Principles of Software Construction: Objects, Design, and Concurrency

Object-Oriented Programming in Java

Josh Bloch Charlie Garrod Darya Melicher

Carnegie Mellon University School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 1 due Thursday 11:59 p.m.
  - Everyone must read and sign our collaboration policy
- First reading assignment due Tuesday
  - Effective Java Items 15 and 16

Key concepts from Thursday

- Bipartite type system – primitives & object refs
  - Single implementation inheritance
  - Multiple interface inheritance
- Easiest output – `println` , `printf`
- Easiest input – Command line args, `Scanner`
- Collections framework is powerful & easy to use

Outline

I. Object-oriented programming basics
II. Information hiding
III. Exceptions

institute for SOFTWARE RESEARCH

Objects

- An **object** is a bundle of state and behavior
- State – the data contained in the object
  - In Java, these are the **fields** of the object
- Behavior – the actions supported by the object
  - In Java, these are called **methods**
  - Method is just OO-speak for function
  - Invoke a method = call a function

Classes

- Every object has a class
  - A class defines methods and fields
  - Methods and fields collectively known as **members**
- Class defines both type and implementation
  - Type ≈ where the object can be used
  - Implementation ≈ how the object does things
- Loosely speaking, the methods of a class are its **Application Programming Interface (API)**
  - Defines how users interact with instances

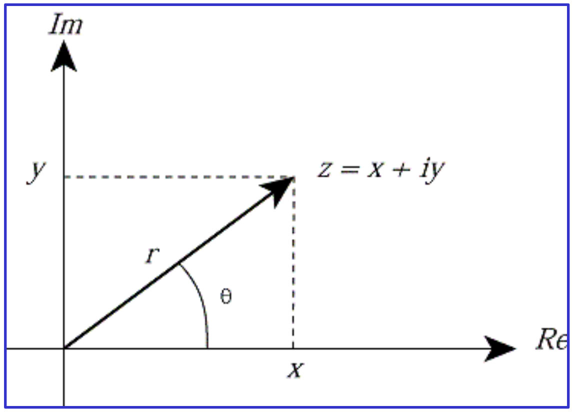
Class example – complex numbers

```
class Complex {
  private final double re;  // Real Part
  private final double im;  // Imaginary Part

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new Complex(re + c.re, im + c.im);
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```



Class usage example

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new Complex(-1, 0);
        Complex d = new Complex(0, 1);

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it prints

```
-1.0 + 1.0i
-0.0 + -1.0i
```

Interfaces and implementations

- Multiple implementations of API can coexist
  - Multiple classes can implement the same API
  - They can differ in performance and behavior
- In Java, an API is specified by *interface* or *class*
  - Interface provides only an API
  - Class provides an API and an implementation
  - A class can implement multiple interfaces

An interface to go with our class

```java
public interface Complex {
    // No constructors, fields, or implementations!

    double realPart();
    double imaginaryPart();
    double r();
    double theta();

    Complex plus(Complex c);
    Complex minus(Complex c);
    Complex times(Complex c);
    Complex dividedBy(Complex c);
}
```

An interface defines but does not implement API

Modifying class to use interface

```
class OrdinaryComplex implements Complex {
  final double re;  // Real Part
  final double im;  // Imaginary Part

  public OrdinaryComplex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

Modifying client to use interface

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new OrdinaryComplex(-1, 0);
        Complex d = new OrdinaryComplex(0, 1);

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it still prints

```
-1.0 + 1.0i
-0.0 + -1.0i
```

Interface permits multiple implementations

```
class PolarComplex implements Complex {
  final double r;
  final double theta;

  public PolarComplex(double r, double theta) {
    this.r = r;
    this.theta = theta;
  }

  public double realPart()      { return r * Math.cos(theta) ; }
  public double imaginaryPart() { return r * Math.sin(theta) ; }
  public double r()             { return r; }
  public double theta()         { return theta; }

  public Complex plus(Complex c)      { ... } // Completely different impls
  public Complex minus(Complex c)     { ... }
  public Complex times(Complex c)     { ... }
  public Complex dividedBy(Complex c) { ... }
}
```

Interface decouples client from implementation

```
public class ComplexUser {
    public static void main(String args[]) {
        Complex c = new PolarComplex(Math.PI,   1);  // -1
        Complex d = new PolarComplex(Math.PI/2, 1);  //  i

        Complex e = c.plus(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
        e = c.times(d);
        System.out.println(e.realPart() + " + "
                           + e.imaginaryPart() + "i");
    }
}
```

When you run this program, it STILL prints

```
-1.0 + 1.0i
-0.0 + -1.0i
```

Why multiple implementations?

- Different performance
  - Choose implementation that works best for your use
- Different behavior
  - Choose implementation that does what you want
  - Behavior *must* comply with interface spec ("contract")
- Often performance and behavior *both* vary
  - Provides a functionality – performance tradeoff
  - Example: `HashSet`, `TreeSet`

Java interfaces and classes

- A type defines a family of objects
  - Each type offers a specific set of operations
  - Objects are otherwise opaque
- Interfaces vs. classes
  - Interface: specifies expectations
  - Class: delivers on expectations (the implementation)

Classes as types

- Classes *do* define types
  - Public class methods usable like interface methods
  - Public fields directly accessible from other classes
- But generally prefer the use of interfaces
  - Use interface types for variables and parameters unless you know a single implementation will suffice
    - Supports change of implementation
    - Prevents dependence on implementation details

```
Set<Criminal> senate = new HashSet<>();          // Do this…
HashSet<Criminal> senate = new HashSet<>();      // Not this
```

Check your understanding

```
interface Animal {
    void vocalize();
}
class Dog implements Animal {
    public void vocalize() { System.out.println("Woof!"); }
}
class Cow implements Animal {
    public void vocalize() { moo(); }
    public void moo() { System.out.println("Moo!"); }
}
```

What Happens?

1. `Animal a = new Animal();`
   `a. vocalize();`
2. `Dog d = new Dog();`
   `d.vocalize();`
3. `Animal b = new Cow();`
   `b.vocalize();`
4. `b.moo();`

Historical note: simulation and the origins of OO programming

- Simula 67 was the first object-oriented language
- Developed by Kristin Nygaard and Ole-Johan Dahl at the Norwegian Computing Center

Dahl and Nygaard at the time of Simula's development

- Developed to support *discrete-event simulation*
  - Application: operations research, e.g. traffic analysis
  - Extensibility was a key quality attribute for them
  - Code reuse was another

Outline

I. Object-oriented programming basics
II. Information hiding
III. Exceptions

institute for SOFTWARE RESEARCH

Information hiding

- Single most important factor that distinguishes a well-designed module from a bad one is the degree to which it hides internal data and other implementation details from other modules
- Well-designed code hides *all* implementation details
  - Cleanly separates API from implementation
  - Modules communicate *only* through APIs
  - They are oblivious to each others' inner workings
- Known as *information hiding* or *encapsulation*
- Fundamental tenet of software design [Parnas, '72]

Benefits of information hiding

- **Decouples** the classes that comprise a system
  - Allows them to be developed, tested, optimized, used, understood, and modified in isolation
- **Speeds up system development**
  - Classes can be developed in parallel
- **Eases burden of maintenance**
  - Classes can be understood more quickly and debugged with little fear of harming other modules
- **Enables effective performance tuning**
  - "Hot" classes can be optimized in isolation
- **Increases software reuse**
  - Loosely-coupled classes often prove useful in other contexts

Information hiding with interfaces

- Declare variables using interface types
- Client can use only interface methods
- Fields not accessible from client code
- But this only takes us so far
  - Client can access non-interface members directly
  - In essence, it's voluntary information hiding

Mandatory Information hiding

visibility modifiers for members

- `private` – Accessible *only* from declaring class
- package-private – Accessible from any class in the package where it is declared
  - Technically known as default access
  - You get this if no access modifier is specified
- `protected` – Accessible from subclasses of declaring class (and within package)
- `public` – Accessible from anywhere

Hiding interior state in OrdinaryComplex

```
class OrdinaryComplex implements Complex {
  private double re;  // Real Part
  private double im;  // Imaginary Part

  public OrdinaryComplex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double realPart()      { return re; }
  public double imaginaryPart() { return im; }
  public double r()             { return Math.sqrt(re * re + im * im); }
  public double theta()         { return Math.atan(im / re); }

  public Complex add(Complex c) {
    return new OrdinaryComplex(re + c.realPart(), im + c.imaginaryPart());
  }
  public Complex subtract(Complex c) { ... }
  public Complex multiply(Complex c) { ... }
  public Complex divide(Complex c)   { ... }
}
```

Discussion

- You know the benefits of private fields
- What are the benefits of private methods?

Best practices for information hiding

- Carefully design your API
- Provide *only* functionality required by clients
  - *All* other members should be private
- You can always make a private member public later without breaking clients
  - But not vice-versa!

Outline

I. Object-oriented programming basics

II. Information hiding

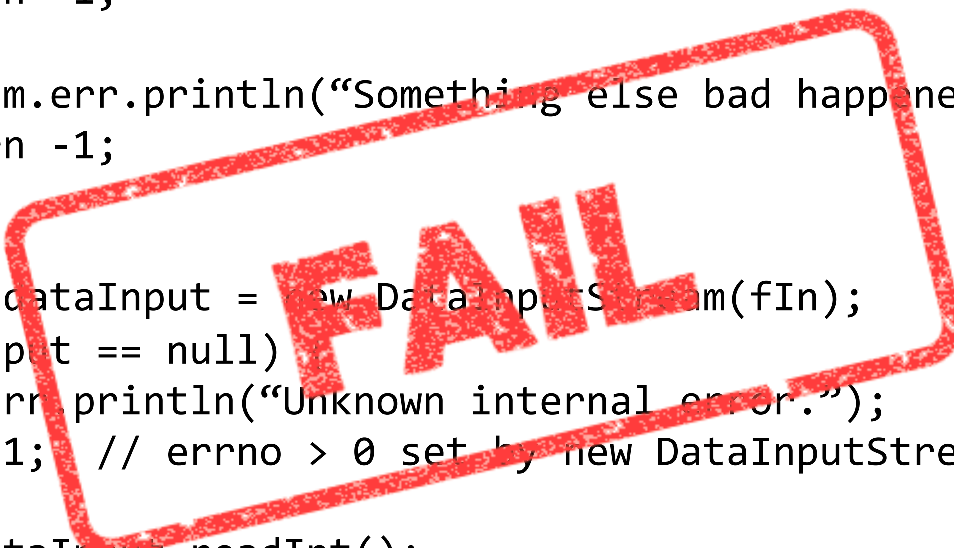
III. Exceptions

What does this code do?

```
FileInputStream fIn = new FileInputStream(fileName);
if (fIn == null) {
  switch (errno) {
  case _ENOFILE:
     System.err.println(“File not found: “ + …);
     return -1;
  default:
     System.err.println(“Something else bad happened: “ + …);
     return -1;
  }
}
DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
  System.err.println(“Unknown internal error.”);
  return -1;  // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
  System.err.println(“Error reading binary data from file”);
  return -1;
}  // The Slide lacks space to close the file.  Oh well.
return i;
```

What does this code do?

```
FileInputStream fIn = new FileInputStream(fileName);
if (fIn == null) {
  switch (errno) {
  case _ENOFILE:
     System.err.println(“File not found: “ + …);
     return -1;
  default:
     System.err.println(“Something else bad happened: “ + …);
     return -1;
  }
}
DataInput dataInput = new DataInputStream(fIn);
if (dataInput == null) {
  System.err.println(“Unknown internal error.”);
  return -1;  // errno > 0 set by new DataInputStream
}
int i = dataInput.readInt();
if (errno > 0) {
  System.err.println(“Error reading binary data from file”);
  return -1;
}  // The Slide lacks space to close the file.  Oh well.
return i;
```



Compare to:

```
FileInputStream fileInput = null;
try {
    fileInput = new FileInputStream(fileName);
    DataInput dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + fileName);
} catch (IOException e) {
    System.out.println("Couldn’t read file: " + e);
} finally {
    if (fileInput != null)
        fileInput.close();
}
```


institute for SOFTWARE RESEARCH

Exceptions

- Notify the caller of an exceptional condition by automatic transfer of control
- Semantics:
  - Propagates up stack until `main` method is reached (terminates program), or exception is caught
- Sources:
  - Program – e.g., `IllegalArgumentException`
  - JVM – e.g., `StackOverflowError`

Control-flow of exceptions

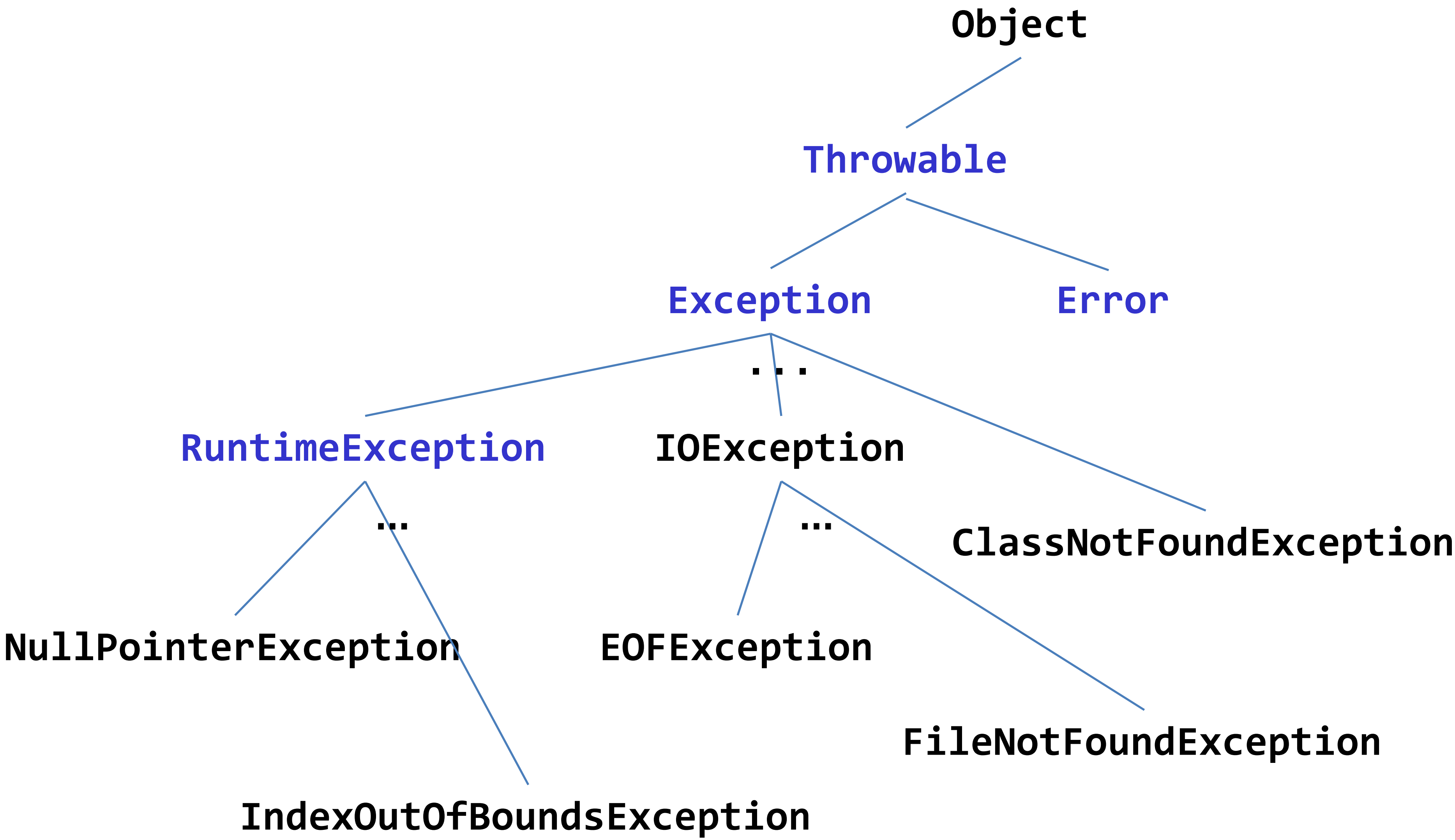
```
public static void main(String[] args) {
    try {
        test();
    } catch (IndexOutOfBoundsException e) {
        System.out.println"("Caught index out of bounds");
    }
}

public static void test() {
    try {
        System.out.println("Top");
        int[] a = new int[10];
        a[42] = 42;
        System.out.println("Bottom");
    } catch (NegativeArraySizeException e) {
        System.out.println("Caught negative array size");
    }
}
```

Checked vs. unchecked exceptions

- Checked exception
  - Must be caught or propagated, or program won't compile
- Unchecked exception
  - No action is required for program to compile
  - But uncaught exception will cause program to fail!

The exception hierarchy in Java



Design choice: checked and unchecked exceptions and return values

- Unchecked exception
  - Programming error, other unrecoverable failure
- Checked exception
  - An error that every caller should be aware of and handle
- Special return value (e.g., `null` from `Map.get`)
  - Common but atypical result
- Do NOT use return codes
- NEVER return `null` to indicate a zero-length result
  - Use a zero-length list or array instead

One more alternative – return `Optional<T>`

- `Optional<T>` is a single T instance or nothing
  - A value is said to *present*, or the optional is *empty*
  - Can think of it as a *subsingleton* collection
- Similar in spirit to checked exceptions
  - **Force** caller to confront possibility of no value
- But optionals demand less boilerplate in client
- Can be tricky to decide which alternative to use
- *See Effective Java* Item 55 for more information

A sample use of `Optional<T>`

```
// Returns maximum value in collection as an Optional<E>
public static <E extends Comparable<E>>
              Optional<E> max(Collection<E> c) {
    if (c.isEmpty())
        return Optional.empty();


    E result = null;
    for (E e : c)
        if (result == null || e.compareTo(result) > 0)
            result = Objects.requireNonNull(e);

    return Optional.of(result);
}
```

Creating and throwing your own exceptions

```
public class SpanishInquisitionException extends RuntimeException {
    public SpanishInquisitionException() {
    }
}

public class HolyGrail {
    public void seek() {
        ...
        if (heresyByWord() || heresyByDeed())
            throw new SpanishInquisitionException();
        ...
    }
}
```


isr institute for SOFTWARE RESEARCH

Benefits of exceptions

- You can’t forget to handle common failure modes
  - Compare: using a flag or special return value
- Provide high-level summary of error, and stack trace
  - Compare: core dump in C
- Improve code structure
  - Separate normal code path from exceptional
  - Ease task of recovering from failure
- Ease task of writing robust, maintainable code

Guidelines for using exceptions (1)

- Avoid unnecessary checked exceptions (EJ Item 71)
- Favor standard exceptions (EJ Item 72)
  - `IllegalArgumentException` – invalid parameter value
  - `IllegalStateException` – invalid object state
  - `NullPointerException` – null param where prohibited
  - `IndexOutOfBoundsException` – invalid index param
- Throw exceptions appropriate to abstraction (EJ Item 73)

institute for SOFTWARE RESEARCH

Guidelines for using exceptions (2)

- Document all exceptions thrown by each method
  - Checked and unchecked (EJ Item 74)
  - But don't *declare* unchecked exceptions!
- Include failure-capture info in detail message (Item 75)
  - ```
    throw new IlegalArgumentException(
        "Modulus must be prime: " + modulus);
    ```
- Don't ignore exceptions (EJ Item 77)

```
// Empty catch block IGNORES exception – Bad smell in code!
try {
    ...
} catch (SomeException e) { }
```


institute for SOFTWARE RESEARCH
```

Remember this slide?

You can do much better!

```
FileInputStream fileInput = null;
try {
    FileInputStream fileInput = new FileInputStream(fileName);
    DataInput dataInput = new DataInputStream(fileInput);
    return dataInput.readInt();
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + fileName);
} catch (IOException e) {
    System.out.println("Couldn’t read file: " + e);
} finally {
    if (fileInput != null) fileInput.close();
}
```

Manual resource termination is ugly and error prone

- Even good programmers usually get it wrong
  - Sun's Guide to Persistent Connections got it wrong in code that claimed to be exemplary
  - Solution on page 88 of Bloch and Gafter's *Java Puzzlers* is badly broken; no one noticed for years
- 70% of the uses of the `close` method in the JDK itself were wrong in 2008(!)
- Even "correct" idioms for manual resource management are deficient

The solution: `try`-with-resources (TWR)

Automatically closes resources

```
try (DataInput dataInput =
        new DataInputStream(new FileInputStream(fileName))) {
    return dataInput.readInt();
} catch (FileNotFoundException e) {
    System.out.println("Could not open file " + fileName);
} catch (IOException e) {
    System.out.println("Couldn’t read file: " + e);
}
```

File copy without TWR

```
static void copy(String src, String dest) throws IOException {
    InputStream in = new FileInputStream(src);
    try {
        OutputStream out = new FileOutputStream(dest);
        try {
            byte[] buf = new byte[8 * 1024];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
            } finally {
                out.close();
            }
        } finally {
            in.close();
        }
    }
}
```

File copy with TWR

```
static void copy(String src, String dest) throws IOException {
    try (InputStream in = new FileInputStream(src);
         OutputStream out = new FileOutputStream(dest)) {
        byte[] buf = new byte[8 * 1024];
        int n;
        while ((n = in.read(buf)) >= 0)
            out.write(buf, 0, n);
    }
}
```

Summary

- Interface-based designs handle change well
- Information hiding is crucial to good design
- Exceptions are far better than error codes
- The need for checked exceptions is rare
- `try`-with-resources (TWR) is a big win