Poll Everywhere

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

What will happen if we run:

```
AngryCat 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!"); }
}
```

AngryCat

this.respond();

dynamic dispatch for new method call

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

Runs and prints "Purrr." **(B)**

Runs and prints "Hiss!" **(C)**

Runs and prints "Hiss!Purrr." **(D)**

Dynamic Dispatch (with Inheritance)

With inheritance, definition of a class's method may not live in that class.

- It may be inherited from a superclass

Dynamic Dispatch: Every (non-super) instance method call starts at the dynamic type of its target and steps up class hierarchy until it finds a definition.

- "Bottom-up rule"
- Ensures the "most specialized" version of the method is always run
- Enables polymorphism

Lecture 11: Additional Java Features

CS 2110

February 26, 2026

Today's Learning Outcomes

23. Explain exceptions and their relationship to specifications and defensive programming.

24. Write code that throws, propagates, and handles exceptions.

48. Determine whether a class is mutable or immutable.

49. Explain the semantics of the final keyword.

50. Describe the semantic differences between the == operator and the equals() method and determine the appropriate one for a given scenario.

51. Identify the requirements of the equals() method specified in the Object class and override this method in user-defined classes.

Part 1: Exception Handling

Exceptional Behavior

Sometimes, our 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 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

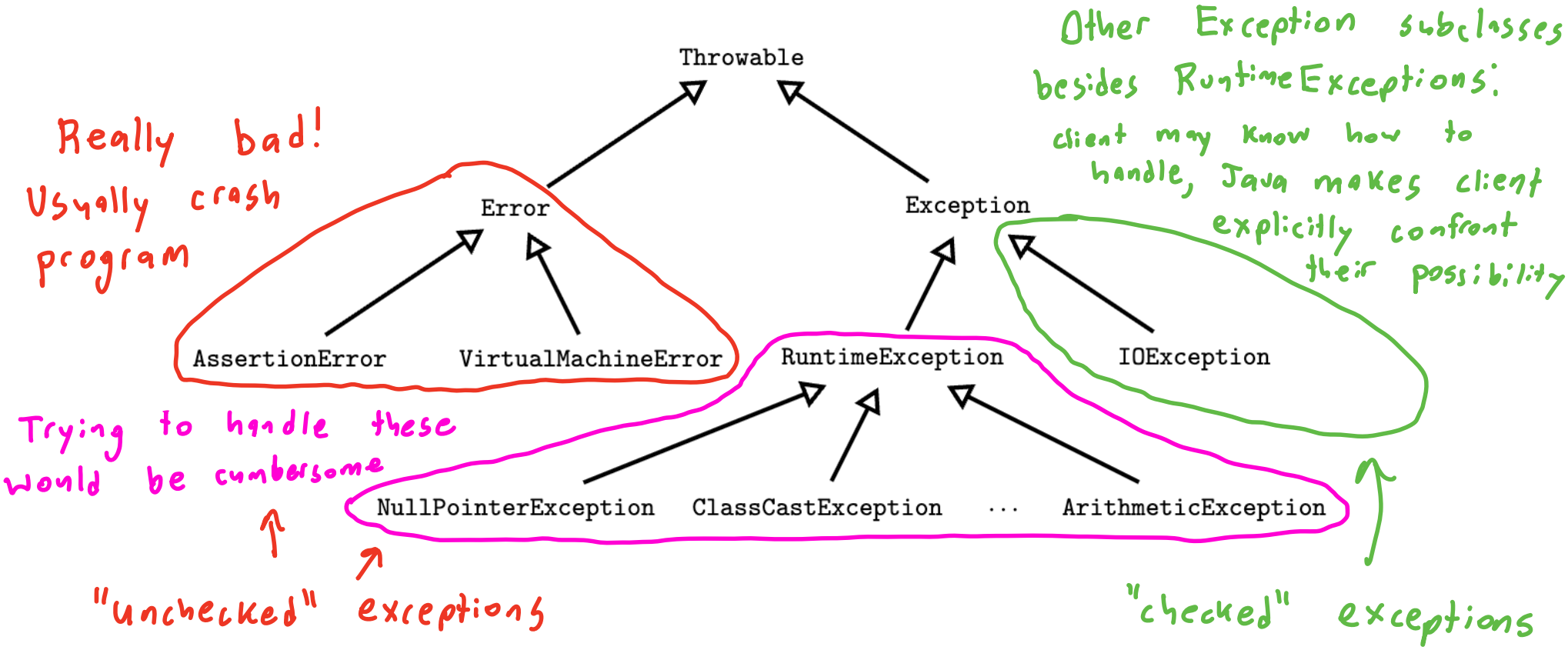
```
static void f(int x) {
   System.out.print(g(x + h(x)));
}

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;
}
```

What will be 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 (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) {
        System.out.print("A");
        throw new IllegalStateException();
    } catch (NullPointerException e) {
        System.out.print("B");
        throw new IllegalStateException();
    } catch (IllegalStateException e) {
        System.out.print("C");
    }
}
```

matches this

propagates out of method

A	(A)
B	(B)
AB	(C)
AC	(D)

Exceptions and Specifications

Subtle interplay (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 can write unit tests that check for them

JUnit assertThrows( )

* 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 it's 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 can't be overridden

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**

What will happen when we attempt to compile and run this code?

```
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;
   }
}
```

"rep exposure" ☹

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

- 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

- The default superclass of any class whose declaration does not include an "extends" clause
- 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 to come...)

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 compare their states
- Any object should be able to determine if it should consider 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.