Warmup

Trace the execution of the following code:

```
Jet[] fleet = new Jet[]{ new U2() };
Vehicle leader = fleet[0];
System.out.println(leader.toString());
```

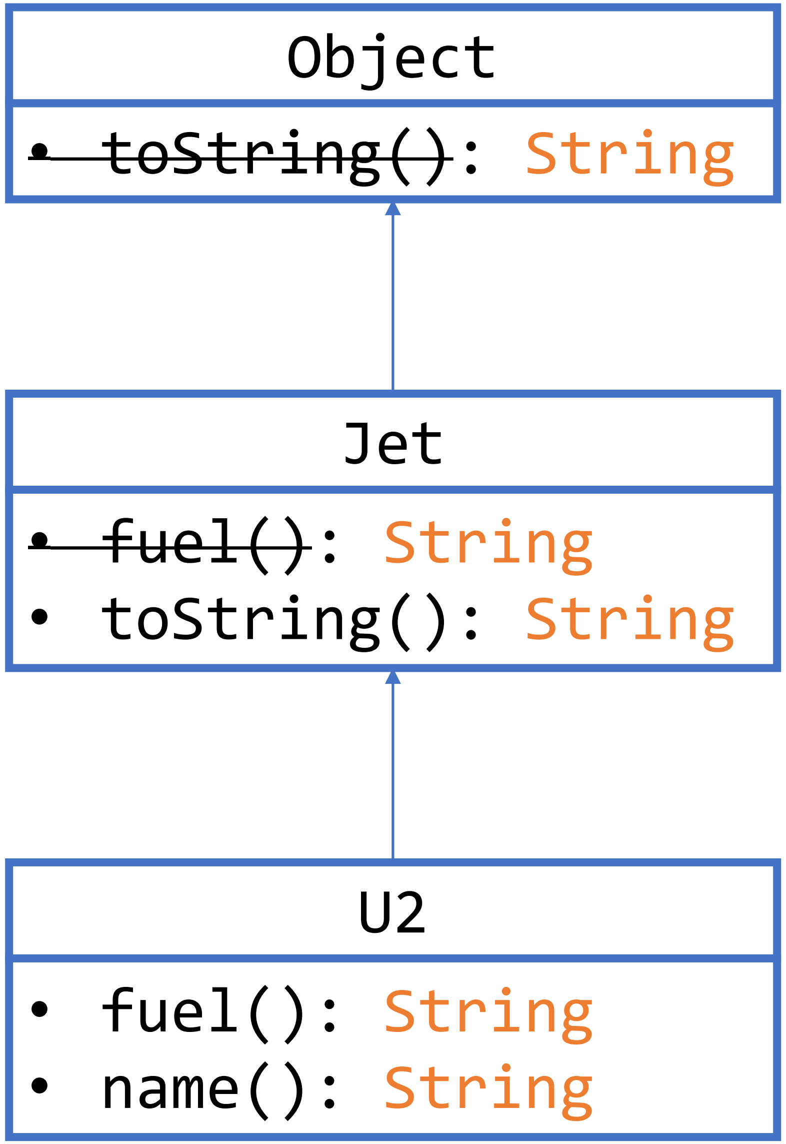
```
interface Vehicle {
    String name();
    String fuel();
}
abstract class Jet implements Vehicle {
    @Override public String fuel() {
        return "JP-8";
    }
    @Override public String toString() {
        return name() + " consuming " + fuel();
    }
}
class U2 extends Jet {
    @Override public String name() {
        return "U-2";
    }
    @Override public String fuel() {
        return "JPTS";
    }
}
```

Warmup

Trace the execution of the following code:

```
Jet[] fleet = new Jet[]{ new U2() };
Vehicle leader = fleet[0];
System.out.println(leader.toString());
```

A. “Jet consuming JP-8”
B. “U-2 consuming JPTS”
C. “U2@8D06F00D”
D. Compile-time error
E. Runtime error



Object

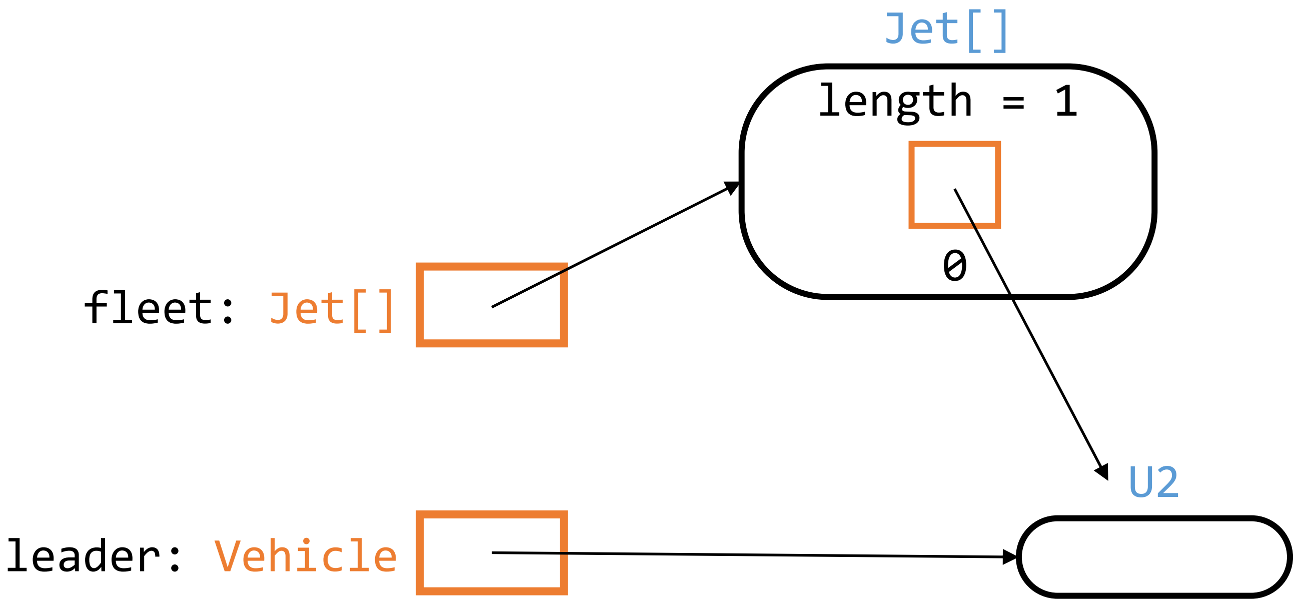
- ~~toString()~~: String

Jet

- ~~fuel()~~: String
- toString(): String

U2

- fuel(): String
- name(): String



Jet[]

length = 1

0

fleet: Jet[]

U2

leader: Vehicle

```
Jet[] fleet = new Jet[]{ new U2() };
Vehicle leader = fleet[0];
System.out.println(leader.toString());
```

CS 2110
Lecture 7

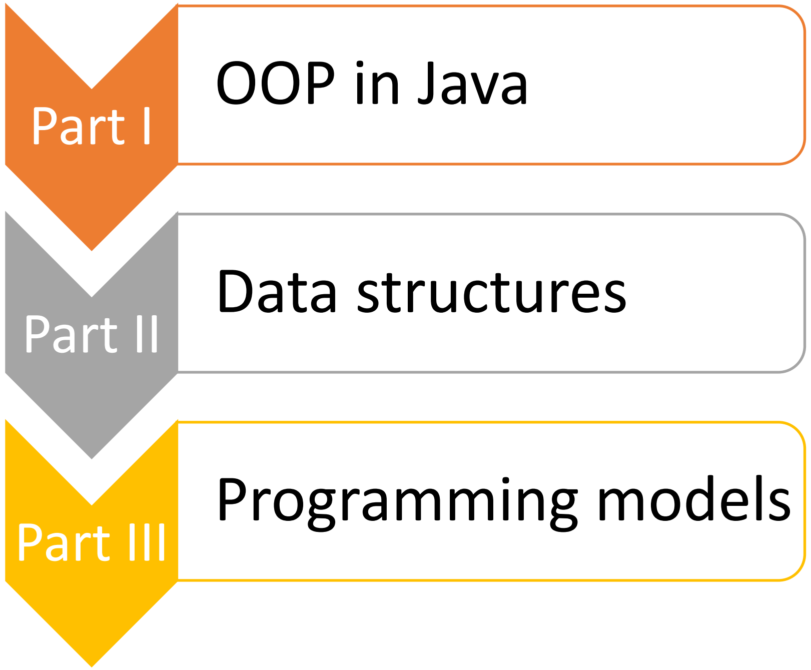
Exceptions, autoboxing, I/O

Coming up

A2 due Thursday

A1 resubmissions

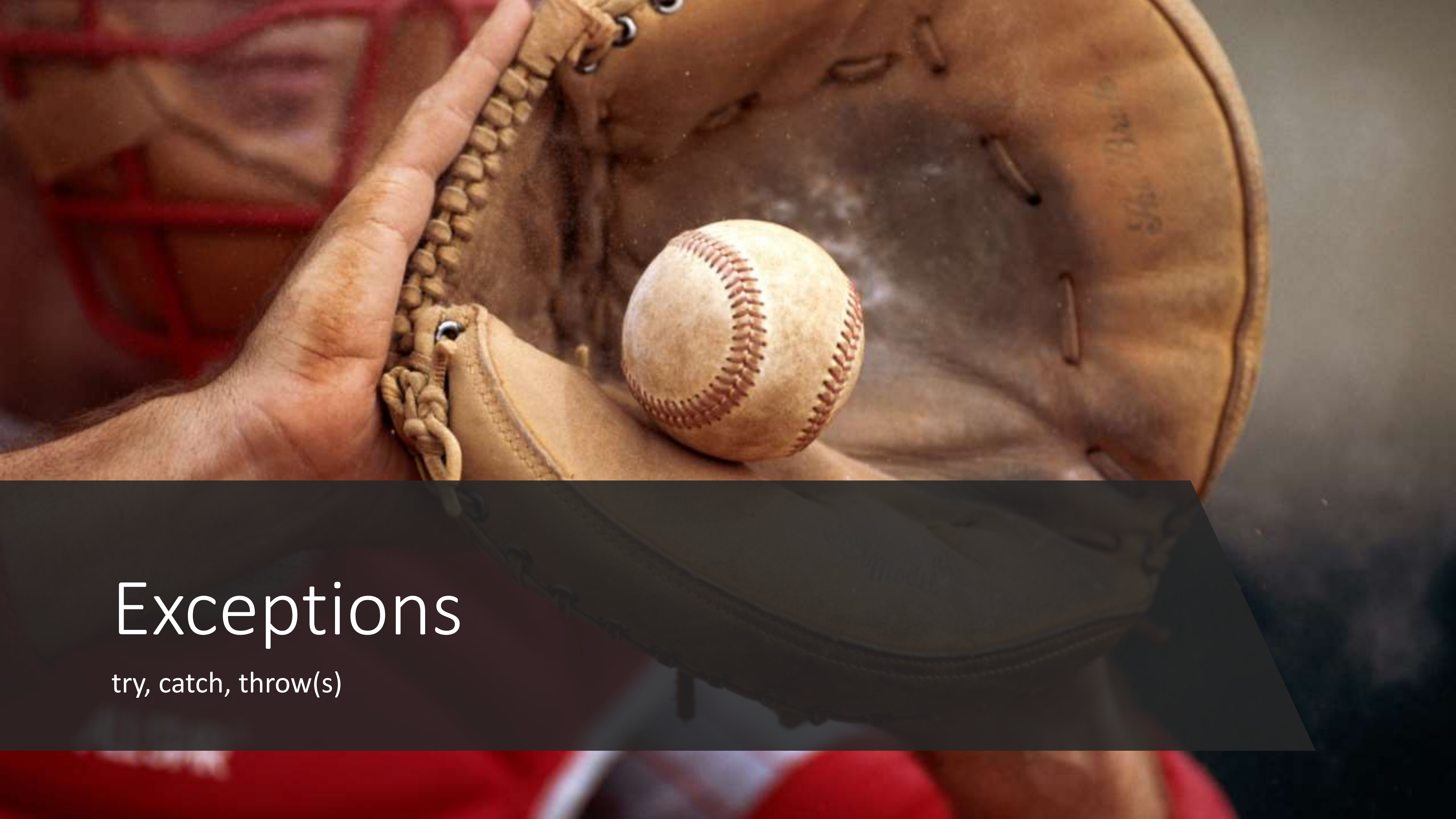
Roadmap



Part I — OOP in Java

Part II — Data structures

Part III — Programming models



Exceptions

try, catch, throw(s)

Sometimes things go wrong

- Negative array indices
- Invoking methods on `null`
- Lost WiFi connection
- Optional feature not supported
- File didn’t contain a valid image
- User typed their email when asked for their age

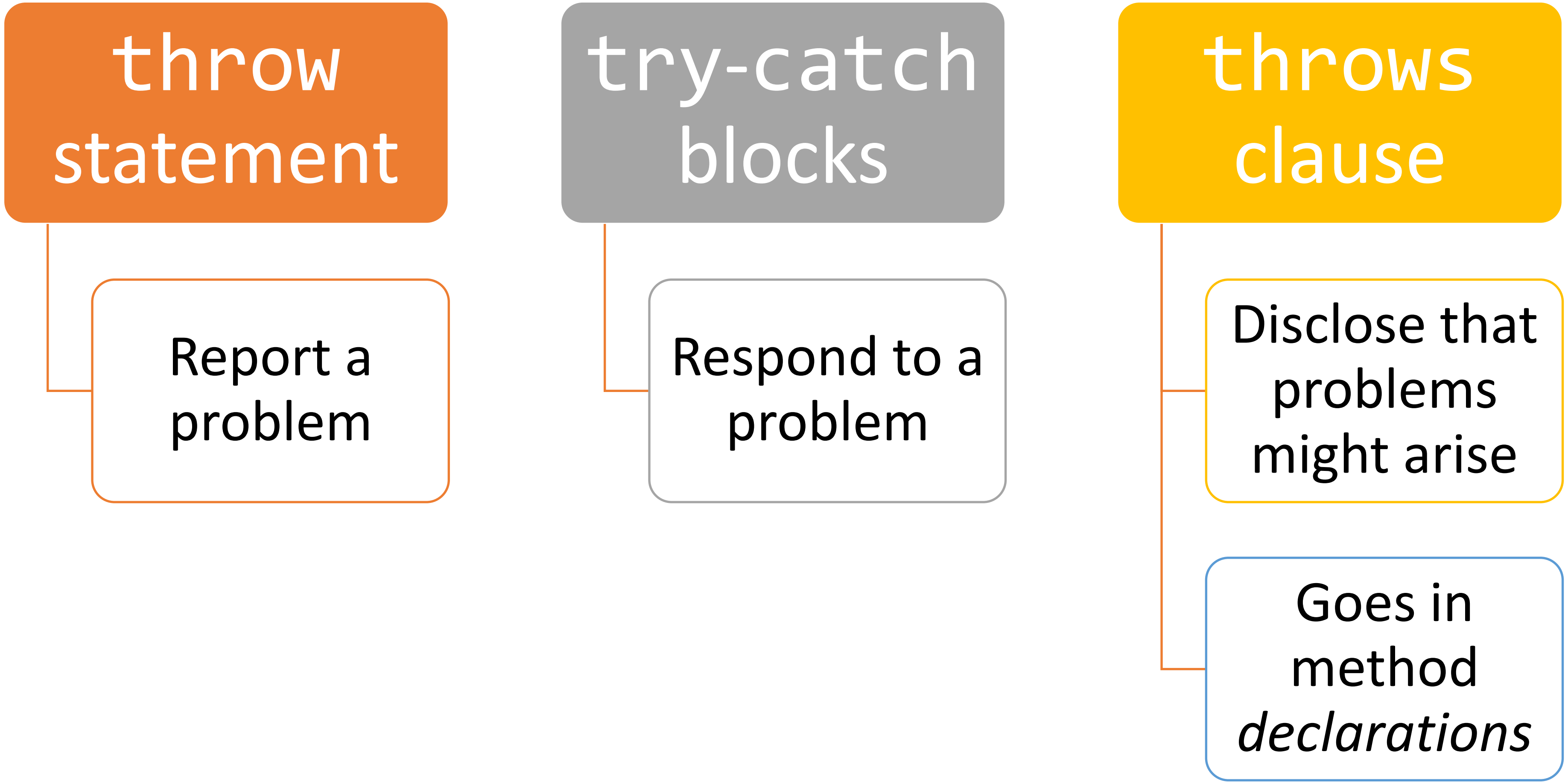
- Can’t just give up or claim “undefined behavior” all the time

Expecting the unexpected

Specifications should define what happens in “exceptional” situations

- Possible responses:
  - Disallow in preconditions
    - Assumes client can *predict* the problem
  - Return a “special value” (-1, `null`)
    - Examples: `String.indexOf()`, `BufferedReader.readLine()`
    - Client might not check value before using it
    - How to get more info?
  - Return a type that can represent success or failure
    - Example: `Optional`
    - Client *must* confront the possibility of failure
  - Throw an **exception**

New syntax



`throw` statement

Report a problem

`try-catch` blocks

Respond to a problem

`throws` clause

Disclose that problems might arise

Goes in method *declarations*

1: Signaling a problem (throw)

- Use the throw keyword, followed by a Throwable object
- Method execution immediately ends (like return)
- Method will *not* yield a value, so no need to fake an answer
    - Example: TODOs in assignments

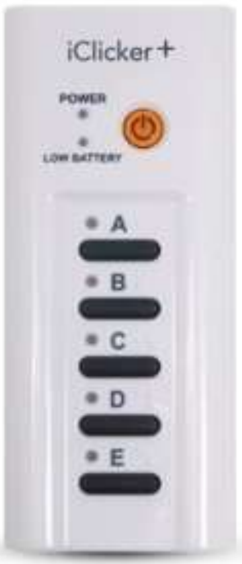
```
if (cmd.equals(
    "open the pod bay doors") {
  throw new
    UnsupportedOperationException(
     "I’m afraid I can’t do that");
} else {
  return true;
}
```

Propagation

```
void f1() {
  print("A");
  f2();
  print("B");
}
void f2() {
  f3(true);
  print("C");
}
void f3(boolean x) {
  if (x) { throw new
           RuntimeException(); }
  print("D");
}
```

What would be printed by running `f1();`? (ignoring any exception backtrace)

A. A
B. AB
C. ACB
D. ADCB
E. other



Backtraces

- Uncaught exceptions will print a backtrace (aka stack trace)
  - Show's the exception's message
  - Shows which line of code threw the exception
  - Shows which method called which method … called the method that threw the exception
- Very helpful for debugging!
  - Know which lines of code were run and which were not

```
Exception in thread "main"
java.lang.RuntimeException: x
should have been false
      at Demo1.f3(Demo1.java:16)
      at Demo1.f2(Demo1.java:9)
      at Demo1.f1(Demo1.java:4)
      at Demo1.main(Demo1.java:23)
```

2: Catching exceptions (try-catch)

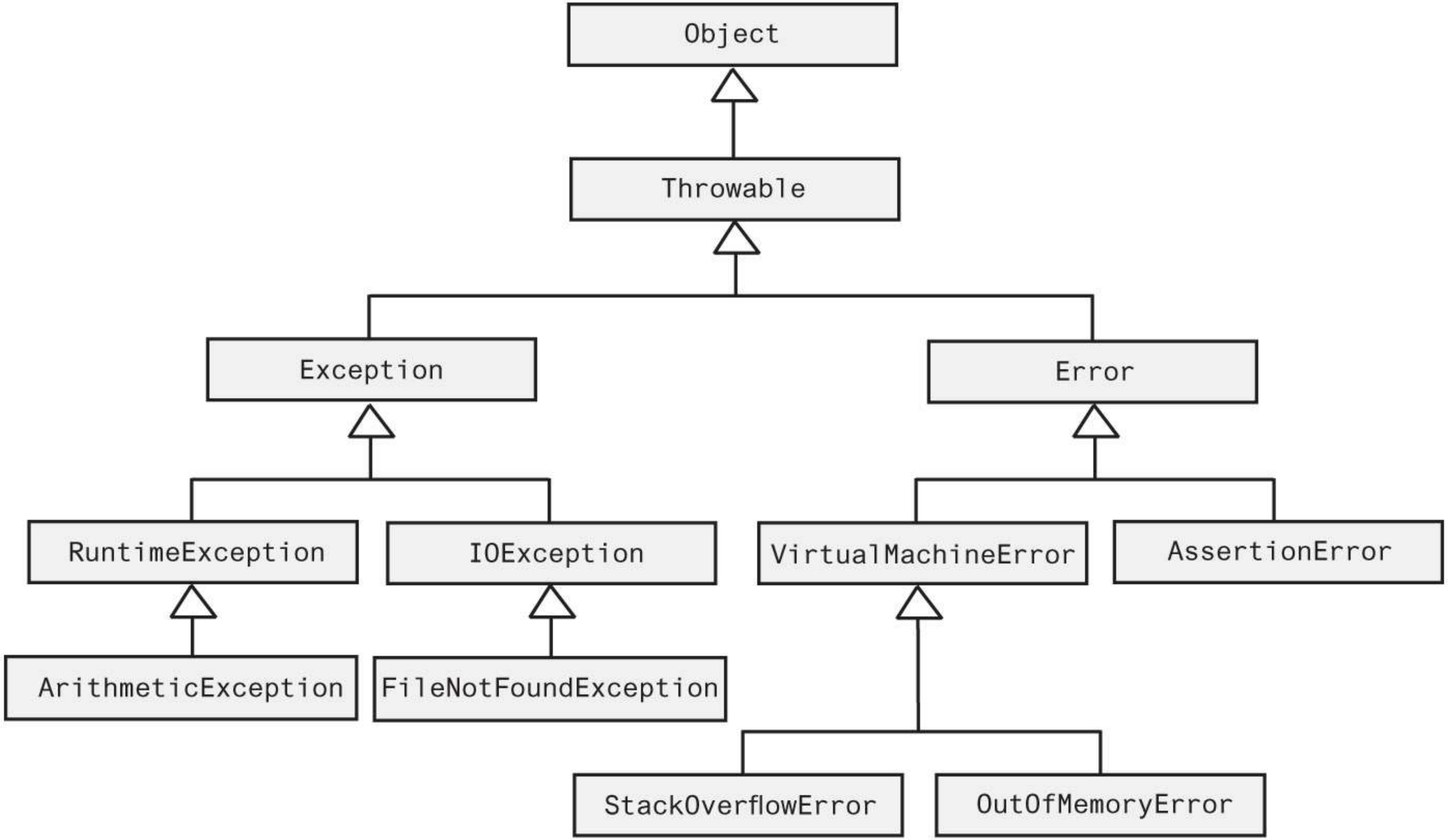
```
try {
  f1();
  // Code that assumes
  // successful f1...
} catch (Exception e) {
  // Code that handles
  // unsuccessful f1...
}
// Code that continues
// either way...
```

- Wrap operations that might throw an exception in a try block
- If an exception is thrown, control will exit the try block and jump to the appropriate catch block
  - At most one catch block is executed; control then jumps to end of entire try/catch statement
  - If no matching catch block, exception *propagates* (exits blocks and methods until caught)

Matching exception types

```
try {
  riskyCall();
} catch
  (FileNotFoundException e) {
  // Handle missing file
} catch (IOException e) {
  // Handle other R/W issue
} catch (Exception e) {
  // Handle other issue
}
// Keep going...
```

- The *first* catch block that catches a supertype of the dynamic type of the thrown object will be executed



© 2019 Pearson Education, Inc.

3: Disclosing possible exceptions (throws)

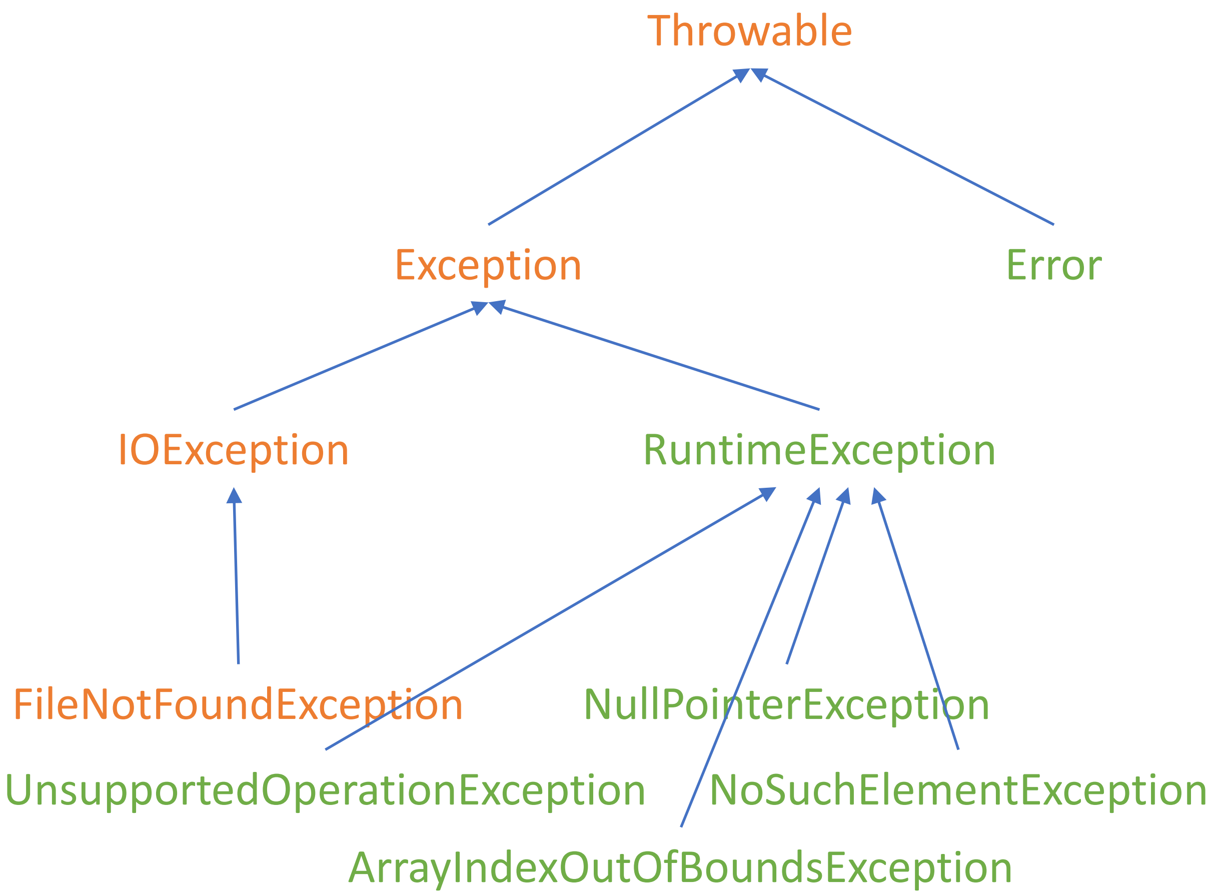
```
int read()
    throws IOException {
  …
  if (failed) {
    throw new IOException();
  }
  …
  return charRead;
}
```

- Inform clients that there are ways this method could fail
  - Lists which *types* of exceptions they should be prepared to handle
- Exceptional circumstances should be elaborated in spec
- Method body might or might not contain a `throw` statement
  - It could call another method that throws

Example of throws

```
/** Read a list of students from the text file at
  * `filename` (one line per name) and return the
  * set of unique students.  Throws
  * `FileNotFoundException` if the file cannot be
  * opened. */
StudentSet readRoster(String filename)
        throws FileNotFoundException {
    // Code that either throws a ne
    // FileNotFoundException, or calls another
    // method that does.
}
```

Exception classes



- Throwables come in two varieties: checked & unchecked

- Error: Serious problem; program should probably just crash
- RuntimeException: Usually a bug the client could have prevented
- Exception: All other exceptional circumstances

Checked vs. unchecked exceptions

Checked

- If you might throw one yourself or might allow one to propagate, *must* add `throws` clause to method declaration
  - Consequence: cannot throw *new kinds* of checked exceptions if overriding

Unchecked

- May throw or allow to propagate without warning
  - Every integer division
  - Every array access
  - Every method call

Handling exceptions

Option A: Catch

- Use a `try` block paired with an appropriate `catch` block
- Client execution resumes after `catch` block
- Use when you know how to handle the situation and can proceed

Option B: Propagate

- Do nothing (need a `throws` clause in declaration if exception type is “checked”)
- Method exits if exception is thrown; control passes to caller
- Use when you needed success in order to proceed; let supervisor figure out what to do now

Permission vs. forgiveness: example 1

Asking permission (preferred)

```
if (selfieCam != null) {
  if (selfieCam.focuser != null) {
    selfieCam.focuser.enableAF();
  } else {
    // Can’t enable autofocus
  }
} else {
  // No selfie camera
}
```

Asking forgiveness (no advantage)

```
try {
  selfieCam.focuser.enableAF();
} catch (NullPointerException e) {
  // Can’t enable autofocus
}
```

- Don’t know which variable was null
- Not avoiding any redundancy
- Catching `NullPointerException` is often considered “code smell”

Permission vs. forgiveness: example 2

Asking permission (redundant)

```
int num = 0;

String token = …;

if (token only contains digits &&
    token is not too long &&
    converted number would not be
      too large && …) {

  num = Integer.parseInt(token);

} else {

  // token is not a valid int

}
```

Asking forgiveness (preferred)

```
int num = 0;

String token = …;

try {

  num = Integer.parseInt(token);

} catch (NumberFormatException e) {

  // token is not a valid int

}
```

- Avoids redundancy

(auto)boxing

Bridging primitives and objects

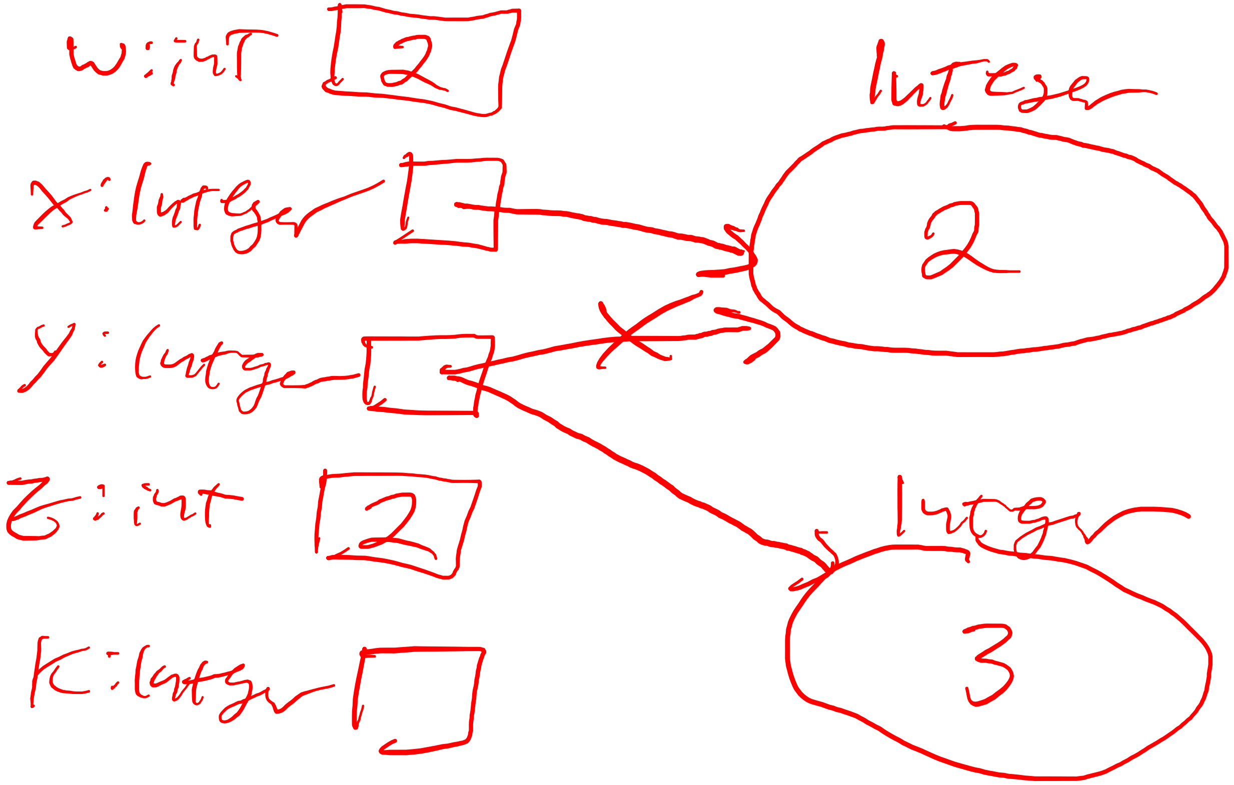
Wrapper classes

- Each primitive type has an associated class
  - `Integer`, `Double`, `Boolean`, `Character`, …
  - Also home to useful utility functions (read the docs)
- An instance represents a single, immutable value
- Can be used where `Object`s are expected
  - E.g. in generic data structures (next few lectures)
  - Must use `equals()` to compare two boxed values
- Java will automatically convert between primitives and wrapper objects when needed

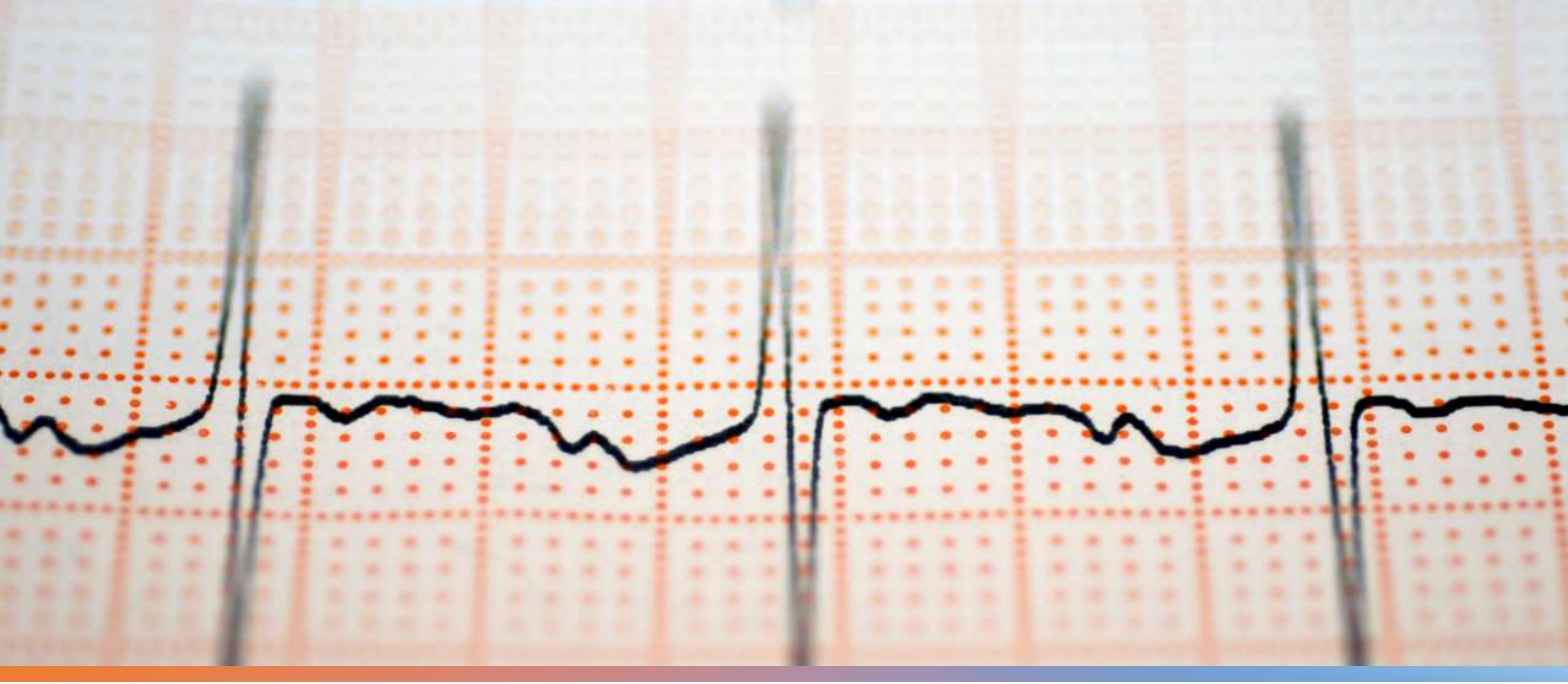
Autoboxing example

```
int w = 2;
Integer x = w;
Integer y = x;
int z = y;


w == z  // true
x.equals(y)  // true
```



y += 1   Integer K = w



Input and output

References:

Textbook Supplement 2

Website reading

Command line I/O

- Most code exchanges data via function arguments and return values
- Some portions of code interact with users
  - Text output: `System.out.println()`
  - Text input: `System.in`, `Scanner` (discussed today)

- In Java, these classes also work for **files**

Files

- Data that is persisted or shared between programs are stored in **files**
    - Examples: text documents, photos
    - Program memory is not persistent (variables disappear when program exits)
- All files look the same to software
    - A sequence of **bytes** (integers between –128 and +127)
    - Programs interpret byte sequence to display human-readable content
- Important case: text files
    - Interpret byte (sub)sequence as a string of letters (characters)
        - No formatting! All data is text
    - Text editor vs. word processor (recommendation: use IntelliJ to edit text files)

Java's IO interfaces (actually abstract classes)

Input

- `InputStream`
  - Read raw bytes
- **`Reader`**
  - Read characters (assuming specified encoding)

Output

- `OutputStream`
  - Write raw bytes
- **`Writer`**
  - Write characters (assuming specified encoding)

File paths

- Files have a filename on your computer's filesystem
  - Example: `Main.java`
- Files live in folders; the list of folders, separated by '`/`', is the file's path
  - Example: `/home/bob/cs2110/a3/Main.java`
- Absolute paths start at the filesystem "root"
  - Examples: `C:/` (Windows; yes, / can be used instead of \), `/` (Mac/Linux)
- Relative paths assume you start from a particular folder
  - Applications are run from a "current directory" (when running from inside IDEA, this is your project's folder)
  - Example: `cs2110/a3/Main.java` (no leading slash) when current directory is `/home/bob`

Demo: Open a text file in an IDEA project

```
import java.io.Reader;
import java.io.FileReader;

Reader in = new FileReader("hello.txt");
```

Error handling

- I/O routinely fails! Examples (brainstorm):
    - Unplug USB drive
    - Internet outage
    - Disk is full
    - Typo in filename

- Most I/O methods (including constructors) can throw an `IOException`, which *must* be accounted for (caught or rethrown)

Demo: Handling I/O exceptions

```
String path = "hello.txt";
try {
    Reader in = new FileReader(path);
    // ...
} catch (FileNotFoundException e) {
    System.err.println("Could not open file " +
                       path);
    System.exit(1);
}
```

Exiting

- Like a return value from main()
  - Except not composable with other Java code!
  - (Multiple programs could be composed using *shell scripts*)
- Rule for CS 2110 (and good advice elsewhere): only call System.exit() from main().
  - If called from anywhere else, MUST document with “effects” clause in spec

Reading more than a char

- `java.util.Scanner`: read logical chunks ("tokens") of text, one at a time
    - Assumes tokens are separated by space
- Construct around a `Reader`
    - `Scanner sc = new Scanner(in);`
    - Or construct around a `String`
- Methods
    - `next()`: Next token as a String
    - `nextLine()`: Remainder of the current line, as a String
    - `nextInt()`, `nextDouble()`, …: Convert next token to a number
    - `hasNext()`, `hasNextLine()`: Is there still more input to read?
    - `useDelimiter()`: split tokens on something other than space

Files as resources

- Important to close files when no longer needed (especially after writing to them)
  - Should always `close()` them, even if an exception occurs
  - Programmers often forgot; source of bugs
- Convenient syntax: try with resources

```
try (Reader in = new FileReader(path)) {
    // …
} catch (IOException e) {
    // …
}
```