

Poll Everywhere

PollEv.com/javabear

text javabear to 22333



What will happen if we run:

```
Cat c = new AngryCat();  
c.pet();
```

```
public class Cat {  
    private int happiness;  
    public Cat() { happiness = 10; }  
    public void pet() { happiness += 2; respond(); }  
    protected void respond() { System.out.print("Purrr."); }  
}  
  
public class AngryCat extends Cat {  
    @Override  
    protected void respond() { System.out.print("Hiss!"); }  
}
```

Handwritten annotations:

- A blue circle around `AngryCat` in the code above the class definition, with a purple arrow pointing to it.
- A red underline under `respond()` in the `pet()` method of the `Cat` class, with the text `this.respond();` written in red above it.
- A blue arrow pointing from the `respond()` call in the `pet()` method to the `respond()` method in the `AngryCat` class, with the text `dynamic dispatch for new method call` written in blue next to it.

Compiler error (Line 2) **(A)**

Prints "Purrr." **(B)**

Prints "Hiss!" **(C)**

Prints "Hiss!Purrr." **(D)**

Exam Reminders

Prelim 1 is Tonight!

Early Exam: 5:30-7:00, Statler 196

Main Exam: 7:30-9:00, Statler 185

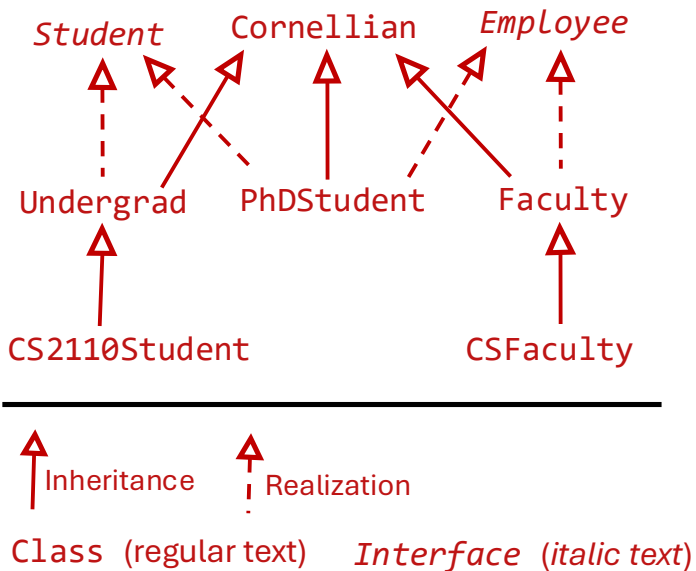
Bring your **Cornell ID Card** and a couple **writing utensils** (pencils, erasers, pens)
Exam is closed-book

More information and review materials linked on website / Ed

You've learned a lot so far! Time to show it off!

Type Hierarchies and Single Inheritance

Interfaces and inheritance let us build deep, complex type hierarchies



Java's single-inheritance rule:

Every* class has exactly one parent class (i.e., immediate superclass).

**choose wisely, this locks you into a rigid relationship*

A class can implement any number of interfaces

- interfaces simply promise existence of behaviors*
- often prefer interfaces to inheritance unless there's significant code reuse*



Lecture 11: Additional Java Features

CS 2110

September 29, 2026

Today's Learning Outcomes

- 21. Explain exceptions and their relationship to specifications and defensive programming.
- 22. Write code that throws, propagates, and handles exceptions.
- 46. Explain the concept of *immutability* and its relation to the final keyword in Java.
- 47. Contrast the == operator and the equals() method, and determine the appropriate one for a given scenario.

Part 1: Exception Handling

Exceptional Behavior

Sometimes, a program may encounter a situation that prevents it from proceeding with its normal control flow.

- "bad values": divide by 0, invalid array index, call method on null
- "bad data": missing file, improperly formatted file / text input
- "bad execution state": ran out of memory space on stack/heap

What should we do?

- proceed as normal? probably not, some post-conditions won't be met
- crash the program? sometimes this is only option
- alert the client to see what they suggest

Exception Handling in Java

An **exception** is an object that signals that a program has encountered an atypical circumstance.

Java has special language features and syntax for working with exceptions.

1. throw statements
(implementer)

produce exceptions

2. throws clauses
(specs)

alert compiler to
possibility of
exceptions

3. try/catch blocks
(client)

detect and
(possibly) handle
exceptions

The throw Statement

When you detect a situation that your code is unable to handle, use a throw statement to generate an exception.

```
throw new <constructor call to Throwable subtype>;
```

throw statements stop normal control flow and **propagate** down the runtime stack until they are **caught** (by try/catch block)

- kinda like alternative to return, we immediately leave method
- program crashes if exception propagates out of main()

since exceptions affect client's view (contract) of method, they should be documented in specs.

Poll Everywhere

PollEv.com/javabear

text javabear to 22333



```
static void f(int x) {  
    System.out.print(g(x + h(x)));  
}  
  
static int g(int x) {  
    System.out.print("A");  
    if (x > 5) { throw new IllegalStateException(); }  
    System.out.print("B");  
    return x + 1;  
}  
  
static int h(int x) {  
    System.out.print("C");  
    if (x > 5) { throw new IllegalStateException(); }  
    System.out.print("D");  
    return x + 2;  
}
```

Handwritten annotations:

- 3 above x in f(x + h(x))
- 5 above h(x)
- propagate (with arrow pointing from h(x) to f)
- evaluate g call before (with arrow pointing to g call)
- thrown (with arrow pointing to throw statement in g)
- 3 above x in h(x)
- skip (with arrow pointing to throw statement in h)

What is printed when we call f(3)?

A

(A)

ACD

(B)

CDA

(C)

CDAB9

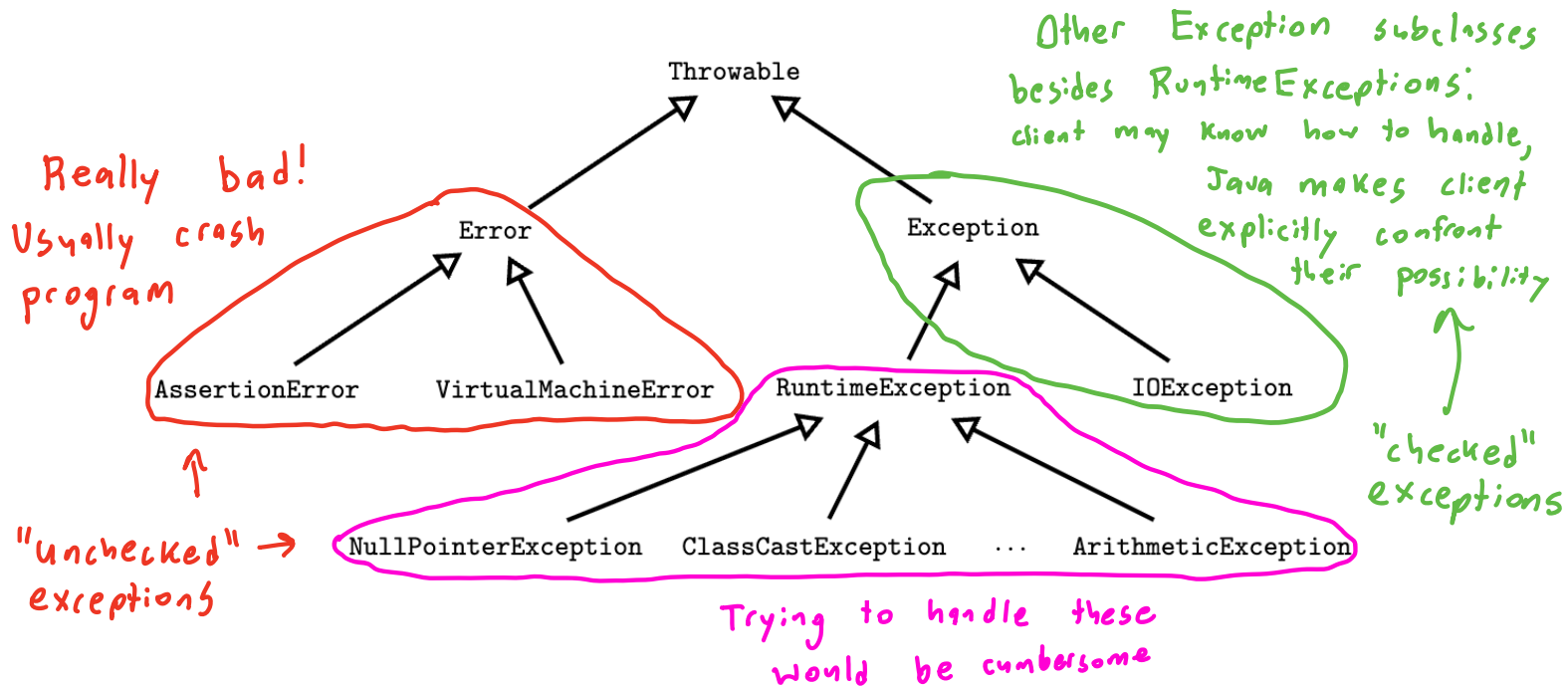
(D)



Coding Demo: Custom Exceptions



The Throwable Hierarchy



Checked Exceptions and throws Clauses

If a method has the possibility of *propagating* (i.e., either throwing itself or passing along from its own call) a *checked* exception, it must disclose this in a throws clause attached to the method signature.

```
static int findLocalMax(int[] a) throws NoLocalMaxException { }
```

Proper disclosure of exceptions is a property checked statically by compiler

- Good informational tool for client

try/catch Blocks

New syntax:

```
try {  
    // code that could propagate exception  
}  
catch ( <Exception Type> <Exception variable Name> ) {  
    // code to handle exception  
}  
catch ( <Exception Type> <Exception variable Name> ) {  
    // code to handle exception  
}
```

When an exception propagates into try block, its dynamic type is matched against catch parameters.

- Enter first (and only first) matching block

catch blocks stop exception propagation

Poll Everywhere

PollEv.com/javabear

text javabear to 22333



Suppose `angry()` throws a `NullPointerException` (a subtype of `RuntimeException`). What will be printed when we run `causeAnger()`?

```
static void causeAnger() {  
    try {  
        angry();  
    } catch (RuntimeException e) { matches this  
        System.out.print("A");  
        throw new IllegalStateException();  
    } catch (NullPointerException e) { propagates  
        System.out.print("B"); out of  
        throw new IllegalStateException(); method  
    } catch (IllegalStateException e) {  
        System.out.print("C");  
    }  
}
```

A

A

B

(B)

AB

(C)

AC

(D)

Exceptions and Specifications

Subtle interplay between these (frequent source of misconceptions)

1. Undefined Behavior may or may not result in an exception.

- defensive programming does turn pre-condition violations into `AssertionErrors`
- don't throw exceptions yourself that aren't documented in specs

2. Exceptions are not undefined behavior.

- included in specs
- offer "alternative post-condition" for method
- must restore class invariant before throwing

Since exceptions are part of specs, we write unit tests that check for them:

`JUnit's assertThrows()` method

* there's some subtlety to this syntax
that we'll return to later in the course

Part 2: Immutability and final

Immutability

A class is **mutable** if modifications can be made to its objects' state after their construction. Otherwise, the class is **immutable**.

Mutability opens door to bugs

- class invariant can be invalidated
- (later) issues in concurrent code

Prefer immutability where possible

Java classes don't have mechanisms to natively enforce immutability

Left up to implementer to enforce with good discipline
and detailed specs



Coding Demo: The Point Class



The final Keyword

Recall: final methods cannot be overridden in subclasses.

New: final fields cannot be reassigned after their initialization (in the class constructor)

- property is statically enforced by the compiler
- helps avoid "accidental mutation"
- can help optimize your code at runtime
- mark variables as final whenever possible (IntelliJ suggests it)

final does not fully enforce immutability.
(even if the variable can't be reassigned to reference a new object, the state of the original object can change)

Poll Everywhere

PollEv.com/javabear

text javabear to 22333



```
public class Point {  
    private final int[] coords;  
  
    public Point(int x, int y) {  
        coords = new int[]{x, y};  
    }  
  
    public int[] coords() { return coords; }  
  
    public static void main(String[] args) {  
        Point p = new Point(1,3);  
        p.coords()[0] = 2;  
    }  
}
```

*can't reassign int[] reference,
but can still update entries*

"rep exposure" 😞

What will happen when we (try to) compile and run this code?

Compiler Error (A)

Runtime Error (B)

Runs, coords = {1, 3} (C)

Runs, coords = {2, 3} (D)

Part 3: The Object Class

The super-est Class

The Object class is the top of Java's type hierarchy.

- Object is the default superclass of any class whose declaration does not include an extends clause.
- Thus, every Java object is an Object.

The Object class contains a few basic methods that are defined for every reference type.

Today: `toString()` : allows every object to be printed
 `equals()` : allows any pair of objects to be compared

 (more coming soon...)



Coding Demo: Overriding Object Methods



Java's Two Notions of Equality

`==` : Compares the **values** of the expressions on both sides

For reference types, "equality of reference", do the expressions hold references to the same object in memory

`equals()` : Allows objects to define a way to compare their **states**

- Any object should be able to determine if it considers itself equal to another object (perhaps at a different memory address)

specs require certain properties:

reflexivity, symmetry, transitivity (equivalence relation)

A Good equals() Template

```
@Override
```

```
public boolean equals(Object obj) {
```

```
    if ((obj == null) || (this.getClass() != obj.getClass())) {
```

```
        return false;
```

```
    }
```

```
    // replace this to a cast of your object type
```

```
    Point other = (Point) obj;
```

```
    // compare all fields (== for primitive types, equals() for reference types)
```

```
    return (this.x == other.x) && (this.y == other.y);
```

```
}
```

← we must be able to

compare to any type

check that type is the same

← get correct "view" of obj

* Never use instanceof in equals() definition,
this can violate symmetry property in specs.



Coding Demo: record Classes



Modeling an immutable type as a record can save us from writing a lot of "boilerplate" code.