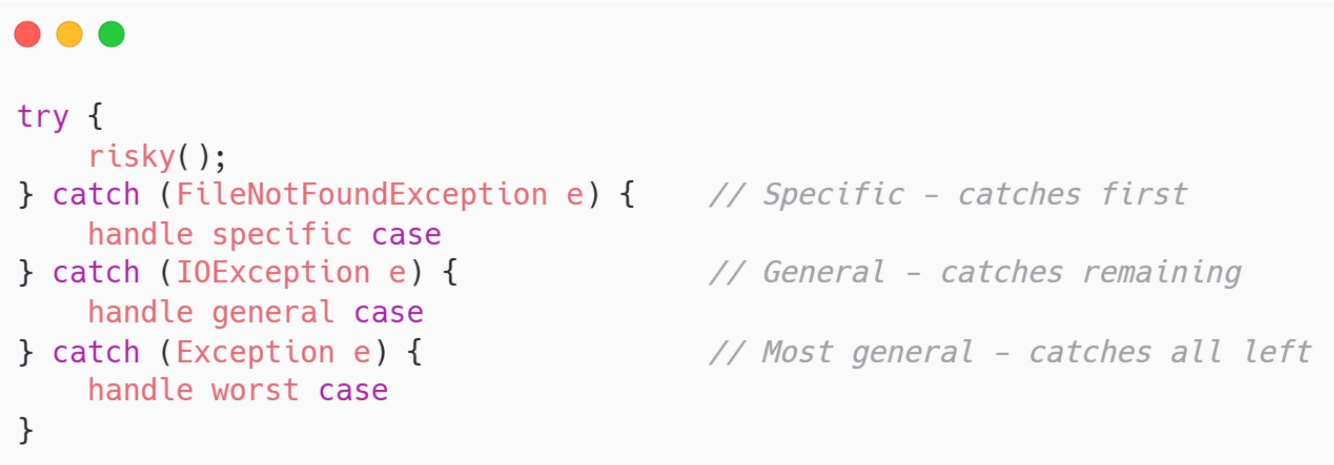


```java
public void saveToFile(String filename)
            throws IOException {

  File file = new File(filename);
    if (!file.exists()) throw new
        FileNotFoundException(filename);
        // Write to file
}
```

EXCEPTION PRIORITY ORDER

Catch blocks execute from the most specific to the most general catching first match

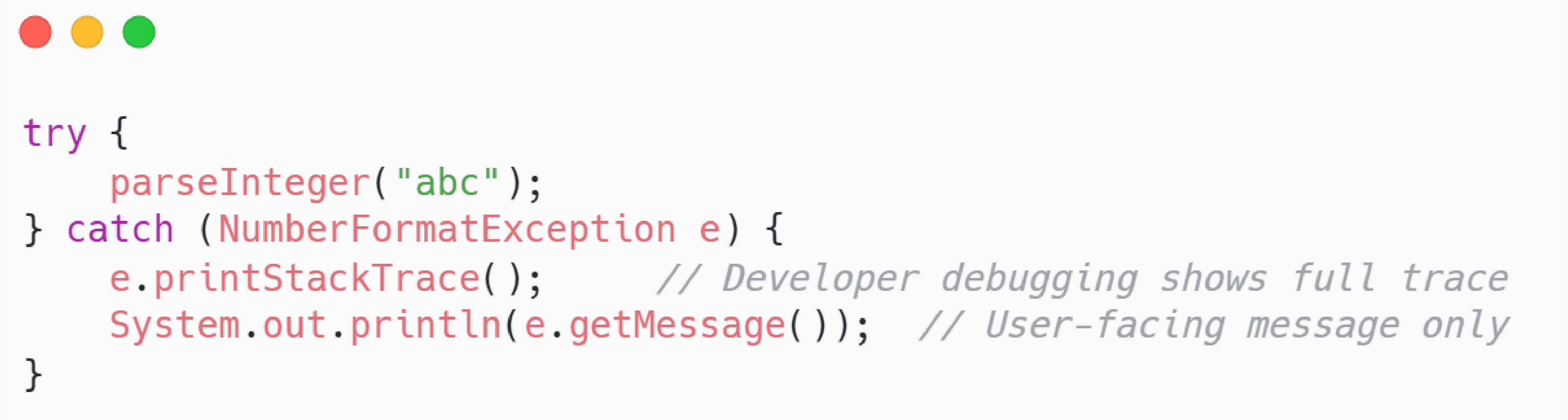
- Specific exceptions must come before general ones
- FileNotFoundException is more specific than IOException so must come first
- Compiler enforces this



```
try {
    risky();
} catch (FileNotFoundException e) {     // Specific - catches first
    handle specific case
} catch (IOException e) {               // General - catches remaining
    handle general case
} catch (Exception e) {                 // Most general - catches all left
    handle worst case
}
```

Exception objects contain crucial debugging information message and call stack

- getMessage() returns human-readable description of what went wrong
- printStackTrace() prints full call stack showing where exception originated



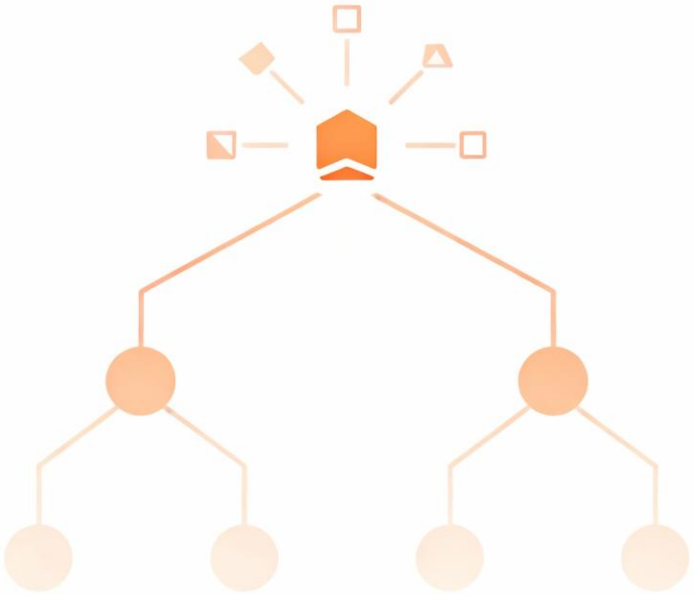
```
try {
    parseInteger("abc");
} catch (NumberFormatException e) {
    e.printStackTrace();     // Developer debugging shows full trace
    System.out.println(e.getMessage());  // User-facing message only
}
```

- Use both for debugging but only show message to end users

COMMON EXCEPTION TYPES

Understanding common exceptions helps write appropriate handlers for typical problems

- IOException: generic I/O problem; file doesn't exist; permission denied
- FileNotFoundException: specific file not found extends IOException
- NumberFormatException: string cannot convert to number invalid format



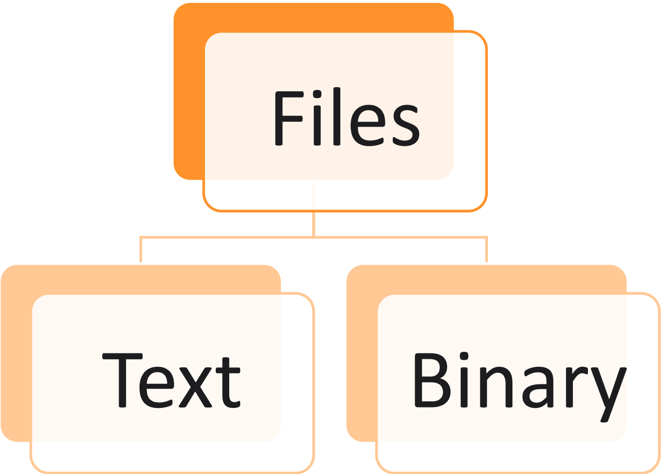
Core Concepts

Files

TEXT VS BINARY FILES

Text files store human-readable characters
Binary files store arbitrary byte sequences

- Text files contain characters encoded as bytes readable with text editor
- CSV JSON XML files are text human-readable structured formats
- Binary files contain serialized objects or raw data not human-readable



Java provides several approaches to reading text files from lines to custom parsing

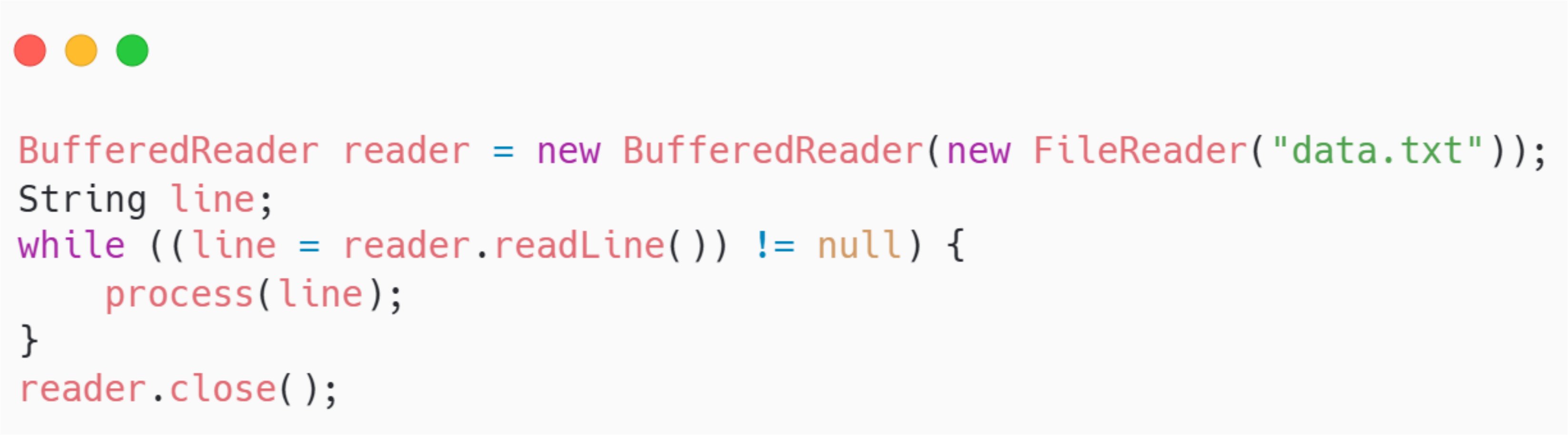
- BufferedReader wraps FileReader. It provides efficient line-by-line reading (`readLine()`)

- Scanner wraps input stream and provides `nextLine()` and `next()` method for parsing tokens

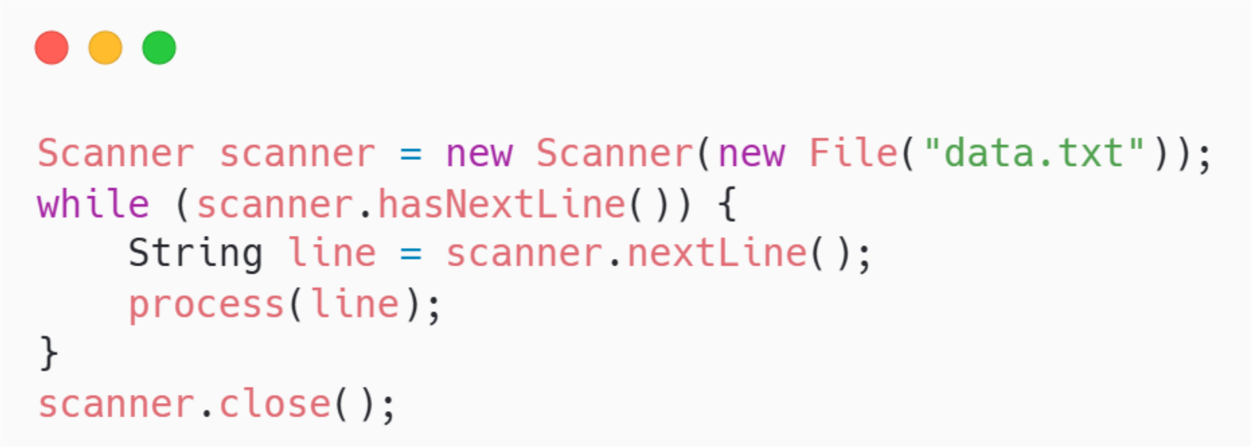
- Files.readAllLines() reads entire file into list of strings

READING TEXT FILES

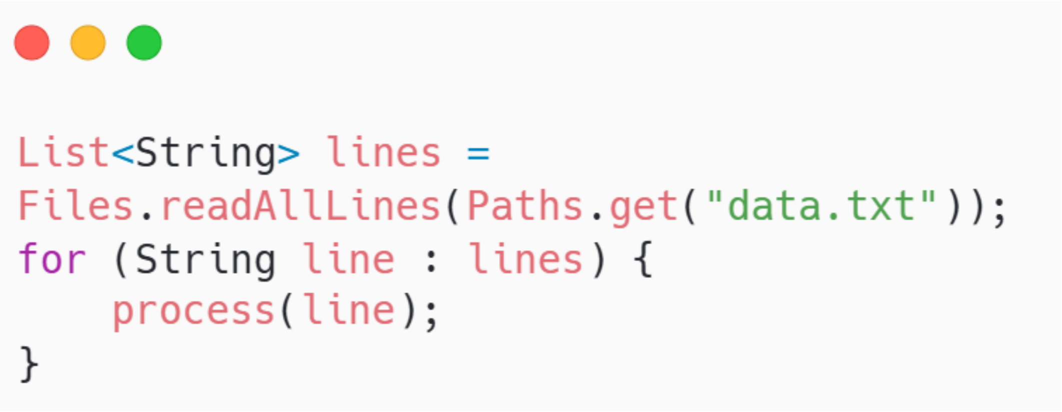
Java provides several approaches to reading text files from lines to custom parsing



```java
BufferedReader reader = new BufferedReader(new FileReader("data.txt"));
String line;
while ((line = reader.readLine()) != null) {
    process(line);
}
reader.close();
```



```java
Scanner scanner = new Scanner(new File("data.txt"));
while (scanner.hasNextLine()) {
    String line = scanner.nextLine();
    process(line);
}
scanner.close();
```



```java
List<String> lines =
Files.readAllLines(Paths.get("data.txt"));
for (String line : lines) {
    process(line);
}
```

WRITING TEXT FILES

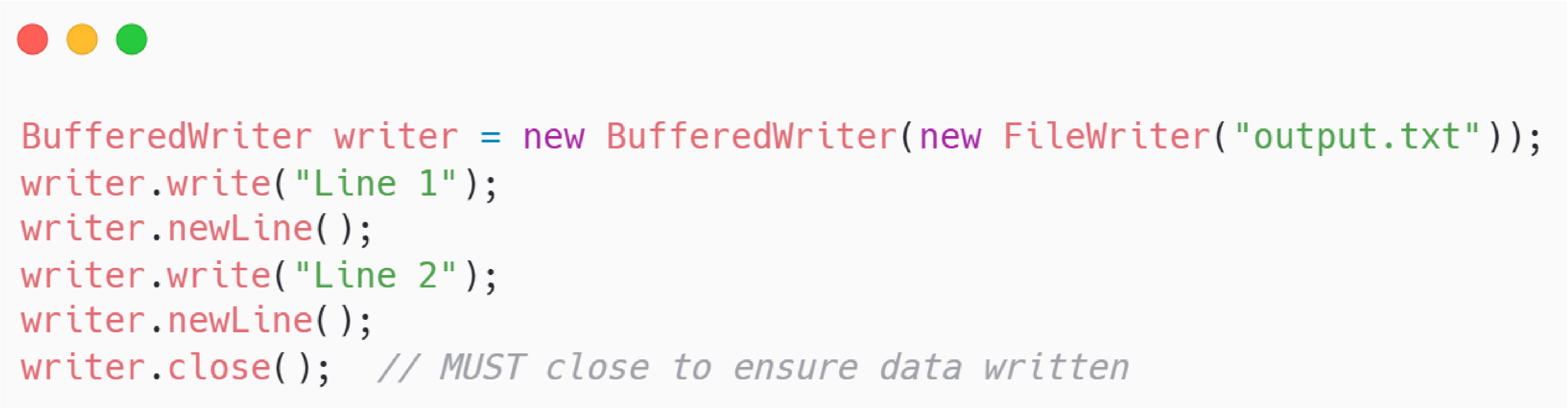
Writing creates new files or overwrites existing ones with character data

- FileWriter opens file for writing or creates new file if not exist

- BufferedWriter wraps FileWriter for efficient writing adds buffering

- write() outputs strings; newLine() adds platform-specific line break

Java provides several approaches to reading text files from lines to custom parsing

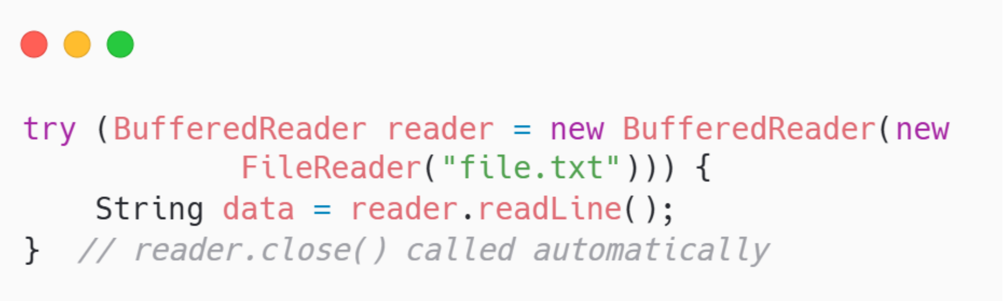


```java
BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt"));
writer.write("Line 1");
writer.newLine();
writer.write("Line 2");
writer.newLine();
writer.close();  // MUST close to ensure data written
```

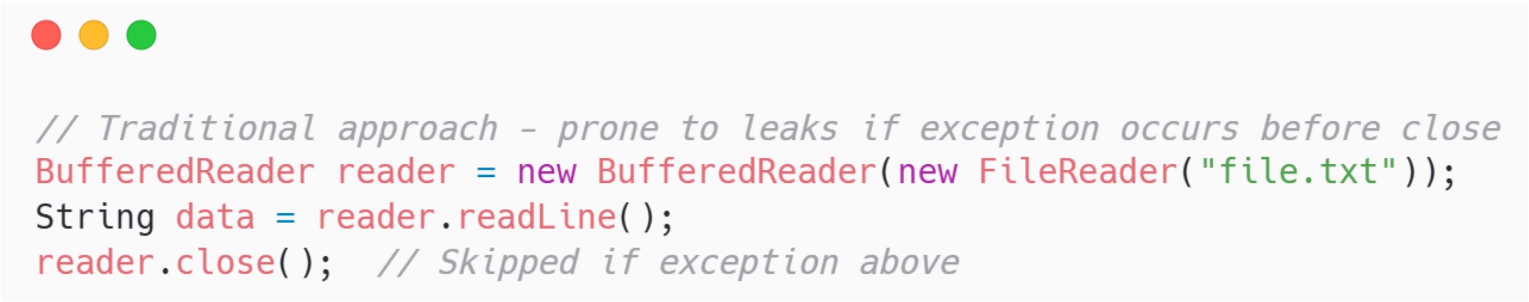
Must close file to flush buffered data and release system resources

Resources must be properly closed

- Opening files allocates system resources
- Forgetting to close exhausts file descriptors
  System eventually refuses new files



```
try (BufferedReader reader = new BufferedReader(new
                FileReader("file.txt"))) {
    String data = reader.readLine();
}  // reader.close() called automatically
```

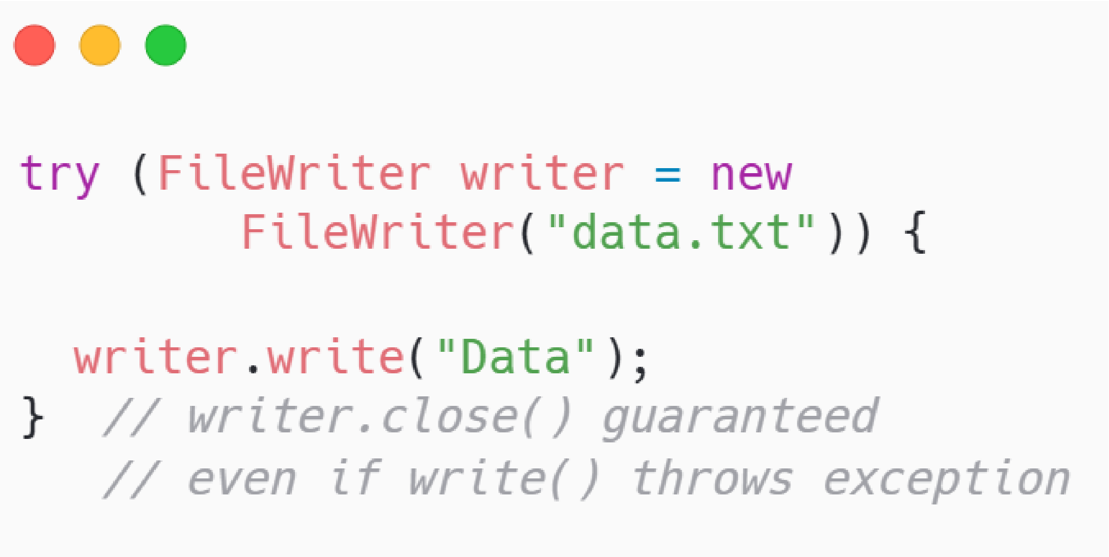


```
// Traditional approach - prone to leaks if exception occurs before close
BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
String data = reader.readLine();
reader.close();  // Skipped if exception above
```

- Finally block or try-with-resources ensures closure even if exceptions occur

Try-with-resources automatically closes resources

- Declares resource in parentheses after try keyword
- Resource automatically closed after try block completes successfully or with exception
- Cleaner and safer than manual close in finally blocks



```java
try (FileWriter writer = new
        FileWriter("data.txt")) {

  writer.write("Data");
}  // writer.close() guaranteed
   // even if write() throws exception
```

Paths represent file locations relative to working directory or absolute filesystem paths

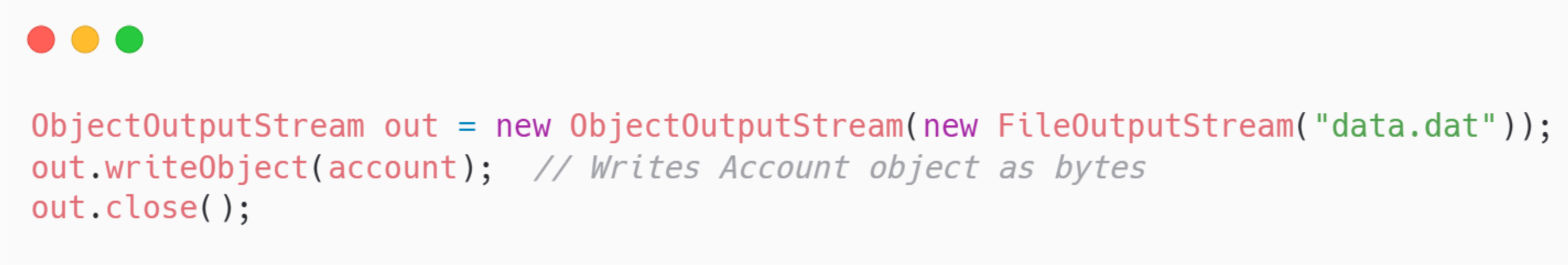
- Relative paths: "data/customers.txt" relative to program working directory
- Absolute paths: "/home/user/data/customers.txt" full path from filesystem root

- `Paths.get()` creates Path objects
- `File.exists()` checks if file present
- `canRead()` and `canWrite()` check permissions

Serialization converts objects to byte streams
Deserialization reconstructs them

- Serialization writes object state to bytes enabling persistence or network transmission

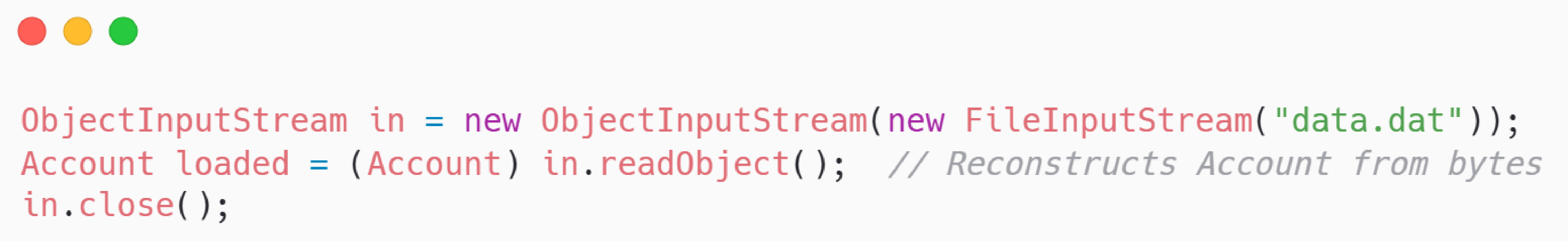
- ObjectOutputStream writes serializable objects to files or streams



```java
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("data.dat"));
out.writeObject(account);  // Writes Account object as bytes
out.close();
```

Serialization converts objects to byte streams Deserialization reconstructs them

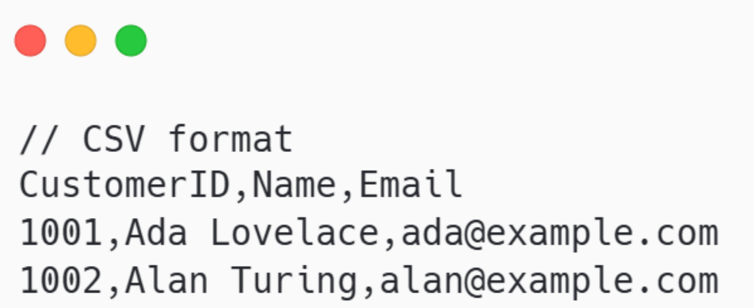
- ObjectInputStream reads bytes back reconstructs objects from saved state
- Objects must implement Serializable interface



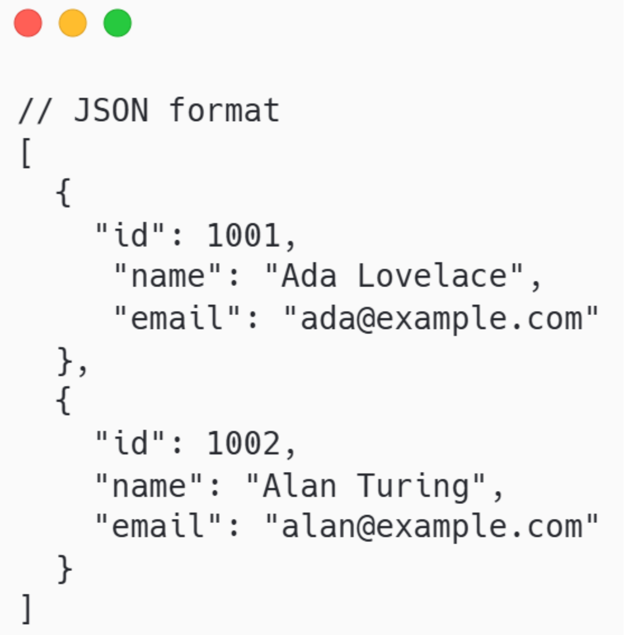
```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.dat"));
Account loaded = (Account) in.readObject();  // Reconstructs Account from bytes
in.close();
```

CSV, JSON, XML formats structure data as text parseable without binary serialization

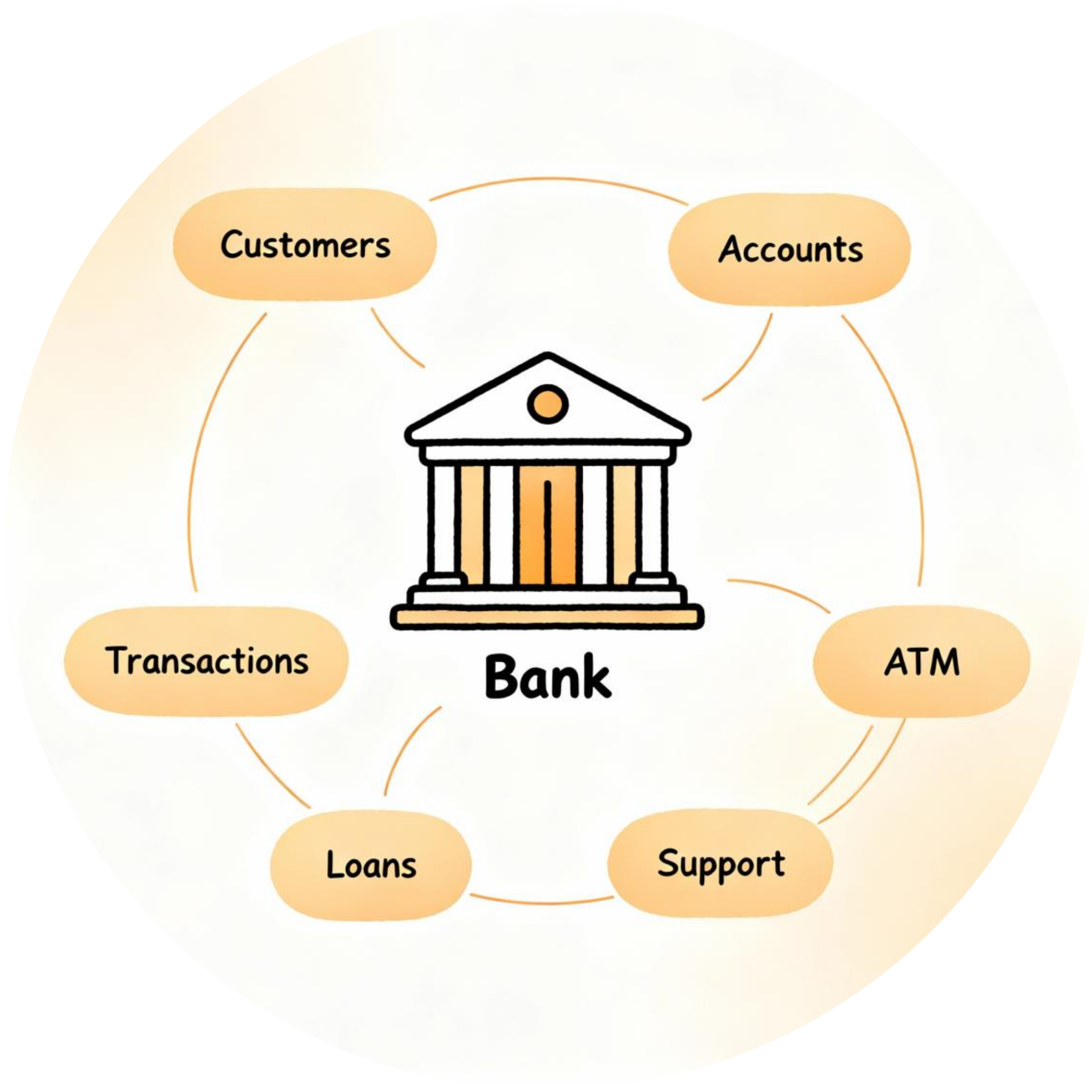
- CSV comma-separated values stores tabular data rows and columns simple portable
- JSON JavaScript Object Notation represents hierarchical data human-readable structured
- All human-readable languages debuggable flexible platform-independent



```
// CSV format
CustomerID,Name,Email
1001,Ada Lovelace,ada@example.com
1002,Alan Turing,alan@example.com
```



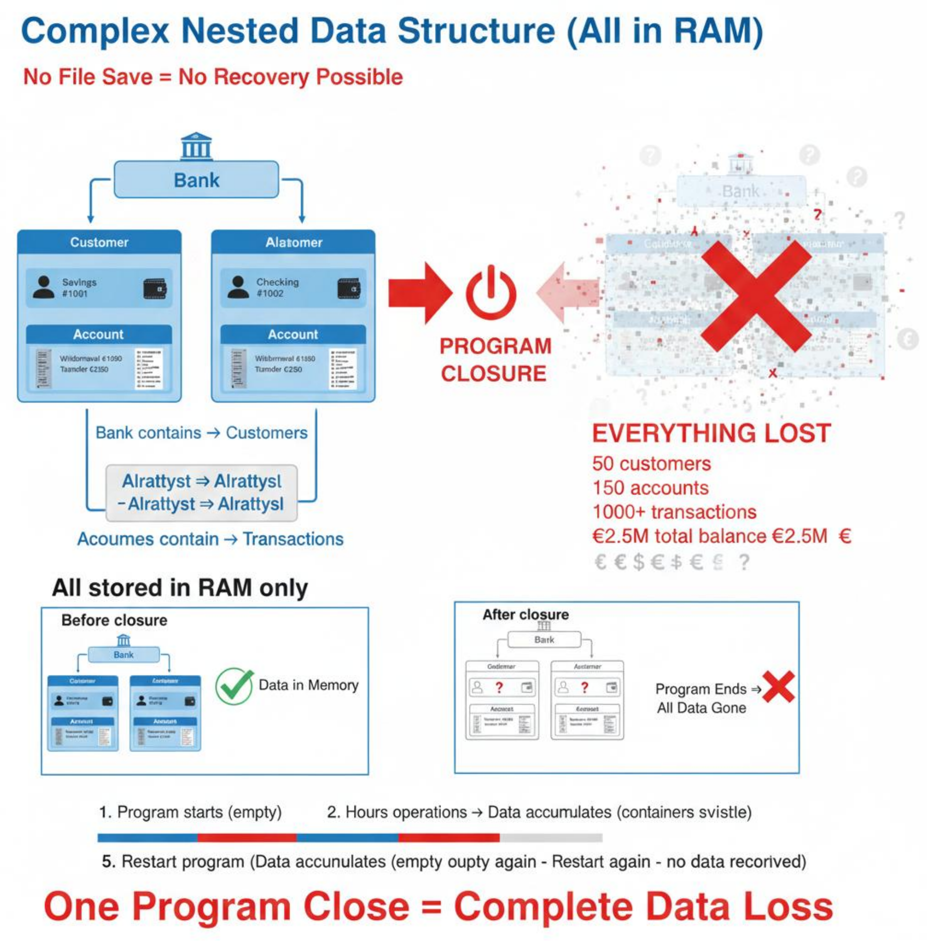
```
// JSON format
[
  {
    "id": 1001,
     "name": "Ada Lovelace",
     "email": "ada@example.com"
  },
  {
    "id": 1002,
    "name": "Alan Turing",
    "email": "alan@example.com"
  }
]
```



Bank System Example

Version 5 had all classes and accounts in memory lost on program termination

- Bank stores customers ArrayList
  Customer stores accounts ArrayList
- Account stores transaction history
  Immutable transactions record all changes
- No persistence everything gone when program ends no way to recover data



Complex Nested Data Structure (All in RAM)

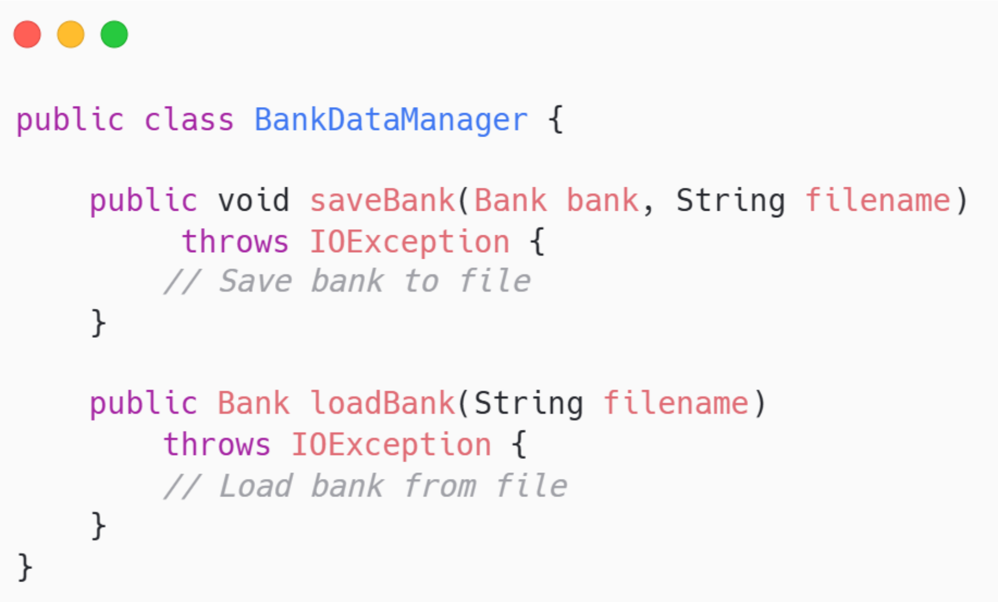
No File Save = No Recovery Possible

One Program Close = Complete Data Loss

THE BANK DATA MANAGER CLASS

New class centralizes all file I/O for saving and loading complete bank state

- Methods: saveBank() saves entire bank to file loadBank() restores from file
- Handles all IOException thrown by file operations
- Single responsibility: manage persistence all domain classes unchanged

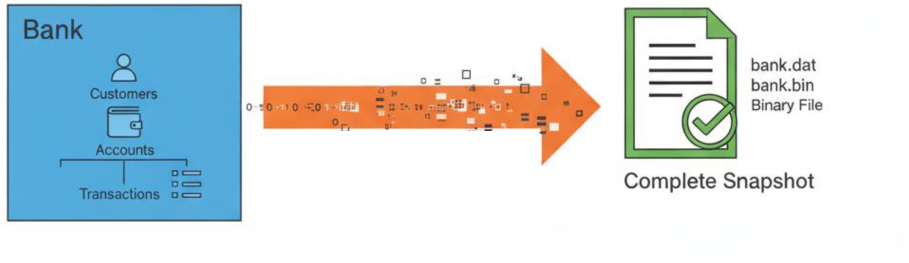


```java
public class BankDataManager {

    public void saveBank(Bank bank, String filename)
         throws IOException {
        // Save bank to file
    }

    public Bank loadBank(String filename)
        throws IOException {
        // Load bank from file
    }
}
```

SAVING BANK STATE

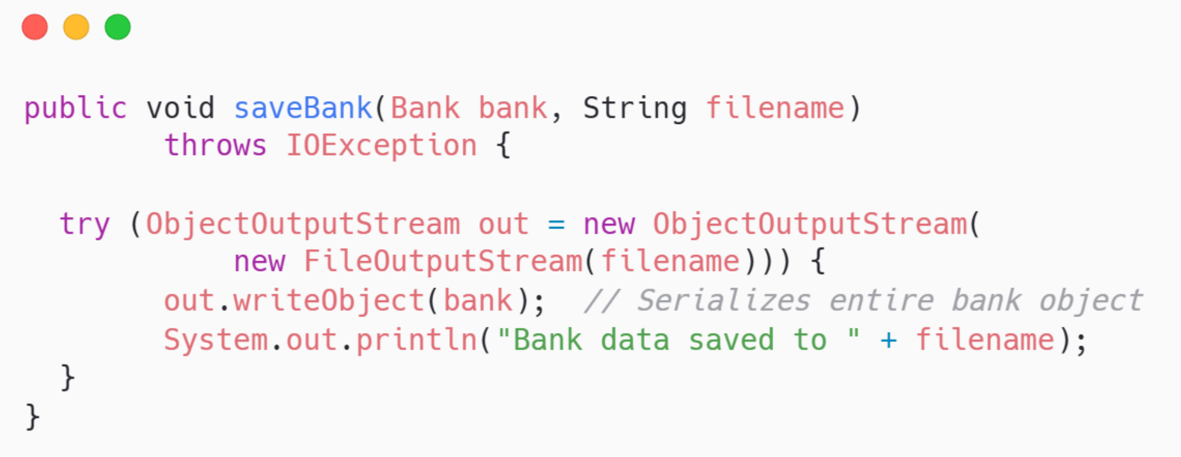
saveBank() writes entire Bank object to file using ObjectOutputStream

- ObjectOutputStream serializes Bank object, all customers and accounts recursively



- Try-with-resources ensures stream properly closed even if exception occurs

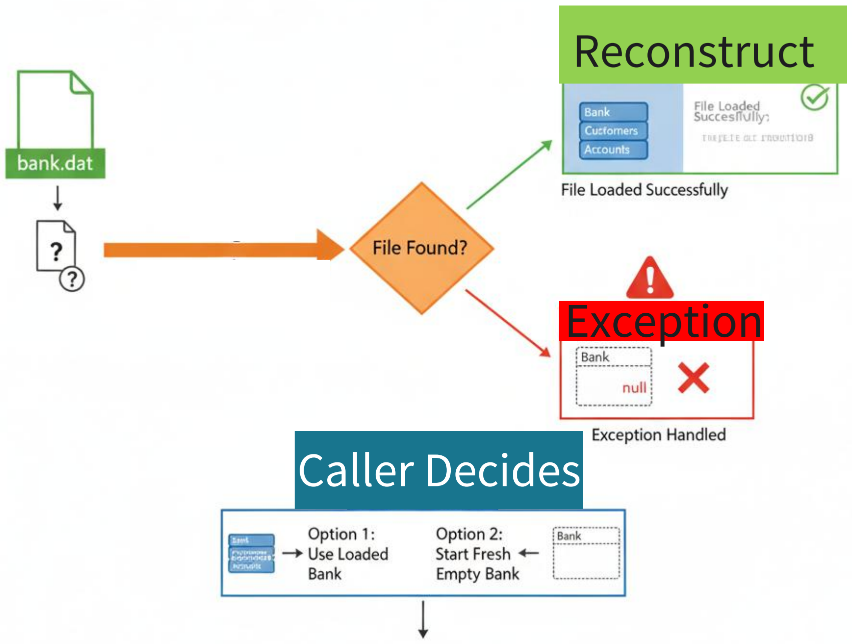
- Writes binary file containing complete system state snapshot



```
public void saveBank(Bank bank, String filename)
        throws IOException {

  try (ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream(filename))) {
        out.writeObject(bank);  // Serializes entire bank object
        System.out.println("Bank data saved to " + filename);
  }
}
```

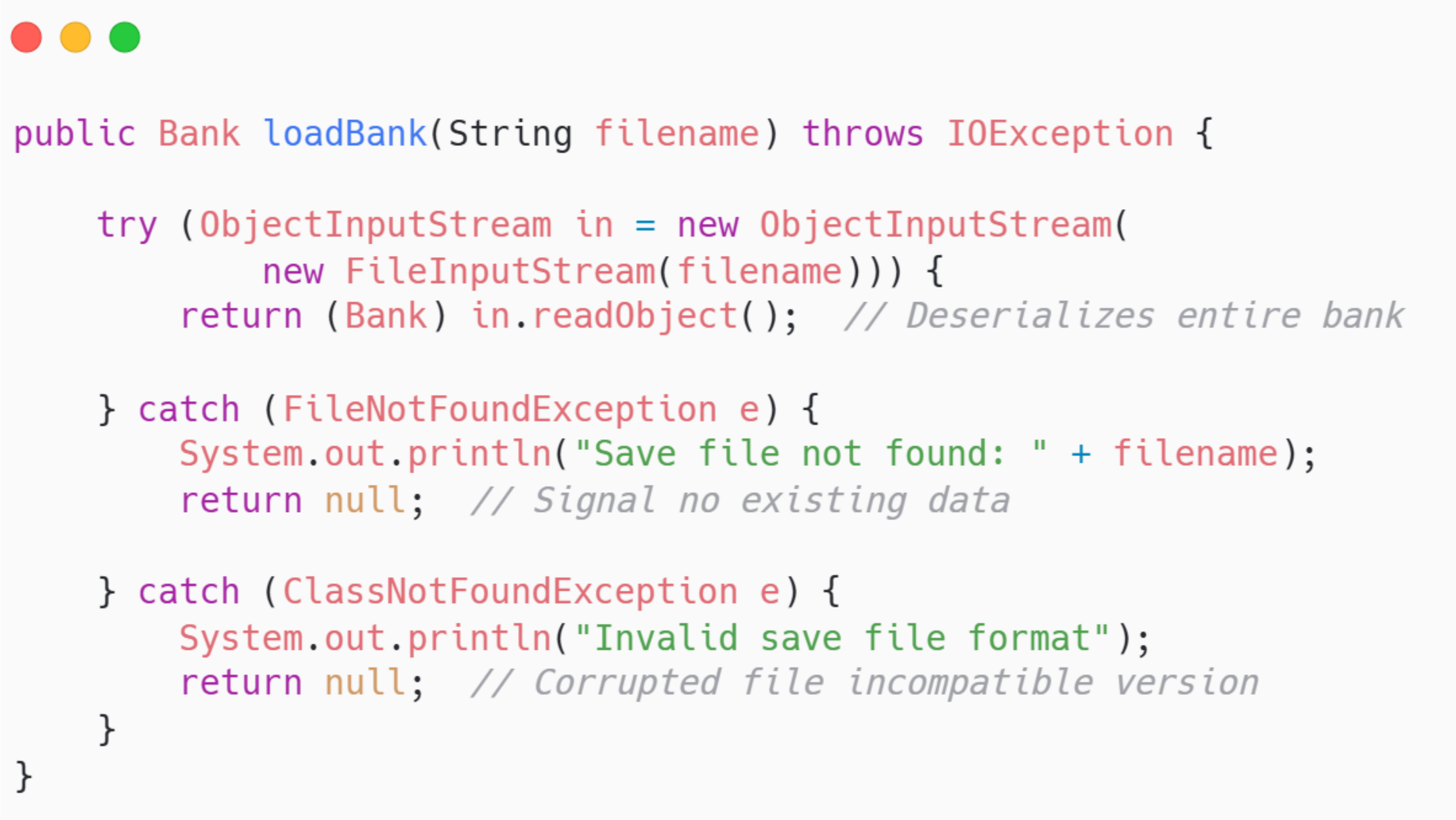
New class centralizes all file I/O for saving and loading complete bank state

- ObjectInputStream reads bytes reconstructs Bank object with all nested data
- Catches FileNotFoundException if save file doesn't exist
- Caller decides whether to use loaded bank or start fresh empty bank



LOADING BANK STATE

New class centralizes all file I/O for saving and loading complete bank state



```java
public Bank loadBank(String filename) throws IOException {

    try (ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(filename))) {
        return (Bank) in.readObject();  // Deserializes entire bank

    } catch (FileNotFoundException e) {
        System.out.println("Save file not found: " + filename);
        return null;  // Signal no existing data

    } catch (ClassNotFoundException e) {
        System.out.println("Invalid save file format");
        return null;  // Corrupted file incompatible version
    }
}
```

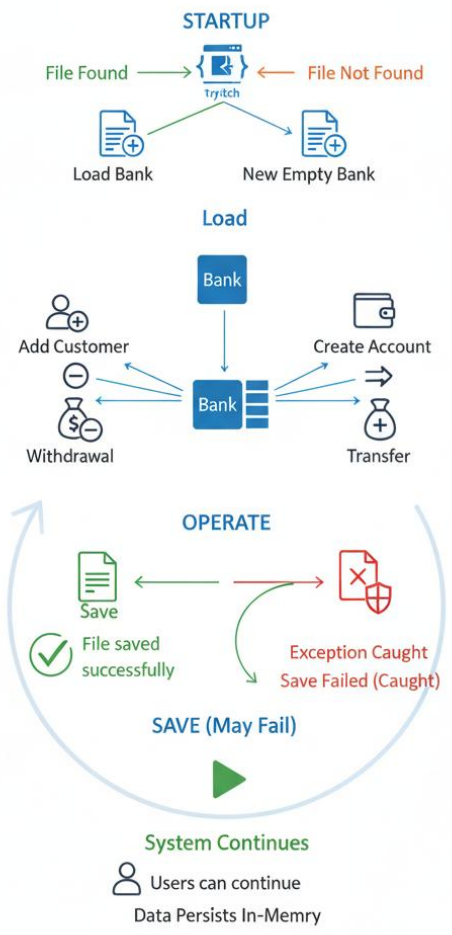
EXCEPTION HANDLING IN MAIN

BankSystemMain wraps save and load operations in try-catch ensuring program survives errors

- Loads existing bank data if available
  Catches FileNotFoundException and uses new bank

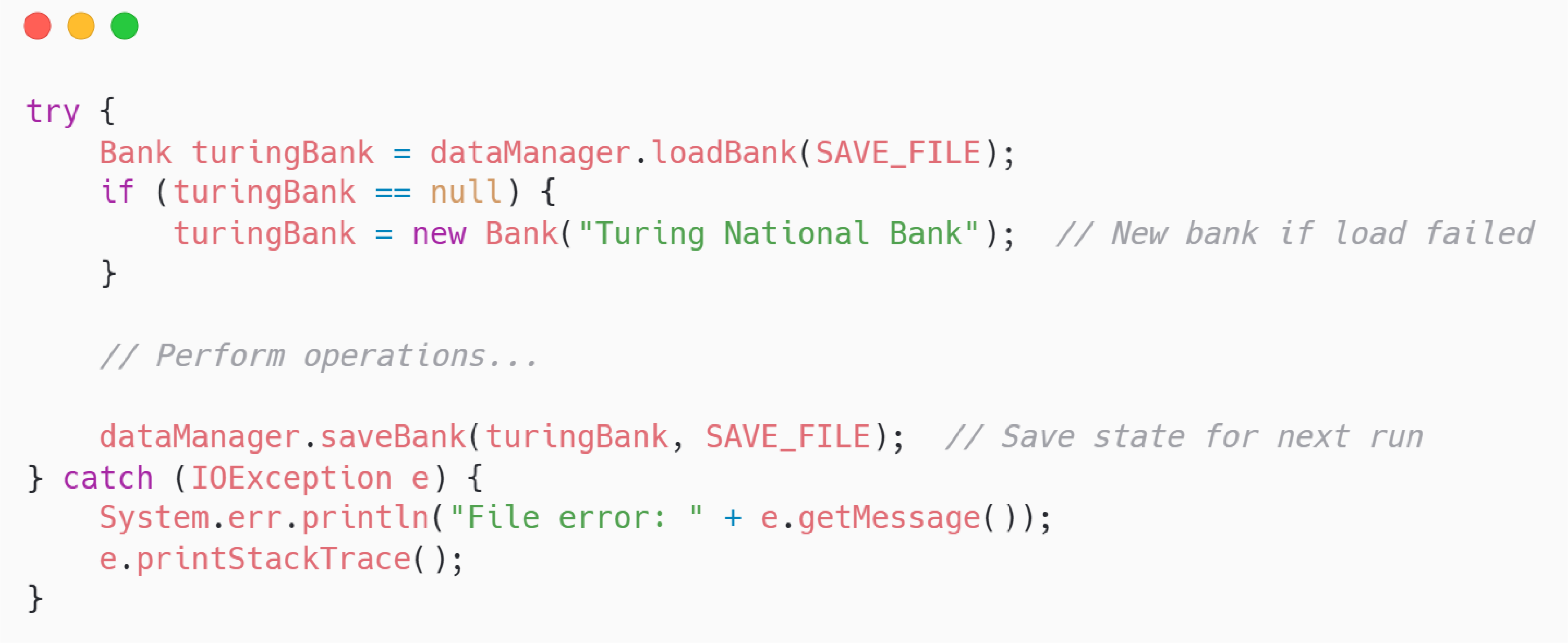
- Performs operations: add customers, create accounts
  withdrawals, deposits, transfers

- Program continues functioning even
  if save fails users can continue in-memory



EXCEPTION HANDLING IN MAIN

BankSystemMain wraps save and load operations in try-catch ensuring program survives errors

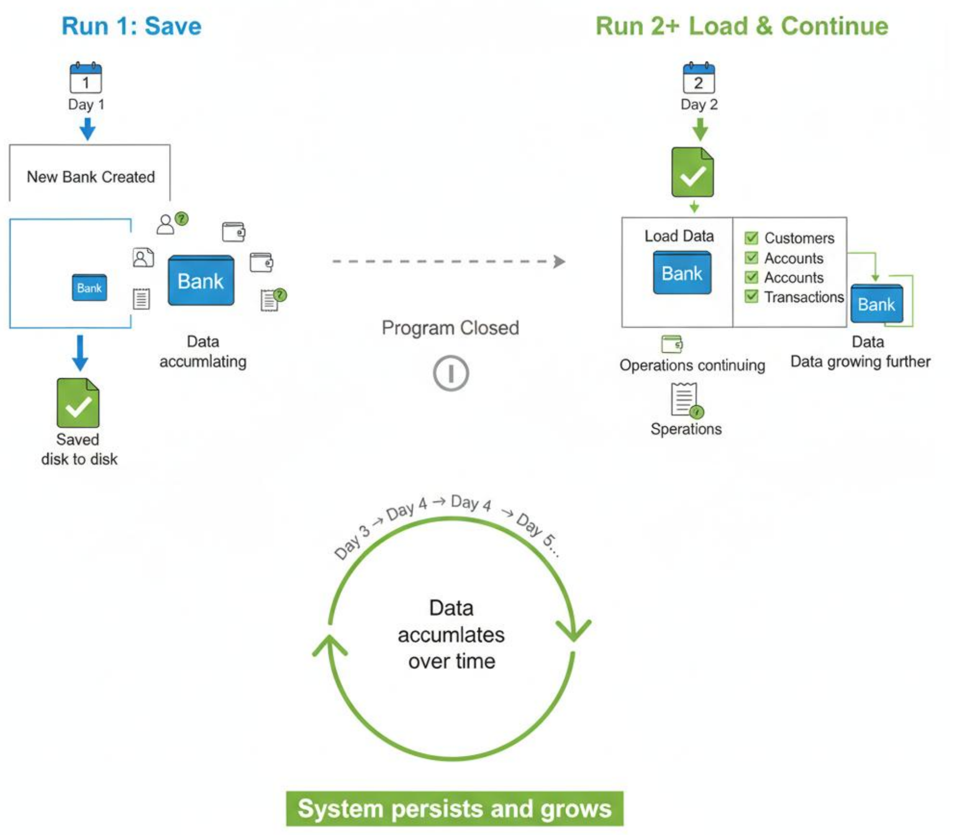


```java
try {
    Bank turingBank = dataManager.loadBank(SAVE_FILE);
    if (turingBank == null) {
        turingBank = new Bank("Turing National Bank");  // New bank if load failed
    }

    // Perform operations...

    dataManager.saveBank(turingBank, SAVE_FILE);  // Save state for next run
} catch (IOException e) {
    System.err.println("File error: " + e.getMessage());
    e.printStackTrace();
}
```

DATA PERSISTENCE ACROSS SCENARIOS

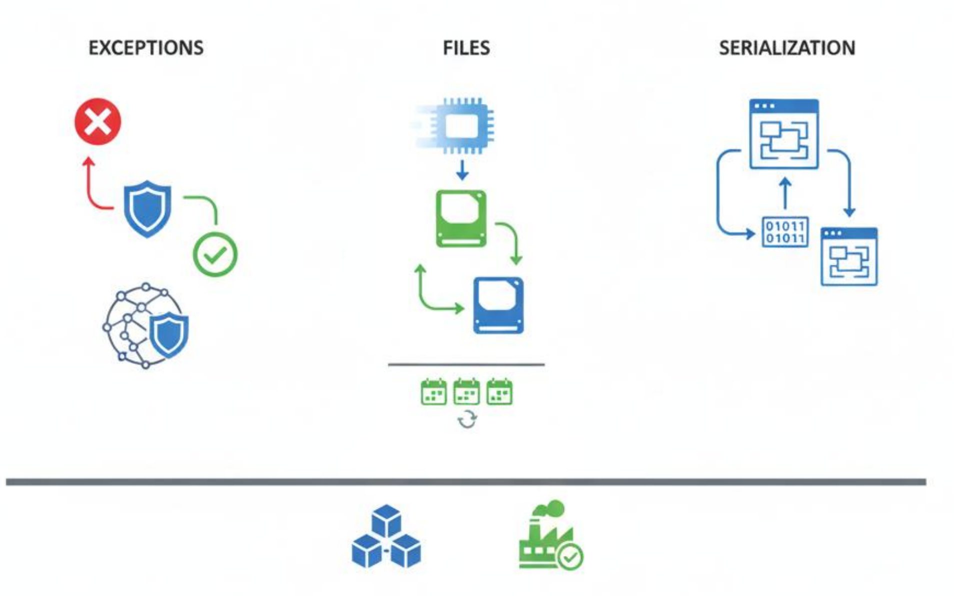
Users expect data saved by program to be restored when program runs again

- First run: no save file exists → new bank created → fresh transactions performed → saved
- Second run: save file found → loaded data restored → customers, accounts, transactions intact



Exceptions and files transform programs from toys into real systems that persist and survive errors

- Exceptions catch errors at source.
- Files persist state beyond program runtime and enable sharing data across runs
- Serialization enables saving complex objects transparently load them restored



QUOTE OF THE WEEK

Errors will find you

Be ready when they do