Lecture 11: Exceptions, Immutability, and Object

CS 2110, Matt Eichhorn and Leah Perlmutter

September 30, 2025

Announcements

- A4 grades published
- Prelim 1 on October 9 (in 2 weeks)
  - Practice exam coming soon
  - Make sure you’re on top of studying!

Today's Learning Outcomes

Exceptions & Exception Handling

1. Explain exceptions and their relationship to specifications and defensive programming.
2. Write code that throws, propagates, and handles exceptions.

Mutability and Immutability

1. Determine whether a class is mutable or immutable.
2. Explain the semantics of the final keyword.

Object Class and its Methods

1. Describe the semantic differences between the == operator and the equals() method and determine the appropriate one for a given scenario.
2. Identify the requirements of the equals() method specified in the Object class and override this method in user-defined classes.

Exceptions and Exception Handling

Sometimes things go wrong

- …

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

Demo: mean

Expecting the unexpected

Specifications should define what happens in “exceptional” situations

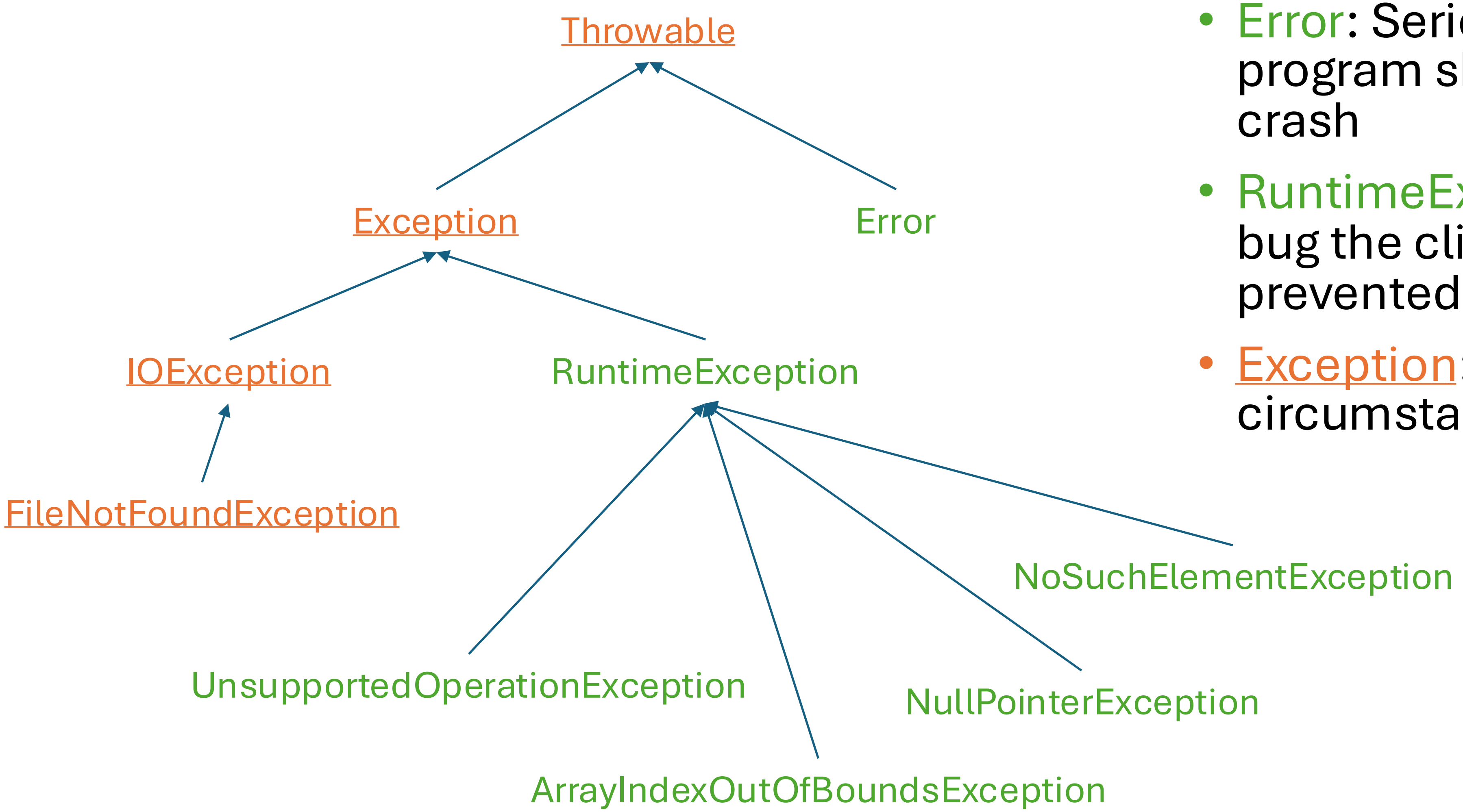
- Possible responses:
  - Disallow in preconditions
    - Assumes client can predict the problem
  - Return a “special value” (-1, `null`)
    - Examples: `String.indexOf()`, `BufferedReader.readLine()`
    - Client might not check value before using it
    - How to get more info?
  - Return a type that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

Signaling a problem – Throwing exceptions

- 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;
}
```

Exception classes



- Throwables come in two varieties: checked & unchecked (by the compiler)

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

Demo: findLocalMax()

Handling exceptions

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

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

Catching exceptions

```
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)

Demo: reduceMax()

Propagation

```java
public static void f1() {
  System.out.println("A");
  f2();
  System.out.println("B");
}
public static void f2() {
  f3(true);
  System.out.println("C");
}
public static void f3(boolean x) {
  if (x) {
    throw new RuntimeException();
  }
  System.out.println("D");
}
```

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

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

PollEv.com/leahp
text **leahp** to **22333**

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:12)

        at Demo1.f2(Demo1.java:8)

        at Demo1.f1(Demo1.java:4)

        at Demo1.main(Demo1.java:17)
```

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
- When a supertype and subtype are included among types to catch, put the subtype first!

Recall:
FileNotFoundException <:
IOException <:
Exception

Checked vs. unchecked exceptions

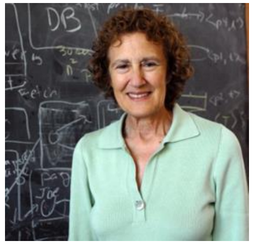
Reminder: Checked means checked by the compiler, and depends on the exception's compile-time type

Checked

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

Unchecked

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



Substitutatiliby!
Principle of Least Surprise!

Demo

- **`parseBookList()`** in starter code from **`BookSorter`**

Testing Exceptions

- **`ExceptionTest`**

Exceptions: Summarize what you learned

- …

Mutability and Immutability

Mutability and Immutability

- mutable – can be modified
- immutable – cannot be modified

`Account`	mutable (can change its balance)
`String`	immutable (a new one gets created for every operation!)

Why immutable?

- avoid representation exposure to maintain representation invariant
- simplicity
- less space for bugs to creep in

Point

```
/** An immutable class representing a point in the 2D coordinate plane with `double`
  coordinates. */
public class Point {
  /** The x-coordinate of this point. */
  private double x;

  /** The y-coordinate of this point. */
  private double y;

  /** Constructs a `Point` object with the given `x`- and `y`-coordinates. */
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  /** Returns the x-coordinate of this point. */
  public double x() {
    return x;
  }

  /** Returns the y-coordinate of this point. */
  public double y() {
    return y;
  }
}
```

reflectOver()

```
/** MUTATOR (doesn't work with immutability)
 * Reflects this point over the line y = `m`x + `b` for the given slope `m`
 * and y-intercept `b`.
 */
public Point reflectOver(double m, double b) {
  this.x = this.x - 2 * b * m + 2 * m * this.y - this.x * m * m;
  this.y = 2 * this.x * m + 2 * b + m * m * this.y - this.y;
  double d = 1 + m * m;
  this.x /= d;
  this.y /= d;
}

/** CREATOR (works with immutability)
 * Returns a new `Point` object that is obtained by reflecting this point about
 * the line y = `m`x + `b` for the given slope `m` and y-intercept `b`.
 */
public Point reflectOver(double m, double b) {
  double xp = this.x - 2 * b * m + 2 * m * this.y - this.x * m * m;
  double yp = 2 * this.x * m + 2 * b + m * m * this.y - this.y;
  double d = 1 + m * m;
  return new Point(xp / d, yp / d);
}
```

Enforcing immutability

```
/** An immutable class representing a point in the 2D coordinate plane with
  `double` coordinates. */
public class Point {
  /** The x-coordinate of this point. */
  private final double x;

  /** The y-coordinate of this point. */
  private final double y;

  /** Constructs a `Point` object with the given `x`- and `y`-coordinates. */
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

 /** Returns a new `Point` obtained by reflecting this point about
  * the line y = `m`x + `b` for the given slope `m` and y-intercept `b`.   */
  public Point reflectOver(double m, double b) {...}

  ...
}
```

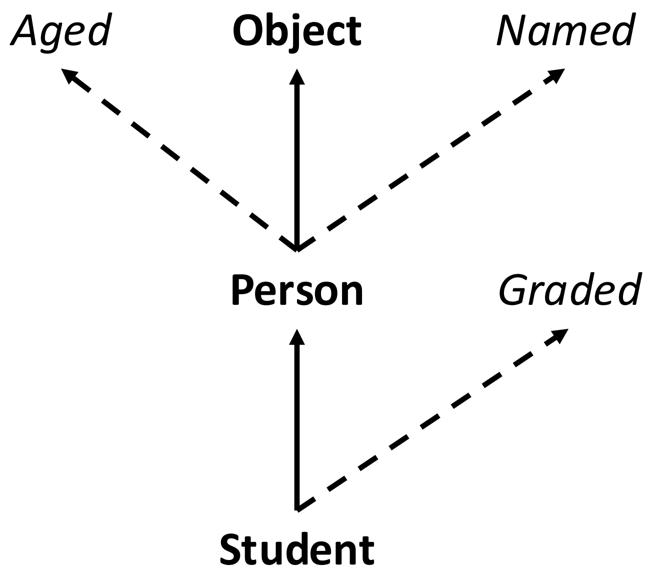
Immutability: summarize what you learned

. . .

`Object` and its Methods

Relationships

- Java only supports *single inheritance*
  - Only one superclass
  - Reserve for “is-a” relationship
- Classes may implement multiple interfaces
  - “Can-do” relationship



Object

- All classes are a subtype of Object
  - If no extends clause, then Object is the superclass
  - Interfaces implicitly must be implemented by an Object
- Object provides useful universal methods that you may want to override
  - `toString()`
  - `equals()`
  - `hashCode()`

toString() example: Point

```
public class Point {
  private final double x;
  private final double y;

  @Override
  public String toString() {
    return "(" + x + "," + y + ")";
  }
}
```

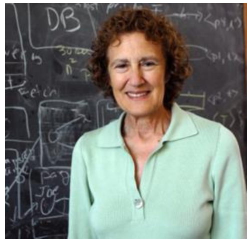
What would print out if point didn’t override toString()? (demo)

Equality

Referential equality (identity)

- Are two objects the same object?
- Test using ==
- Usually not desired

Logical equality (state)



- Should two objects be considered equivalent (substitutable)?
- Override `equals()` to define separately from identity
- Danger if class is mutable

Equivalence relations

- Reflexive
  - You equal yourself
  - x = x
- Symmetric
  - If you equal someone, they equal you
  - x = y if and only if y = x
- Transitive
  - If you equal someone and they equal someone else, you also equal that someone else
  - if x = y and y = z, then x = z

Note: Expressions on this slide such as “x=x” are math expressions, not code

Demo: Point equality

Overriding .equals()

```
@Override
public boolean equals(Object other) {
    if (!(other instanceof Point)) {
        return false;
    }
    Point p = (Point) other;
    return x == p.x && y == p.y;
}
```

Object and its methods: summarize what you learned

. . .

`record` classes

Record classes

```
public class Point {
  private final double x;
  private final double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double x() { return this.x; }
  public double y() { return this.y; }

  public String toString() { ... }

  public boolean equals ...
  public boolean hashCode...
}
```

Simple, standard code patterns like this are known as boilerplate code. How can we avoid writing boilerplate?

Record classes

```
~~public class Point {~~
~~  private final double x;~~
~~  private final double y;~~

~~  public Point(double x, double y) {~~
~~     this.x = x;~~
~~     this.y = y;~~
~~  }~~

~~  public double x() { return this.x; }~~
~~  public double y() { return this.y; }~~

~~  public String toString() { ... }~~

~~  public boolean equals ...~~
~~  public boolean hashCode...~~
~~}~~

public record Point(double x, double y) { }
```

Highlighted code is equivalent to crossed out code. Yay for conciseness!

Demo: Point record.
We can add methods to the Point record and override the defaults.

Metacognition

- Take 1 minute to write down a brief summary of what you have learned today

closing announcements to follow...

Announcements

- A4 grades published
- Prelim 1 on October 9 (in 2 weeks)
  - Practice exam coming soon
  - Make sure you’re on top of studying!