

# Computer System Design & Application

## 计算机系统设计与应用A

陶伊达 (TAO Yida)

taoyd@sustech.edu.cn

An abstract graphic on the left side of the slide, featuring concentric circles and various digital patterns like binary code and pixelated shapes in shades of blue, green, and white.

# Lecture 6

---

- Persistence and Serialization
- Working with Files
- Exception Handling

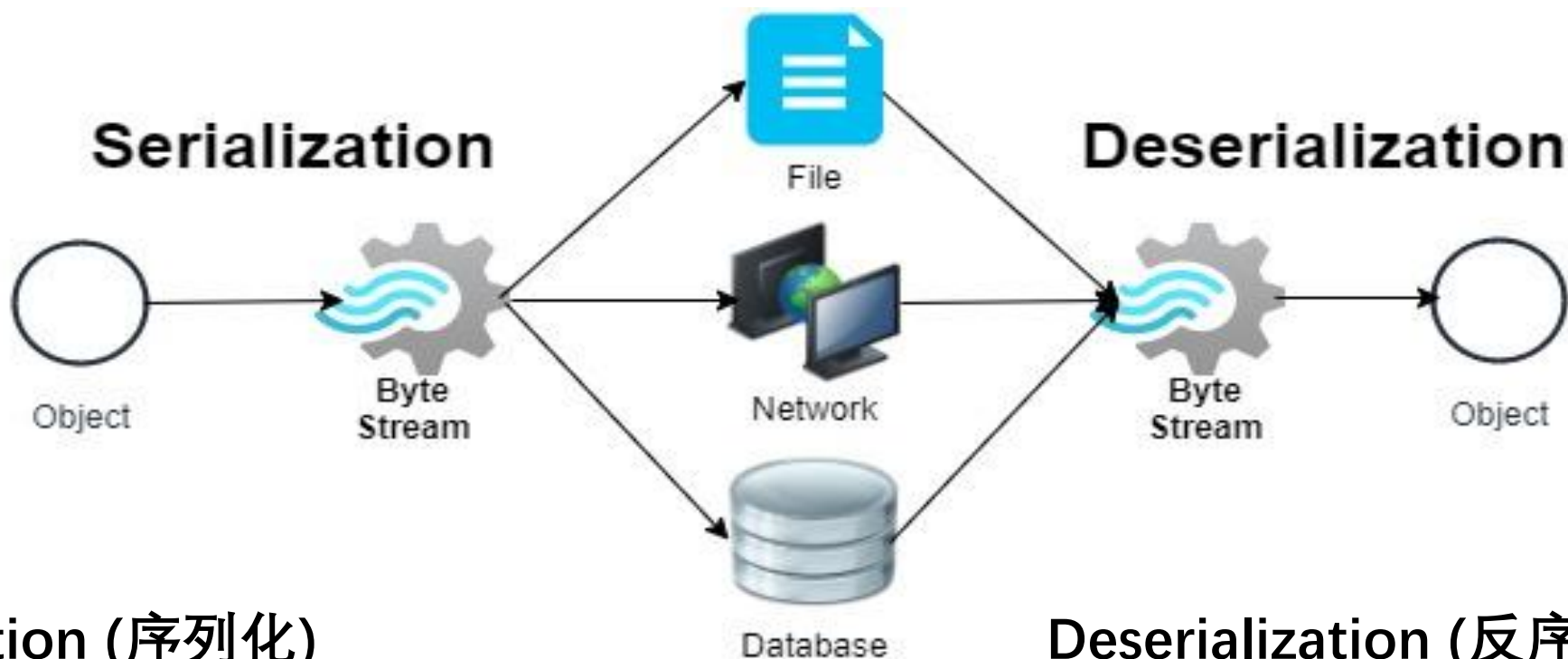
# Data Persistence (数据持久化)

- Objects created in Java programs live in memory; they are removed by the garbage collector once they are not used anymore
- What if we want to persist the objects?

Data survives after the process that created it has ended.  
Reuse the data without having to executing the program all over again to reach that state.

## Data persistence

Store it on a disk, send  
it over the network



### Serialization (序列化)

Converting the state of an object  
into a byte stream

### Deserialization (反序列化)

Using the byte stream to recreate  
the object in the same state

# The Serializable Interface

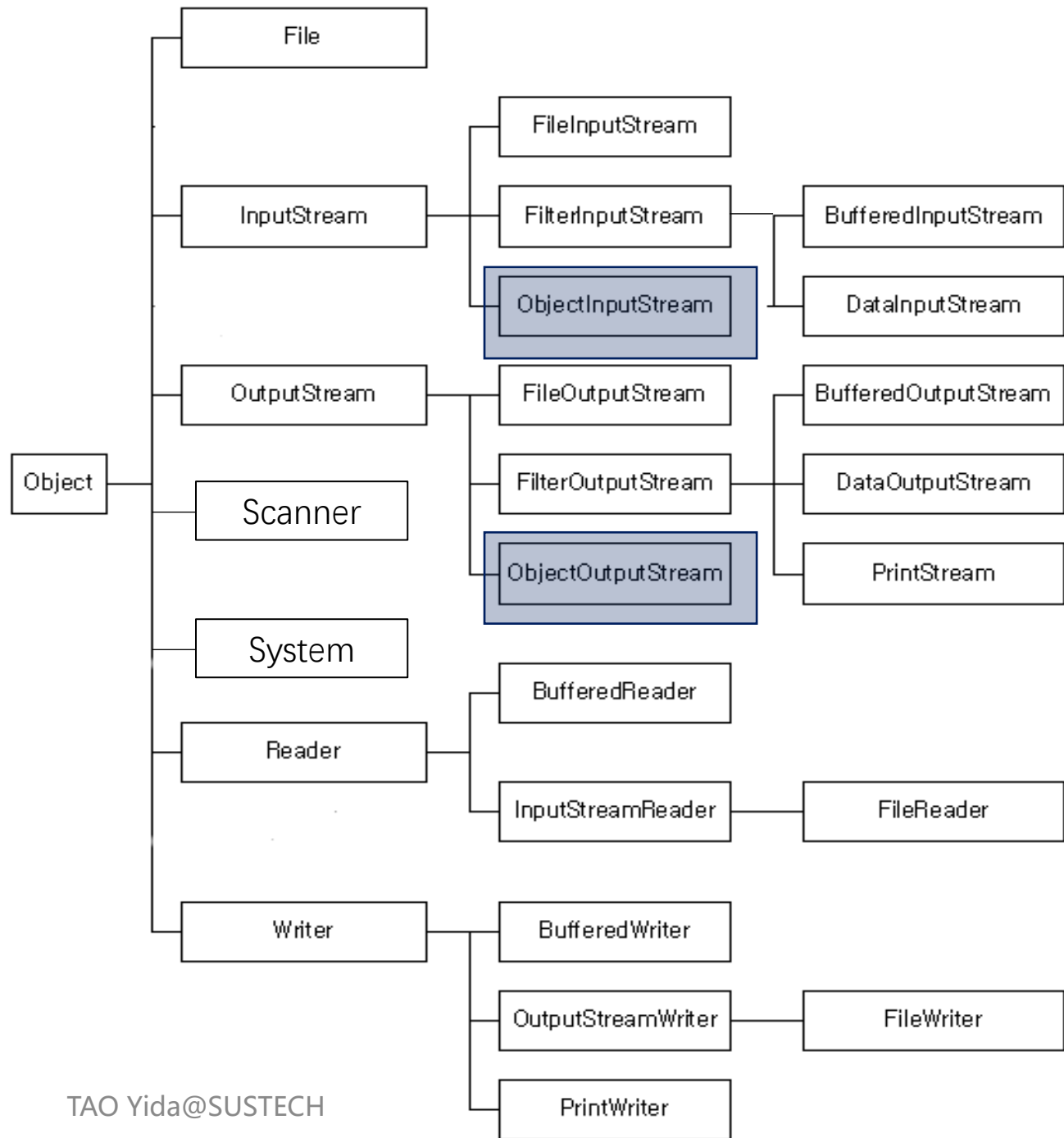
- Classes need to implement the serializable interface for their instances to be serialized or deserialized
- The serializable interface is called a *marker* interface or *tagging* interface (like putting a tag on the class, so the compiler and JVM, when seeing the tag, knows that the object of the class could be serialized)
  - The serializable interface is an empty interface, without any method or field
  - Classes implementing serializable do not have to implement any methods

```
class Student implements Serializable {  
    String name;  
    String dept;  
    int age;  
  
    public String getName() {  
        return name;  
    }  
  
    public String getDept() {  
        return dept;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public Student(String name, String dept, int age) {  
        this.name = name;  
        this.dept = dept;  
        this.age = age;  
    }  
}
```

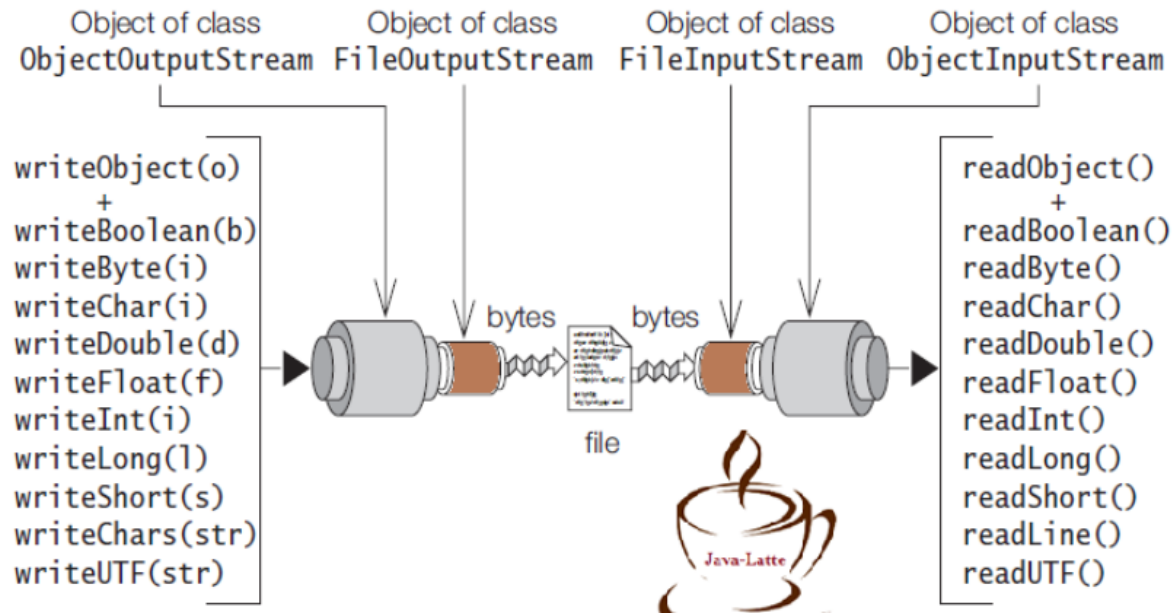
# Using Object Input/Output Streams

`ObjectOutputStream` writes primitive data types and Java objects to an `OutputStream`, using `writeXXX`

`ObjectInputStream` deserializes primitive data and objects previously written using an `ObjectOutputStream`. Using `readXXX`



# Example



<http://java-latte.blogspot.com/2013/11/serialization-in-java.html>

```
Student student = new Student("Alice", "CS", 20);
```

```
// Setup where to store the byte stream
```

```
FileOutputStream fos = new FileOutputStream("student.ser");  
ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```
// serialization
```

```
oos.writeObject(student);
```

```
// Setup where to read the byte stream
```

```
FileInputStream fis = new FileInputStream("student.ser");  
ObjectInputStream ois = new ObjectInputStream(fis);
```

```
// deserialization
```

```
Student student2 = (Student) ois.readObject(); // down-casting object
```

```
System.out.println(student.getName() + " " + student2.getName());
```

```
System.out.println(student.getDept() + " " + student2.getDept());
```

```
System.out.println(student.getAge() + " " + student2.getAge());
```

```
oos.close();
```

```
ois.close();
```



+

•

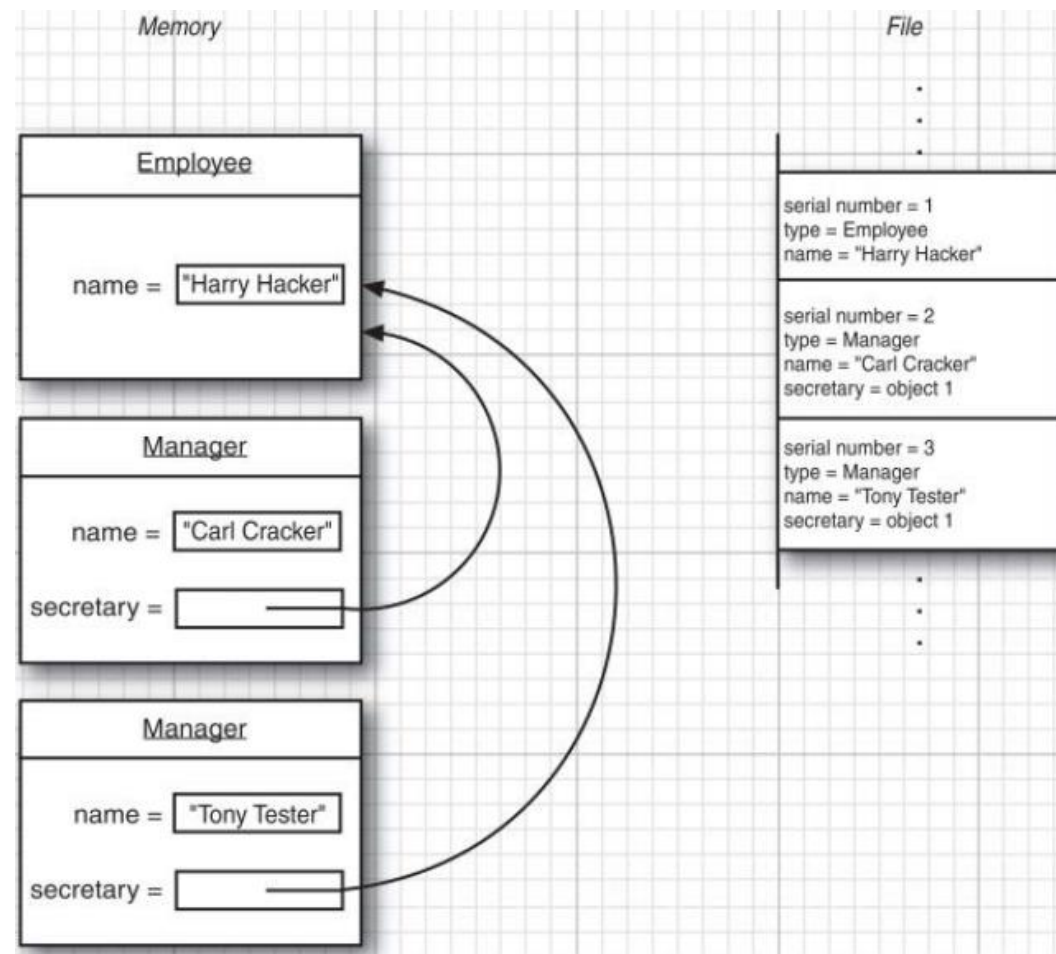
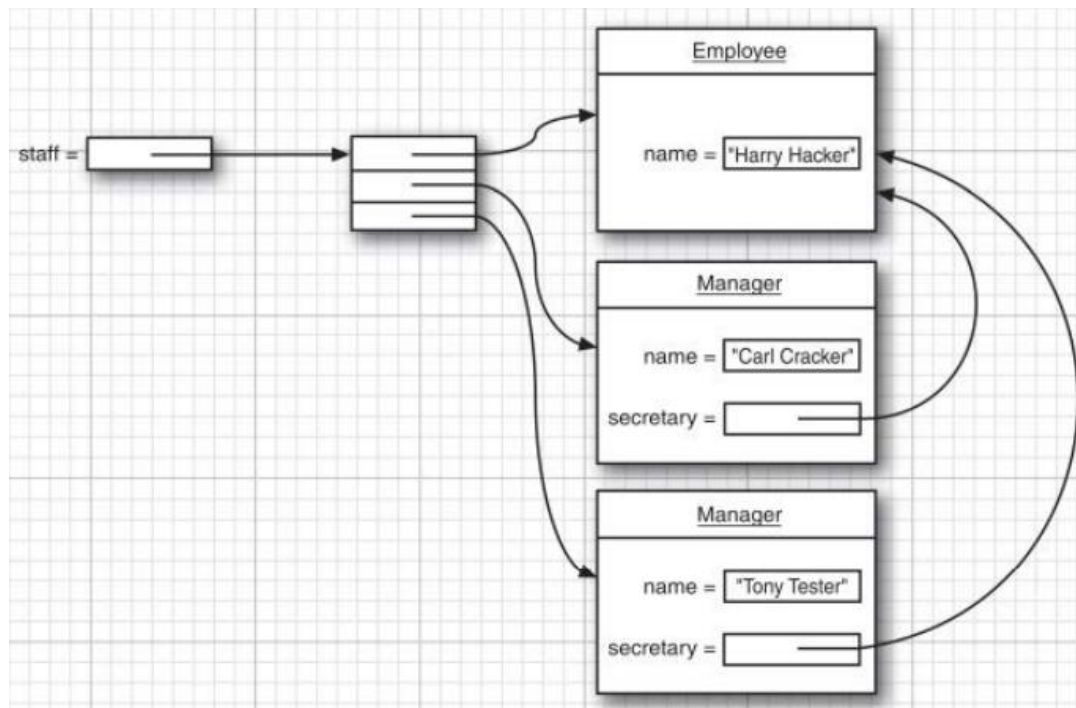
○

# Default Serialization Mechanism

- An ObjectOutputStream looks at all the fields of the objects and save their contents.
- The serialized format contains the types and data fields of all objects.
- Each object is assigned a serial number.
- Repeated occurrences of the same object are stored as references to that serial number.



# Example



Reference: Core Java Volume II, 2.4

# Modifying the Default Serialization Mechanism

- To prevent a field to be serialized, mark it using the transient keyword (e.g., transient int age;)
- If we want to add validation or any other desired action to the default read and write behavior, we can define the following methods in the serializable class

```
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
    throws IOException;
```

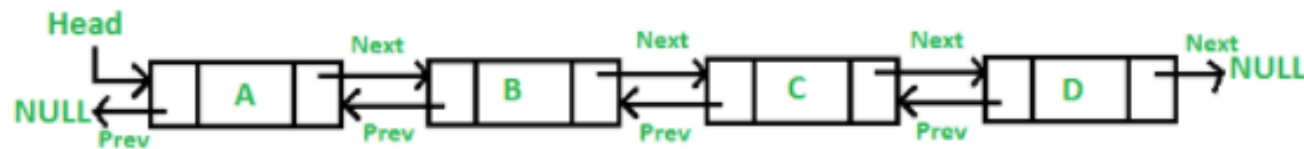
# Example of Customized Serialization

```
// Awful candidate for default serialized form
public final class StringList implements Serializable {
    private int size = 0;
    private Entry head = null;

    private static class Entry implements Serializable {
        String data;
        Entry next;
        Entry previous;
    }

    ... // Remainder omitted
}
```

- The default serialization behavior will serialize every entry and all the links between them in both directions
- Take a long time and consume excessive space



Doubly linked list

Example from Effective Java, Chapter 12

# Example of Customized Serialization

```
// StringList with a reasonable custom serialized form
public final class StringList implements Serializable {
    private transient int size = 0;
    private transient Entry head = null;

    // No longer Serializable!
    private static class Entry {
        String data;
        Entry next;
        Entry previous;
    }

    // Appends the specified string to the list
    public final void add(String s) { ... }

    /**
     * Serialize this {@code StringList} instance.
     *
     * @serialData The size of the list (the number of strings
     * it contains) is emitted ({@code int}), followed by all of
     * its elements (each a {@code String}), in the proper
     * sequence.
     */
    private void writeObject(ObjectOutputStream s)
        throws IOException {
        s.defaultWriteObject();
        s.writeInt(size);

        // Write out all elements in the proper order.
        for (Entry e = head; e != null; e = e.next)
            s.writeObject(e.data);
    }
}
```

“Serialize an object’s logic data rather than its physical implementation”

- We only care about the size of the StringList and the data of each entry
- Implement customized writeObject() inside the class to be serialized to replace the default behavior
  - defaultWriteObject(): a special method of the ObjectOutputStream class that can only be called from within a writeObject method of a serializable class. This method writes the object descriptor (e.g., fingerprint and the number of fields) and all serializable fields.
  - writeInt()
  - writeObject()
  - ...

# Example of Customized Deserialization

- Suppose we serialized a `Period` class, which describing valid time ranges, to a byte stream

```
// Byte stream couldn't have come from a real Period instance!
private static final byte[] serializedForm = {
    (byte)0xac, (byte)0xed, 0x00, 0x05, 0x73, 0x72, 0x00, 0x06,
    0x50, 0x65, 0x72, 0x69, 0x6f, 0x64, 0x40, 0x7e, (byte)0xf8,
    0x2b, 0x4f, 0x46, (byte)0xc0, (byte)0xf4, 0x02, 0x00, 0x02,
    0x4c, 0x00, 0x03, 0x65, 0x6e, 0x64, 0x74, 0x00, 0x10, 0x4c,
    0x6a, 0x61, 0x76, 0x61, 0x2f, 0x75, 0x74, 0x69, 0x6c, 0x2f,
    0x44, 0x61, 0x74, 0x65, 0x3b, 0x4c, 0x00, 0x05, 0x73, 0x74,
    0x61, 0x72, 0x74, 0x71, 0x00, 0x7e, 0x00, 0x01, 0x78, 0x70,
    0x73, 0x72, 0x00, 0x0e, 0x6a, 0x61, 0x76, 0x61, 0x2e, 0x75,
    0x74, 0x69, 0x6c, 0x2e, 0x44, 0x61, 0x74, 0x65, 0x68, 0x6a,
    (byte)0x81, 0x01, 0x4b, 0x59, 0x74, 0x19, 0x03, 0x00, 0x00,
    0x78, 0x70, 0x77, 0x08, 0x00, 0x00, 0x00, 0x66, (byte)0xdf,
    0x6e, 0x1e, 0x00, 0x78, 0x73, 0x71, 0x00, 0x7e, 0x00, 0x03,
    0x77, 0x08, 0x00, 0x00, 0x00, (byte)0xd5, 0x17, 0x69, 0x22,
    0x00, 0x78
};
```

```
private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();

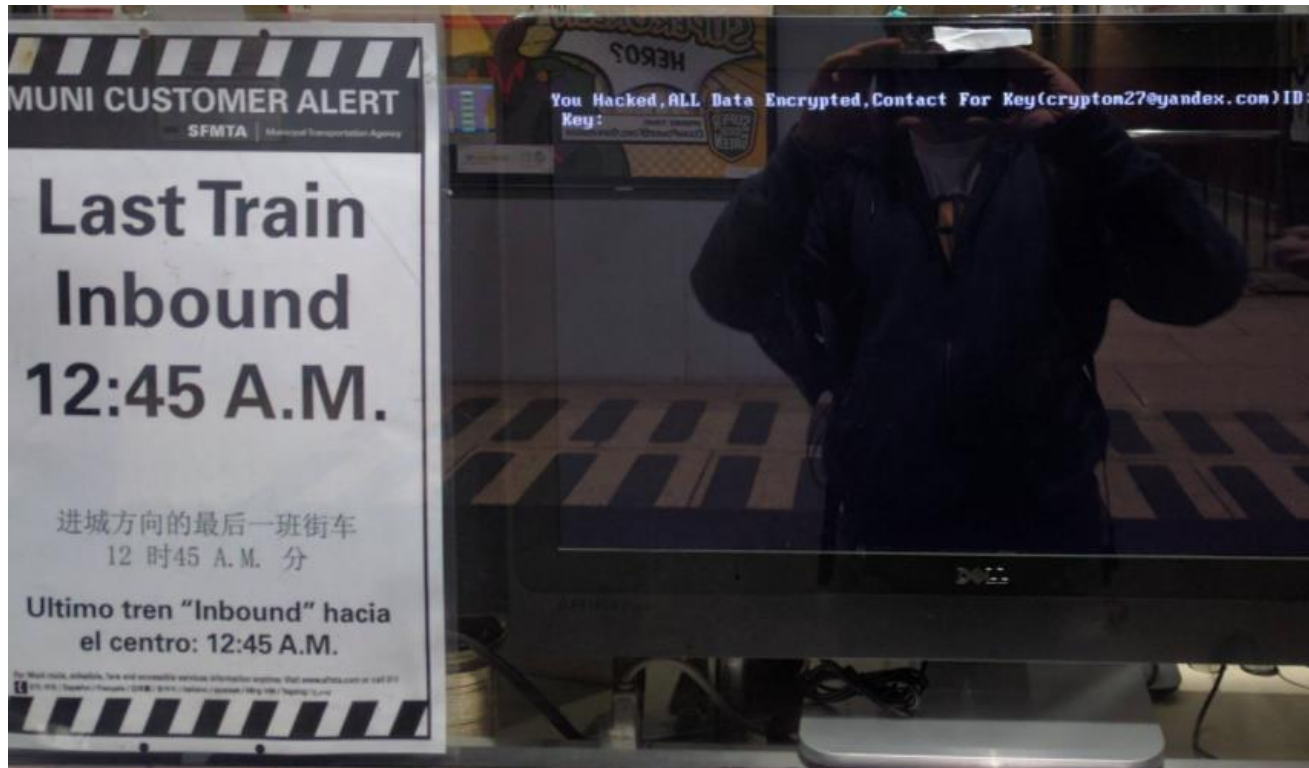
    // Check that our invariants are satisfied
    if (start.compareTo(end) > 0)
        throw new InvalidObjectException(start + " after " + end);
}
```

A bad guy modifies the byte stream; So after we deserialize it, we'll get an invalid time period (e.g., Fri. Jan 1 2021 to Sun Jan. 1 2021)

Example from Effective Java, Chapter 12

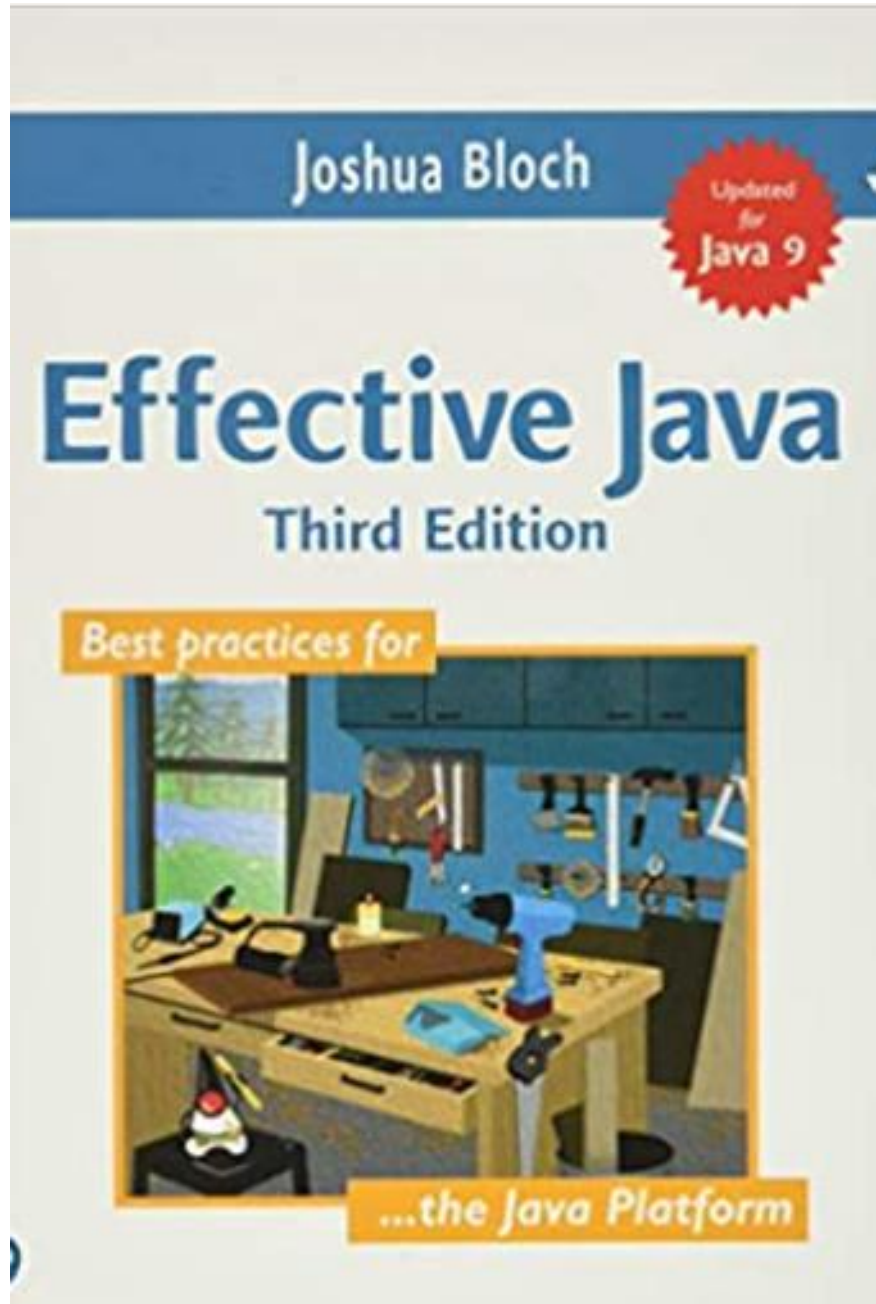


# Attacks exploiting de/serialization



- Attackers could submit a carefully crafted byte stream for the target to deserialize, enable attackers to execute arbitrary code on the target machine (SFMTA Muni Attack)

# Serialization



**T**HIS chapter concerns *object serialization*, which is Java's framework for encoding objects as byte streams (*serializing*) and reconstructing objects from their encodings (*deserializing*). Once an object has been serialized, its encoding can be sent from one VM to another or stored on disk for later deserialization. This chapter focuses on the dangers of serialization and how to minimize them.

## Further Reading

---



An abstract graphic on the left side of the slide, featuring concentric circles and digital patterns in shades of blue, green, and white, resembling a stylized data visualization or a futuristic interface.

# Lecture 6

---

- Persistence and Serialization
- Working with Files
- Exception Handling

# Working with Files

- We have learned how to read and write data from a file, yet there is more to file management than reading & writing
- The Path interface, Paths class, and Files class, introduced in Java 7, are much more convenient than the File class dated back all the way to JDK 1.0



# Path

```
import java.nio.file.Path;  
import java.nio.file.Paths;
```

- A Path instance contains the information used to specify the location of a file or directory
- You can easily create a Path object by using one of the following get methods from the Paths helper class

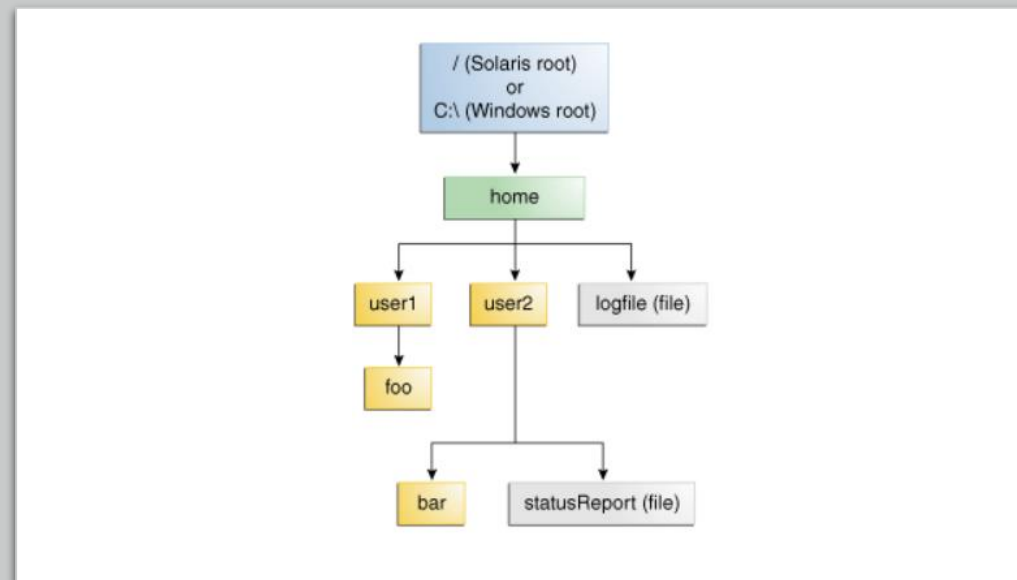
```
Path p1 = Paths.get("resources");  
  
Path p2 = Paths.get(args[0]);  
  
Path p3 = Paths.get(URI.create("file:///D:/CS209A/sample.txt"));
```

<https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html>

# Path

- Path stores these name elements as a sequence.
  - The highest element in the directory structure would be located at index 0.
  - The lowest element in the directory structure would be located at index  $[n-1]$ , where  $n$  is the number of name elements in the Path.
- A path that starts with root is *absolute*; otherwise, it is *relative*

<https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html>



Method Invoked	Returns in the Solaris OS	Returns in Microsoft Windows
toString	/home/joe/foo	C:\home\joe\foo
getFileName	foo	foo
getName(0)	home	home
getNameCount	3	3
subpath(0,2)	home/joe	home\joe
getParent	/home/joe	\home\joe
getRoot	/	C:\

# Path

// Microsoft Windows syntax

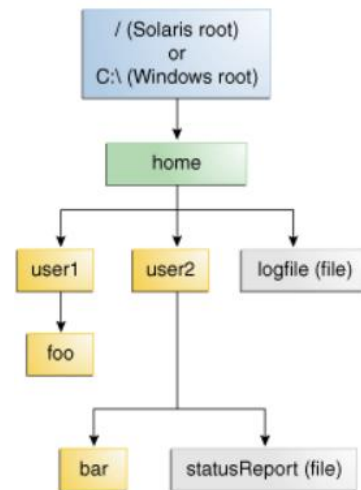
```
Path path = Paths.get("C:\\home\\joe\\foo");
```

// Solaris syntax

```
Path path = Paths.get("/home/joe/foo");
```

```
System.out.format("toString: %s\n", path.toString());
System.out.format("getFileName: %s\n", path.getFileName());
System.out.format("getName(0): %s\n", path.getName(0));
System.out.format("getNameCount: %d\n", path.getNameCount());
System.out.format("subpath(0,2): %s\n", path.subpath(0,2));
System.out.format("getParent: %s\n", path.getParent());
System.out.format("getRoot: %s\n", path.getRoot());
```

<https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html>



Method Invoked	Returns in the Solaris OS	Returns in Microsoft Windows
toString	/home/joe/foo	C:\home\joe\foo
getFileName	foo	foo
getName(0)	home	home
getNameCount	3	3
subpath(0,2)	home/joe	home\joe
getParent	/home/joe	home\joe
getRoot	/	C:\



# Dot notations

- When working with relative paths, you may use two special notations inside the path string:
  - . (current directory)
  - .. (parent directory)
- You may use `normalize()` method to remove redundancies from a path

```
Path rp1 = Paths.get("C:\\Users\\admin\\CS209A_Lectures\\.");  
Path rp2 = Paths.get("C:\\Users\\admin\\test\\..\\CS209A_Lectures");  
  
System.out.format("rp1 normalize: %s%n", rp1.normalize());  
System.out.format("rp2 normalize: %s%n", rp2.normalize());
```

Both normalizes to "C:\\Users\\admin\\CS209A\_Lectures"

# Converting a Path

```
Path cp = Paths.get("resources\\..\\resources\\math.txt");  
// C:\Users\admin\CS209A_Lectures\resources\..\resources\math.txt  
System.out.println(cp.toAbsolutePath());  
// C:\Users\admin\CS209A_Lectures\resources\math.txt  
System.out.println(cp.toRealPath());
```

```
Path cp2 = Paths.get("resources\\..\\resources\\notexist.txt");  
// C:\Users\admin\CS209A_Lectures\resources\..\resources\notexist.txt  
System.out.println(cp2.toAbsolutePath());  
// Throws NoSuchFileException  
System.out.println(cp2.toRealPath());
```



# Converting a Path

## `toAbsolutePath()`

- Converts a path to an absolute path. If the passed-in path is already absolute, it returns the same Path object.
- The file does not need to exist for this method to work.

## `toRealPath()`

- If the Path is relative, it returns an absolute path.
- If the Path contains any redundant elements, it returns a path with those elements removed.
- Throws an exception if the file does not exist or cannot be accessed.

# Files

This class, `java.nio.file.Files`, consists exclusively of static methods that operate on files, directories, or other types of files.

## Creating Files and Directories

`java.nio.file.Files` 7

- `static Path createFile(Path path, FileAttribute<?>... attrs)`
- `static Path createDirectory(Path path, FileAttribute<?>... attrs)`
- `static Path createDirectories(Path path, FileAttribute<?>... attrs)`  
creates a file or directory. The `createDirectories` method creates any intermediate directories as well.
- `static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs)`
- `static Path createTempFile(Path parentDir, String prefix, String suffix, FileAttribute<?>... attrs)`
- `static Path createTempDirectory(String prefix, FileAttribute<?>... attrs)`
- `static Path createTempDirectory(Path parentDir, String prefix, FileAttribute<?>... attrs)`  
creates a temporary file or directory, in a location suitable for temporary files or in the given parent directory. Returns the path to the created file or directory.

Core Java, Volume II, Chapter 2

# Files

This class, `java.nio.file.Files`, consists exclusively of static methods that operate on files, directories, or other types of files.

## Copying, Moving, and Deleting Files

`java.nio.file.Files` 7

- static `Path copy(Path from, Path to, CopyOption... options)`
- static `Path move(Path from, Path to, CopyOption... options)`  
copies or moves `from` to the given target location and returns `to`.
- static `long copy(InputStream from, Path to, CopyOption... options)`
- static `long copy(Path from, OutputStream to, CopyOption... options)`  
copies from an input stream to a file, or from a file to an output stream, returning the number of bytes copied.
- static `void delete(Path path)`
- static `boolean deleteIfExists(Path path)`  
deletes the given file or empty directory. The first method throws an exception if the file or directory doesn't exist. The second method returns `false` in that case.

Core Java, Volume II, Chapter 2

# Files

This class, `java.nio.file.Files`, consists exclusively of static methods that operate on files, directories, or other types of files.

## Getting File Information

### `java.nio.file.Files` 7

- `static boolean exists(Path path)`
- `static boolean isHidden(Path path)`
- `static boolean isReadable(Path path)`
- `static boolean isWritable(Path path)`
- `static boolean isExecutable(Path path)`
- `static boolean isRegularFile(Path path)`
- `static boolean isDirectory(Path path)`
- `static boolean isSymbolicLink(Path path)`  
checks for the given property of the file given by the path.
- `static long size(Path path)`  
gets the size of the file in bytes.
- `A readAttributes(Path path, Class<A> type, LinkOption... options)`  
reads the file attributes of type A.

### `java.nio.file.attribute.BasicFileAttributes` 7

- `FileTime creationTime()`
- `FileTime lastAccessTime()`
- `FileTime lastModifiedTime()`
- `boolean isRegularFile()`
- `boolean isDirectory()`
- `boolean isSymbolicLink()`
- `long size()`
- `Object fileKey()`  
gets the requested attribute.

Core Java, Volume II, Chapter 2

# Visiting Directory Entries

```
try(Stream<Path> entries = Files.list(dir2)){  
    entries.forEach(p -> System.out.println(p.toAbsolutePath()));  
}
```

**Files.list**

Does not enter subdirectories

```
try(Stream<Path> entries = Files.walk(dir2)){  
    entries.filter(Files::isRegularFile).forEach(System.out::println);  
}
```

**Files.walk**

Enter all subdirectories in a depth-first manner

```
\---folder  
|   file1.txt  
|   file2.txt  
|  
\---subfolder  
    file3.txt  
    file4.txt
```

# Visiting Directory Entries

- You may also use `walkFileTree` method and supply an object of type `SimpleFileVisitor`, which gets notified
  - When a file is encountered (`visitFile`)
  - Before a directory is processed (`preVisitDirectory`)
  - After a directory is processed (`postVisitDirectory`)
  - When an error occurred (`visitFileFailed`)
- You may perform any actions you want for these events, and specify whether you want to
  - Continue visiting the next file (`FileVisitResult.CONTINUE`)
  - Continue but without visiting the entries in this directory (`FileVisitResult.SKIP_SUBTREE`)
  - Continue but without visiting the siblings of this file (`FileVisitResult.SKIP_SIBLINGS`)
  - Terminate the walk (`FileVisitResult.TERMINATE`)

# Visiting Directory Entries

```
Path path = Paths.get(".");  
Files.walkFileTree(path, new ListFileVisitor());
```

```
class ListFileVisitor extends SimpleFileVisitor<Path> {  
  
    @Override  
    public FileVisitResult visitFile(Path file, BasicFileAttributes attributes) throws IOException {  
        System.out.println("Visiting file:" + file.toRealPath());  
        return FileVisitResult.CONTINUE;  
    }  
  
    @Override  
    public FileVisitResult postVisitDirectory(Path directory, IOException e)  
        throws IOException {  
        System.out.println("Finished directory: "  
            + directory.toRealPath());  
        return FileVisitResult.CONTINUE;  
    }  
  
    @Override  
    public FileVisitResult preVisitDirectory(Path directory,  
        BasicFileAttributes attributes) throws IOException {  
        System.out.println("Start directory: "  
            + directory.toRealPath());  
        return FileVisitResult.CONTINUE;  
    }  
  
    @Override  
    public FileVisitResult visitFileFailed(Path file, IOException exc)  
        throws IOException {  
        System.out.println("An error occurred.");  
        return FileVisitResult.SKIP_SUBTREE;  
    }  
}
```



# Reading Files Line by Line

```
Path file = Paths.get("resources","math.txt");
```

```
System.out.println("Using Scanner:");  
Scanner in = new Scanner(file);  
while(in.hasNext()){  
    System.out.println(in.nextLine());  
}
```

```
System.out.println("Using Files.lines:");  
try (Stream<String> stream = Files.lines(file)) {  
    stream.forEach(System.out::println);  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

```
System.out.println("Using BufferedReader:");  
try (BufferedReader br = Files.newBufferedReader(file)) {  
    String line;  
    while ((line = br.readLine()) != null) {  
        System.out.println(line);  
    }  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

An abstract graphic on the left side of the slide, featuring concentric circles and various digital patterns like binary code and pixelated shapes in shades of blue, green, and white.

# Lecture 6

---

- Persistence and Serialization
- Working with Files
- Exception Handling

# Exception

- An exception indicates that a problem occurs during a program's execution
- An exception disrupts the normal flow of the program

## Happy Path

Files are always there  
Network is always okay  
Memory is always enough  
User input is always valid  
.....



## Unhappy Path

Files are not found  
Network breaks down  
Memory is not enough  
User input is invalid  
.....



# Exception Handling

- A mechanism to handle runtime errors (gracefully) in order to maintain the normal flow of the program

## Handling

Passing control from the point of error detection to a handler that can deal with the error

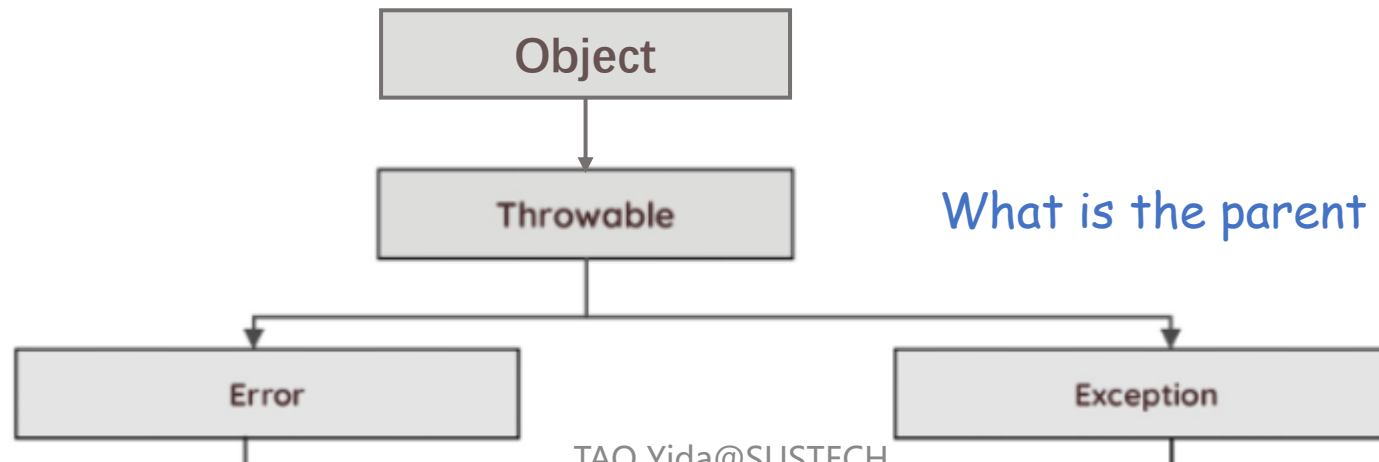
```
try {  
    File text = new File("C:/temp/test.txt");  
    Scanner s = new Scanner(text);  
} catch (FileNotFoundException e) {  
    System.err.println("file not found.");  
}
```

## Detection

Scanner class could detect the error, but cannot handle it

# Java Exception Hierarchy

- The `Throwable` class is at the top of the Java exception class hierarchy; has two direct subclass: `Error` and `Exception`
- Only `Throwable` or its subclasses
  - Can be thrown by JVM or the `throw` keyword
  - Can be caught by the `catch` keyword



What is the parent class of Throwable?

# Error

- The Error hierarchy describes internal errors and resource exhaustion situations inside the Java runtime system.
- An error indicates serious problems that a reasonable application **should not** try to catch
- E.g., OutOfMemoryError, StackOverflowError

Mostly thrown by JVM in a scenario considered fatal; no way for the application program to recover from that error

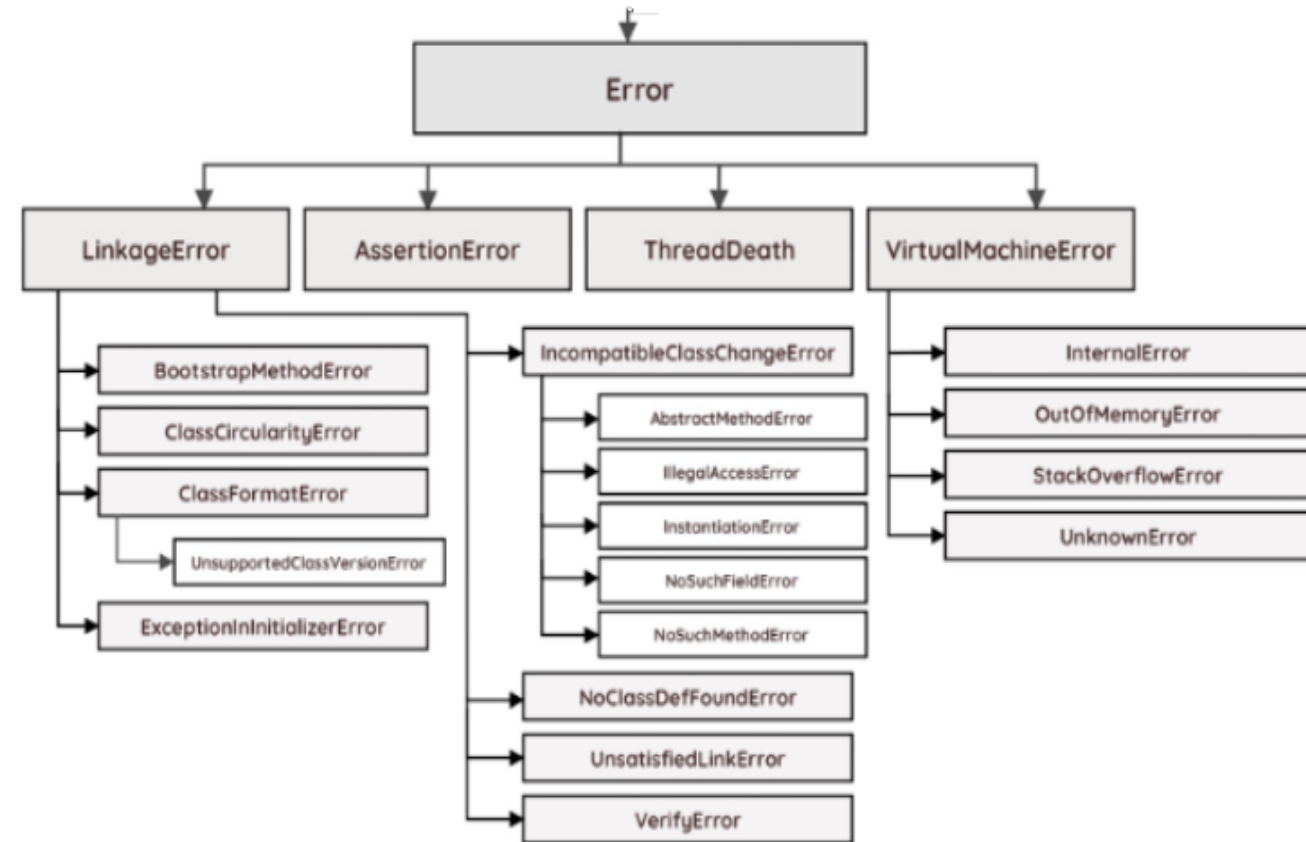


Image source: <https://rollbar.com/blog/java-exceptions-hierarchy-explained>

# Example

- What is the problem with `foo(String s)`?
- Stack is exhausted, leading to **StackOverflowError**
- No recovery during execution, just let it terminate
- Fixing the code or increasing JVM stack size (-Xss)

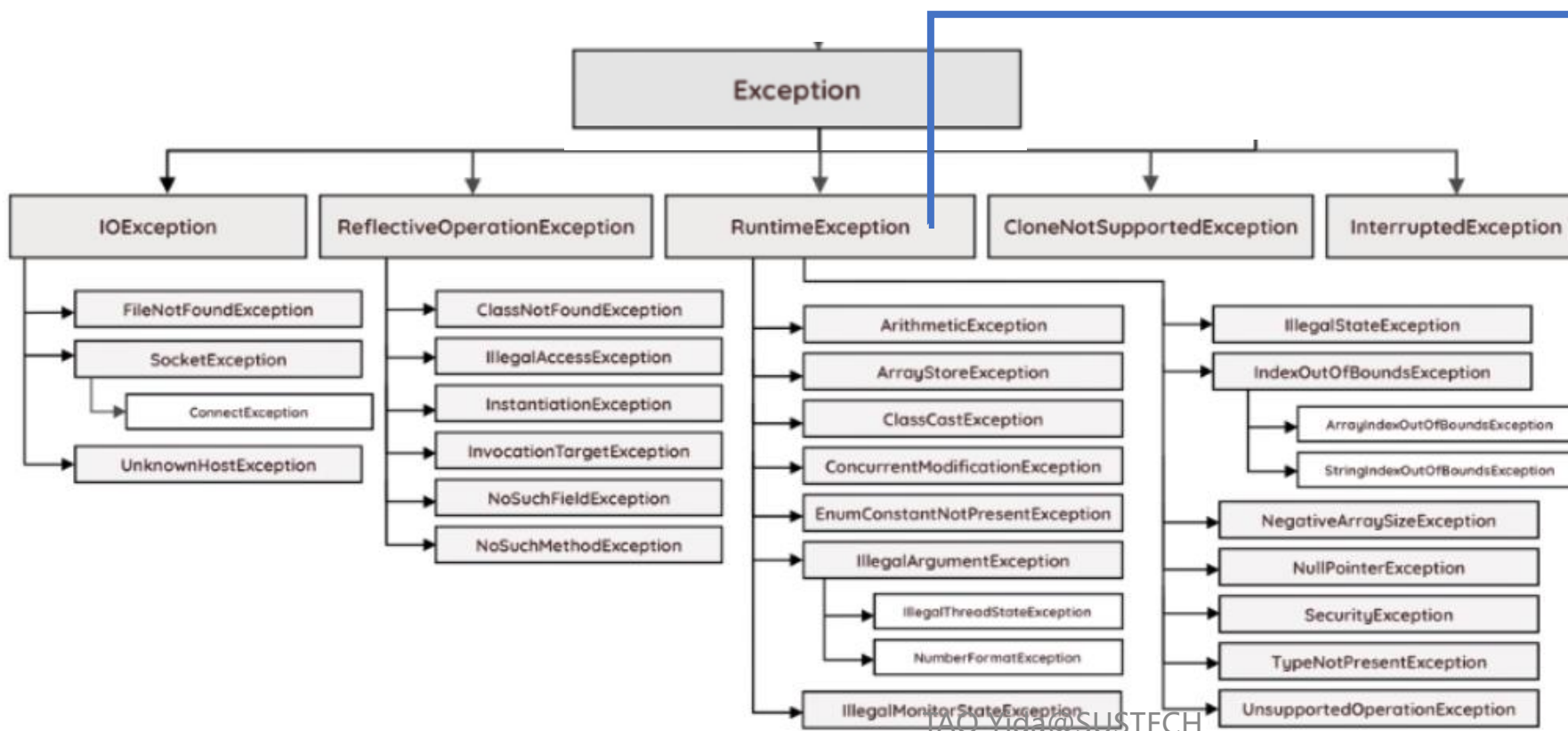
```
public void foo(String s)
{
    foo(s);
}
```

```
Exception in thread "main" java.lang.StackOverflowError
    at examples.foo(examples.java:58)
    at examples.foo(examples.java:58)
    at examples.foo(examples.java:58)
    at examples.foo(examples.java:58)|
    at examples.foo(examples.java:58)
    at examples.foo(examples.java:58)
    at examples.foo(examples.java:58)
```



# Exception

- An exception indicates a condition that a reasonable application might **want to** catch.



- RuntimeException and its subclasses are **unchecked exceptions**
- Others are **checked exceptions** (think of it as “checked” by compiler)

# Checked Exceptions

- Checked Exceptions cannot be ignored at the time of compilation
- Compilers will enforce programmers to handle them
- Two fixes: throw or catch

```
public void processFile() {  
    File text = new File("C:/test.txt");  
    Scanner s = new Scanner(text);  
}
```

Unhandled exception type FileNotFoundException

2 quick fixes available:

- [Add throws declaration](#)
- [Surround with try/catch](#)

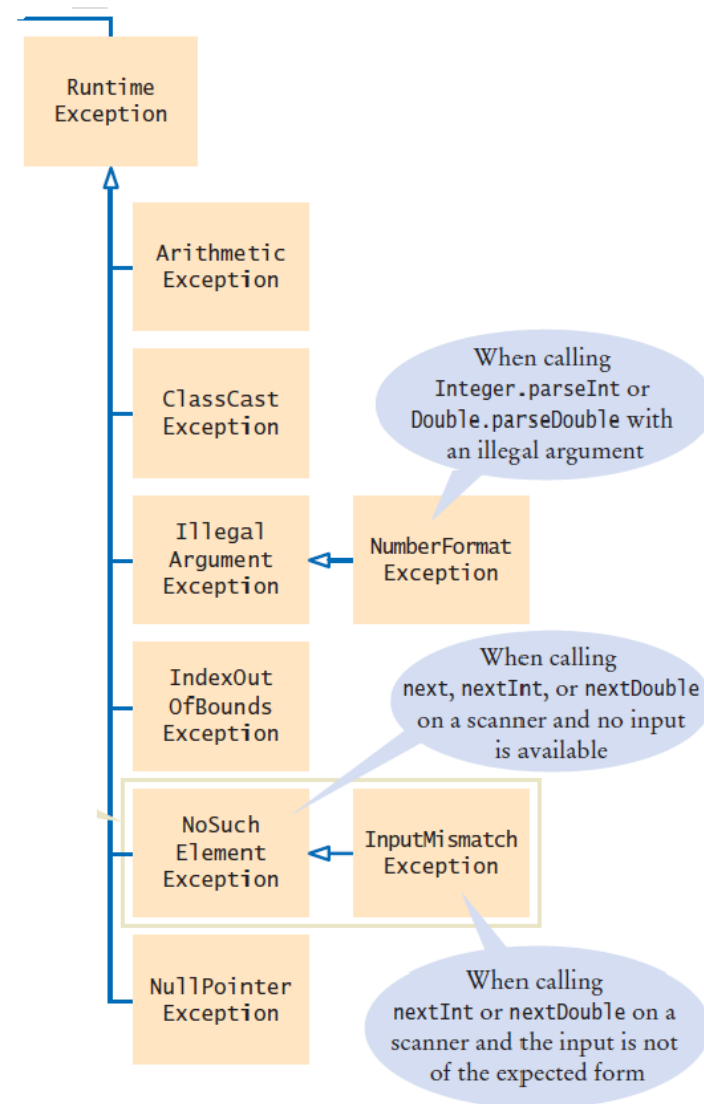
Press 'F2' for focus

```
public void processFile() throws FileNotFoundException {  
    File text = new File("C:/test.txt");  
    Scanner s = new Scanner(text);  
}
```

```
public void processFile() {  
    File text = new File("C:/test.txt");  
    try {  
        Scanner s = new Scanner(text);  
    } catch (FileNotFoundException e) {  
        System.out.println("Cannot find file xxx.");  
    }  
}
```

# Unchecked Exceptions


- Will not be checked by compilers; Occur at runtime
- Usually caused by logic errors in programming
- E.g., NullPointerException, IndexOutOfBoundsException



# Catching Multiple Exceptions

```
try {  
    // some code  
} catch(FileNotFoundException e) {  
    logger.log(e);  
}  
catch(SQLException e) {  
    logger.log(e);  
}  
catch(SocketException e) {  
    logger.log(e);  
}
```

Why not the  
simpler code?



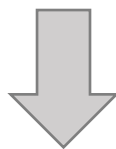
```
try {  
    // some code  
} catch(Exception e) {  
    logger.log(e);  
}
```

catch (Exception e) is often considered a bad practice

- It may not properly handle logics that required for specific exceptions
- It may catch unexpected exceptions
- It may mask the actual error and impeding debugging

# Catching Multiple Exceptions

```
try {  
    // some code  
} catch(FileNotFoundException e) {  
    logger.log(e);  
  
} catch(SQLException e) {  
    logger.log(e);  
  
} catch(SocketException e) {  
    logger.log(e);  
}
```



```
try {  
    // some code  
  
} catch(FileNotFoundException | SQLException | SocketException e) {  
    logger.log(e);  
  
}
```

In Java 7 and later, a single catch block can handle multiple types of exception

- Reduce code duplication
- Avoid using overly broad exception

# try-with-resources

- try-with-resources statement ensures that a resource (e.g., InputStream, JDBC connection) is automatically closed after the program is finished with it

```
Scanner scanner = null;
try {
    scanner = new Scanner(new File("test.txt"));
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
} finally {
    if (scanner != null) {
        scanner.close();
    }
}
```

Before Java 7

```
try (Scanner scanner = new Scanner(new File("test.txt"))) {
    while (scanner.hasNext()) {
        System.out.println(scanner.nextLine());
    }
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
```

Java 7 and later



# try-with-resources

- try-with-resources statement ensures that a resource (e.g., InputStream, JDBC connection) is automatically closed after the program is finished with it

## Syntax Sugar

- Syntax in a programming language that is designed to make things easier to express
- Compiler automatically inserts a “close()” when compiling the source to bytecode



```
try (Scanner scanner = new Scanner(new File("test.txt"))) {  
    while (scanner.hasNext()) {  
        System.out.println(scanner.nextLine());  
    }  
} catch (FileNotFoundException e) {  
    e.printStackTrace();  
}
```

# try-with-resources

- We can also specify multiple resources
- E.g.,: No matter how the block exits, both in and out are closed.

```
try (var in = new Scanner(  
    new FileInputStream("/usr/share/dict/words"), StandardCharsets.UTF_8);  
    var out = new PrintWriter("out.txt", StandardCharsets.UTF_8))  
{  
    while (in.hasNext())  
        out.println(in.next().toUpperCase());  
}
```

# try-with-resources

- Define custom resource:  
Any object that implements the `AutoCloseable` interface and overrides its `close()` method could be used as a resource

```
try (Scanner scanner = new Scanner(new File("test.txt"))) {  
    while (scanner.hasNext()) {  
        System.out.println(scanner.nextLine());  
    }  
} catch (FileNotFoundException e) {  
    e.printStackTrace();  
}
```

```
public class MyResource implements AutoCloseable {  
    @Override  
    public void close() throws Exception {  
        System.out.println("My resource is closed.");  
    }  
}
```

# Rethrowing Exceptions

- If you want to change the exception type, you can throw an exception in a catch clause.

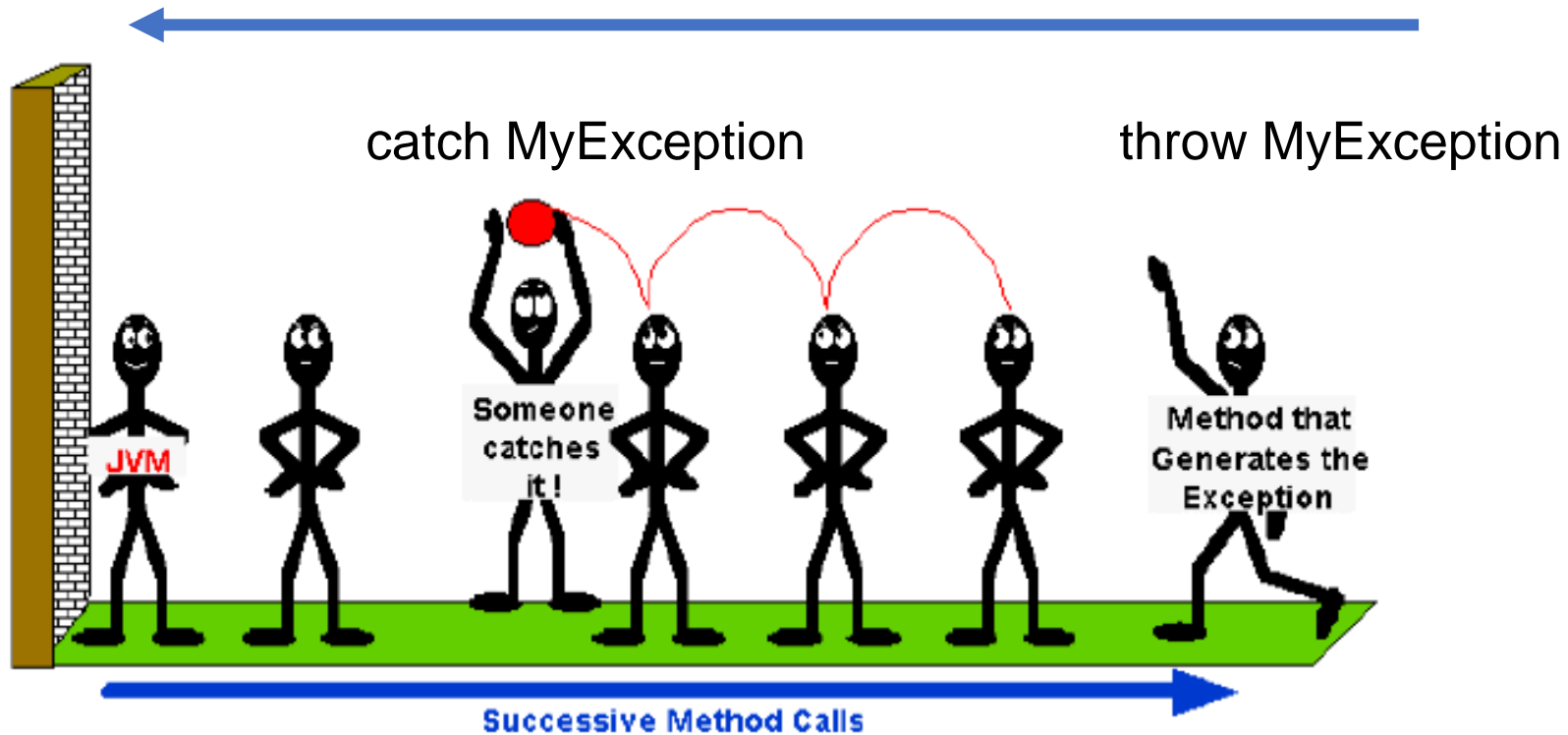
```
try
{
    access the database
}
catch (SQLException original)
{
    var e = new ServletException("database error");
    e.initCause(original);
    throw e;
}
```

This wrapping technique is highly recommended. It allows you to throw high-level exceptions in subsystems without losing the details of the original failure

Reference: Core Java Volume I, 7.2.3

# Exception & Call Stack

JVM searches backward through the call stack to find a matching exception handler (i.e., catch)



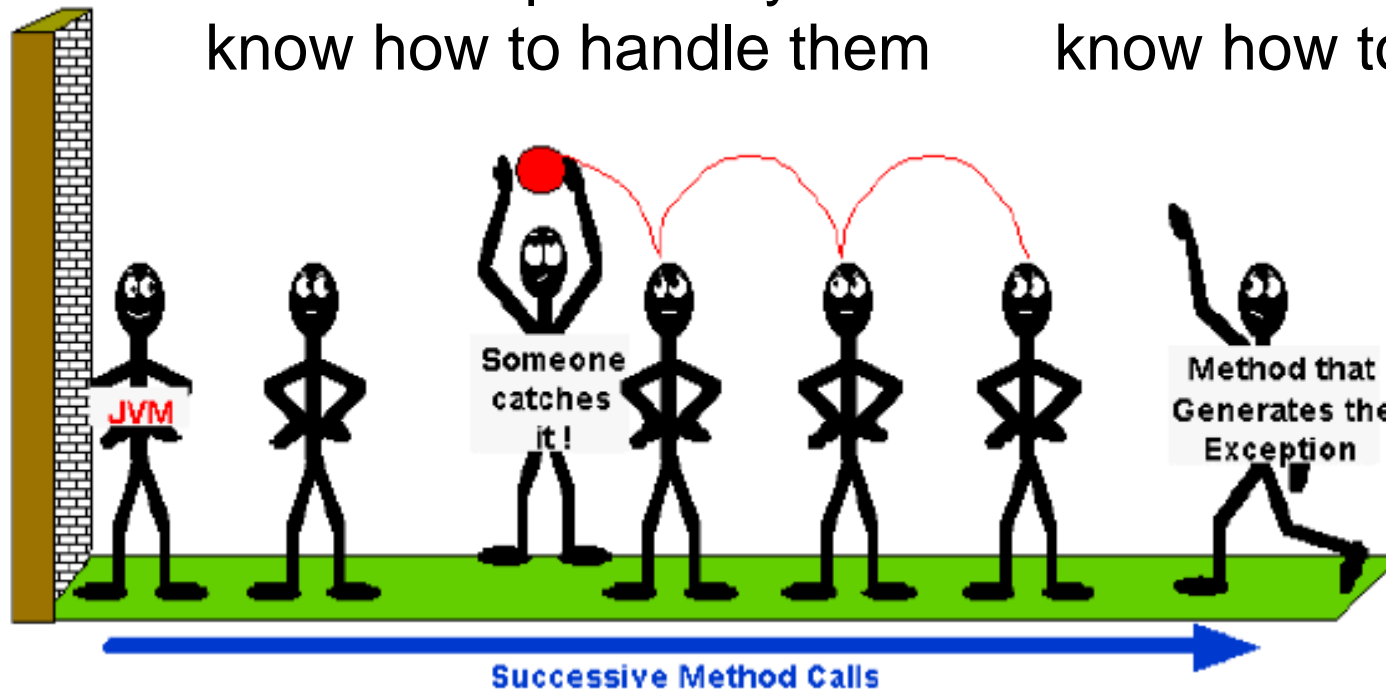
$A() \rightarrow B() \rightarrow C() \rightarrow D() \rightarrow E() \rightarrow F()$

# Where to throw, where to catch?

## A general rule

Catch exceptions if you know how to handle them

Throw exceptions that you do not know how to handle



$A() \rightarrow B() \rightarrow C() \rightarrow D() \rightarrow E() \rightarrow F()$



Where to  
throw,  
where to  
catch?

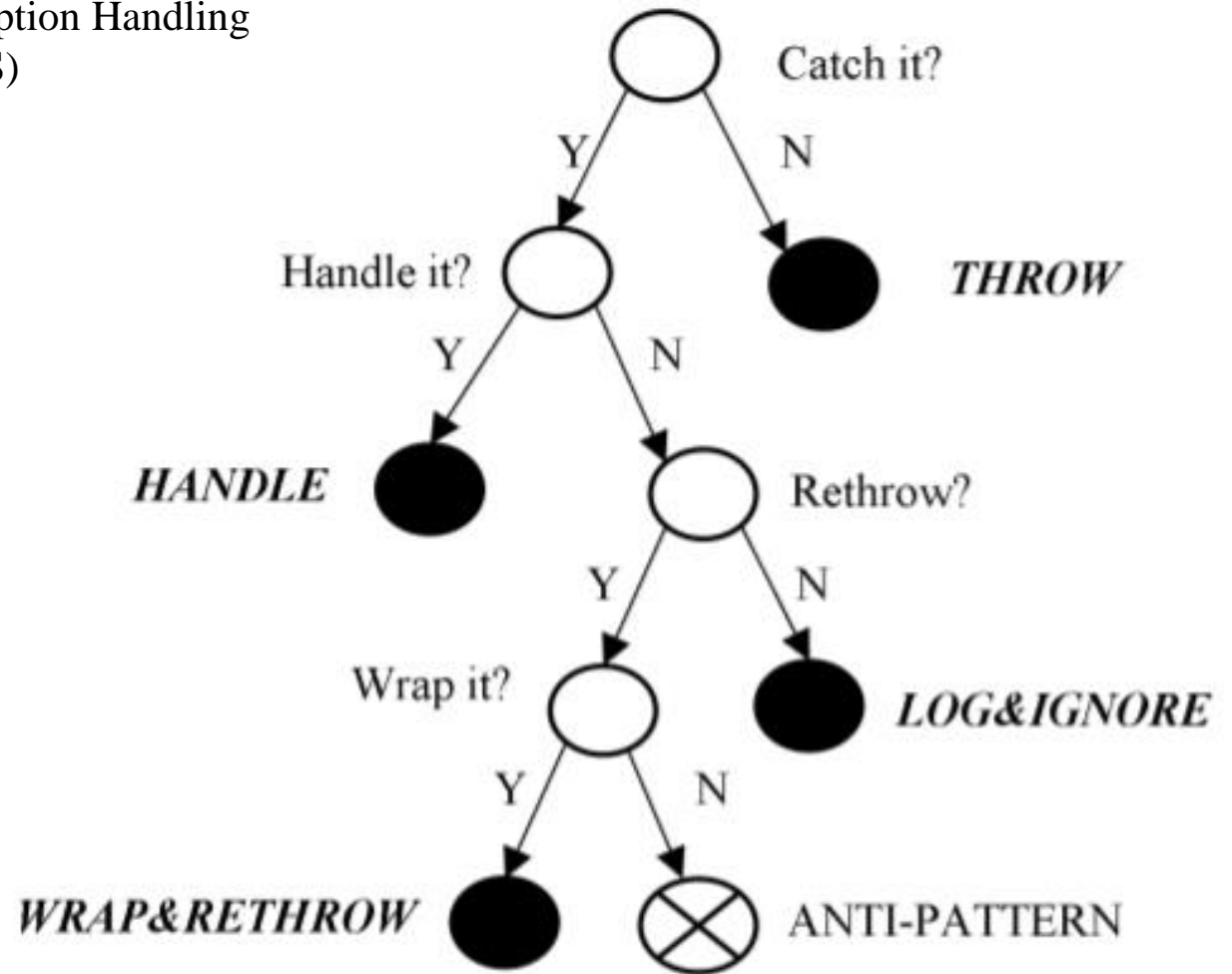


Fig. 1. Decision Process and Exception Handling Strategies

# Where to throw, where to catch?

- Further reading

- Ebert, Felipe, Fernando Castor, and Alexander Serebrenik. "[An exploratory study on exception handling bugs in Java programs.](#)" Journal of Systems and Software 106 (2015): 82-101.
- Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. [Analysis of exception handling patterns in Java projects: an empirical study.](#) MSR'2016
- Y. Li et al., "[EH-Recommender: Recommending Exception Handling Strategies Based on Program Context,](#)" ICECCS'2018
- Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. [Learning to Handle Exceptions.](#) ASE'2020

# Next Lecture

- Multithreading