Programming Languages and Techniques (CIS1200)

Lecture 21

Transition to Java

Chapters 19 & 20

Announcements

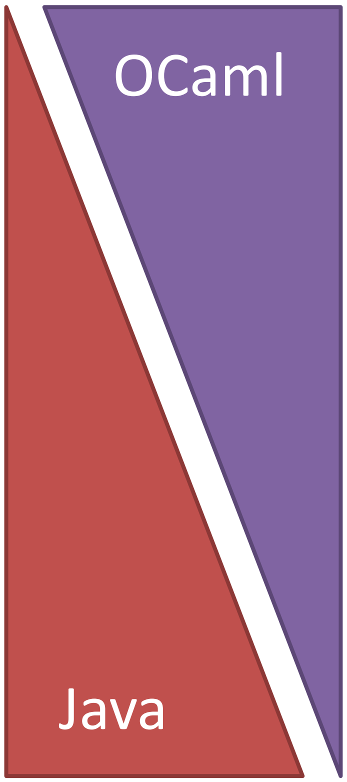
- HW05: GUI programming
  - Due: *Tuesday* at 11.59pm
- Java Bootcamp / Refresher: Wednesday, March 19
  - 7-9pm, Towne 100
  - Will be recorded
  - Look for more details on Ed
- HW06: Pennstagram
  - Java array programming
  - Available on course website
  - Due *Tuesday, March 25th*
- *Midterm 2: Friday, March 28th*
  - OCaml: ASM, mutability, queues/deques, closures, GUI, and Java basics

Goodbye OCaml…

…Hello Java!

CIS 1200 Semester Overview

- Declarative (Functional) programming
  - *persistent* data structures
  - *recursion* is main control structure
  - frequent use of functions as data
- Imperative programming
  - *mutable* data structures (that can be modified “in place”)
  - *iteration* is main control structure
- Object-oriented and reactive programming
  - mutable data structures / iteration
  - heavy use of functions (objects) as data
  - pervasive “abstraction by default”



Recap: The Functional Style

- Core ideas:
  - immutable (persistent / declarative) data structures
  - recursion (and iteration) over tree structured data
  - functions as data
  - generic types for flexibility (i.e. 'a list)
  - abstract types to preserve invariants (i.e. BSTs)
  - *simple model of computation (substitution)*
- Good for:
  - elegant descriptions of complex algorithms & data
  - compositional design
  - "symbol processing" programs (compilers, theorem provers, etc.)
  - reliable software / verification
  - parallelism, concurrency, and distribution

Other Popular Functional Languages

F#: Most similar to OCaml,
Shares libraries with C#

Haskell (CIS 5520)
Purity + laziness

Swift
iOS programming

Verse: Functional/Logic
language for unreal engine

Racket: LISP descendant;
widely used in education

Scala
Java / OCaml hybrid

Java and OCaml together

me

Guy Steele, one of the
principal designers of Java

Xavier Leroy, one of the principal
designers of OCaml

Moral: Java and OCaml are not so far apart…

Functional programming

OCaml

- Immutable lists primitive, tail recursion
- Datatypes and pattern matching for immutable tree structured data
- First-class functions, transform and fold
- Generic types
- Abstract types through module signatures

Java

- No primitive data structures, no tail recursion
- Trees must be encoded by objects, mutable by default, limited pattern matching*
- First-class functions less common**, objects instead
- Generic types***
- Abstract types through interfaces and public/private modifiers

*feature of Java 17 (released 2021)

**late addition, encoded from objects

***not completely "first class" (see, e.g., Arrays)

OCaml vs. Java for FP

OCaml

```
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)

let rec lookup (t:'a tree) (n:’a : bool =
  begin match t with
   | Empty -> false
   | Node(lt, x, rt) ->
        x = n ||
        if n < x then lookup lt n
       else lookup rt n
  end
```

OCaml provides a succinct, clean notation for working with generic, immutable, tree-structured data. Java requires more "boilerplate."

```
public abstract sealed class
 Tree<A extends Comparable<A>>
     permits Tree.Empty, Tree.Node {

 final static class
  Empty<A extends Comparable<A>> extends Tree<A> {}

 final static class
  Node<A extends Comparable<A>> extends Tree<A> {
   final A v;
   final Tree<A> lt;
   final Tree<A> rt;
   public Node(Tree<A> lt, A value, Tree<A> rt) {
    this.lt = lt; this.rt = rt; this.v = v;
   }

 public static <A extends Comparable<A>>
  boolean lookup(A x, Tree<A> t) {
   if (t instanceof Node<A> n) {
    return switch (x.compareTo(n.value)) {
      case -1 -> lookup(x, n.left);
      case 1 -> lookup(x, n.right);
      default -> n.value.equals(x);
    };
   } else {
    return false;
   }
  }
}
```

Java

Recap: The imperative style

- Core ideas:
  - computation as *change of state over time*
  - distinction between primitive and reference values
  - aliasing!
  - linked data-structures and iteration control structures
  - generic types for flexibility (i.e., ‘*a* queue)
  - abstract types to preserve invariants (i.e., queue invariant)
  - *Abstract Stack Machine model of computation*

interior of a pocket watch

- Good for:
  - high performance, low-level code
  - numerical simulations
  - implicit coordination between components (queues, GUI)
  - explicit interaction with hardware

Imperative programming

OCaml

- No null. Partiality must be made explicit with **options**.
- Code is an **expression** that has a value. Sometimes computing that value has other effects.
- References are **immutable** by default, must be explicitly declared to be mutable

Java

- Most types have a **null** element. Partial functions can return **null**.
- Code is a sequence of **statements** that have effects, sometimes using expressions to compute values.
- References are **mutable** by default, must be explicitly declared to be constant

Explicit vs. Implicit Partiality

OCaml identifiers

- Cannot be changed once created; only mutable fields can change

```
type 'a ref = { mutable contents: 'a }
let x = { contents = counter () }
;; x.contents <- counter ()
```

- Cannot be null, must use options

```
let y = { contents = Some (counter ())}
;; y.contents <- None
```

- Accessing option values requires pattern matching

```
;; begin match y.contents with
     | None -> failwith "NPE"
     | Some c ->  c.inc ()
   end
```

Java variables

- Can be assigned to after initialization

```
Counter x = new Counter ();
x = new Counter ();
```

- Can always be null

```
Counter y = new Counter ();
y = null;
```

- Check for null is implicit whenever a variable is used

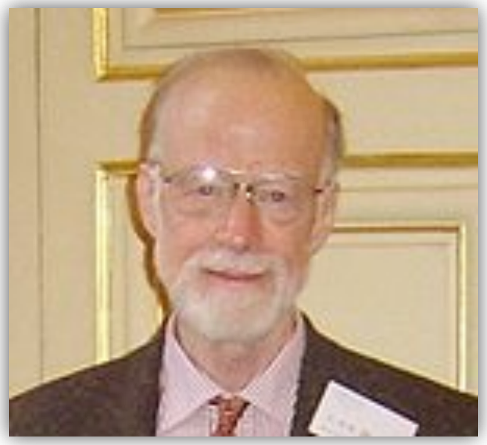
```
y.inc();
```

- If null is used as an object (i.e. for a method call) then a **NullPointerException** occurs

The Billion Dollar Mistake

"***I call it my billion-dollar mistake. It was the invention of the null reference in 1965.** … **This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years**. "*

Sir Tony Hoare, London 2009



Smoothing the transition to Java

- General advice for the next few lectures:
  - Ask questions, but don’t stress about the details
  - Wait till you need them
- Java resources:
  - Our lecture notes
  - Ed and Java Bootcamp
  - CIS 1100 website and textbook
  - Online Java textbooks (http://math.hws.edu/javanotes/) linked from “Resources” on course website

Java Core Language

differences between OCaml and Java

Structure of a Program

OCaml

- All code lives in (perhaps implicitly named) **modules**.
- Modules may import other modules and may contain multiple **type definitions**, **let**-bound value **declarations**, and top-level **expressions**.
- The program starts running at the beginning of a module and executes the definitions in the order that they are encountered.

Java

- All code lives in explicitly named **classes**.
- Classes are types (of objects).
- Classes contain **field declarations** and **method definitions**.
- There is a single "entry point" of the program where it starts running, which must be a method called `main`.

Expressions vs. Statements

- OCaml is an *expression language*
  - Every program phrase is an expression (and returns a value)
  - The special value () of type `unit` is used as the result of expressions that are evaluated only for their side effects
  - Semicolon is an *operator* that combines two expressions (where the left-hand one returns type unit)
- Java is a *statement language*
  - Two-sorts of program phrases: expressions (which compute values) and statements (which don't)
  - Statements are *terminated* by semicolons
  - Any expression can be used as a statement (but not vice-versa)
  - Some statements have expression variants (if, case)

Types

- As in OCaml, every Java *expression* has a type
- The type describes the value that an expression computes

Expression form	**Example**	**Type**
Variable reference	x	Declared type of variable
Operator use	5 + x	Result type of operation
Object creation	new Counter ()	Class of the object
Method call	c.inc()	Return type of method
Equality test	x == y	boolean
Assignment	x = 5	don't use as an expression!!

Type System Comparison

	OCaml	**Java**
primitive types (values stored "directly" in the stack)	int, float, char, bool, …	int, float, double, char, boolean, …
structured types (a.k.a. *reference types* — values stored in the heap)	tuples, datatypes, records, functions, arrays (*objects encoded as records of functions*)	objects, arrays (*records, tuples, datatypes, strings, first-class functions are special cases of objects*)
generics	'a list	List<A>
abstract types	module types (signatures)	interfaces, abstract classes, public/private modifiers

Arithmetic & Logical Operators

OCaml	Java	
=, ==	==	equality test
<>, !=	!=	inequality
>, >=, <, <=	>, >=, <, <=	comparisons
+	+	addition
^	+	string concatenation
-	-	subtraction (and unary minus)
*	*	multiplication
/	/	division
mod	%	remainder (modulus)
not	!	logical “not”
&&	&&	logical “and” (short-circuiting)
||	||	logical “or” (short-circuiting)

New in Java: Operator Overloading

- The *meaning* of an operator in Java is determined by the *types* of the values it operates on:
  - Integer division

    `4/3` ⇒ `1`
  - Floating point division

    `4.0/3.0` ⇒ `1.3333333333333333`
  - Automatic conversion from int to float, then float division

    `4/3.0` ⇒ `1.3333333333333333`

- Method *overloading* is a general mechanism in Java
  - we'll see more of it later

Equality

- like OCaml, Java has two ways of testing reference types for equality:
  - "reference equality"

    `o1 == o2`
  - "deep equality"

    `o1.equals(o2)`

every object provides an "equals" method that should "do the right thing" depending on the class of the object

- Normally, you should use == to compare primitive types and "`.equals`" to compare objects
- Careful: Single-equals (=) means assignment, not equality comparison

Strings: immutable reference type

- `String` is a *built in* Java class
- Strings are sequences of (unicode) characters

  `""` `"Java"` `"3 Stooges"` `"富士山"`

- + means String concatenation (overloaded)

  `"3" + " " + "Stooges"` ⇒ `"3 Stooges"`

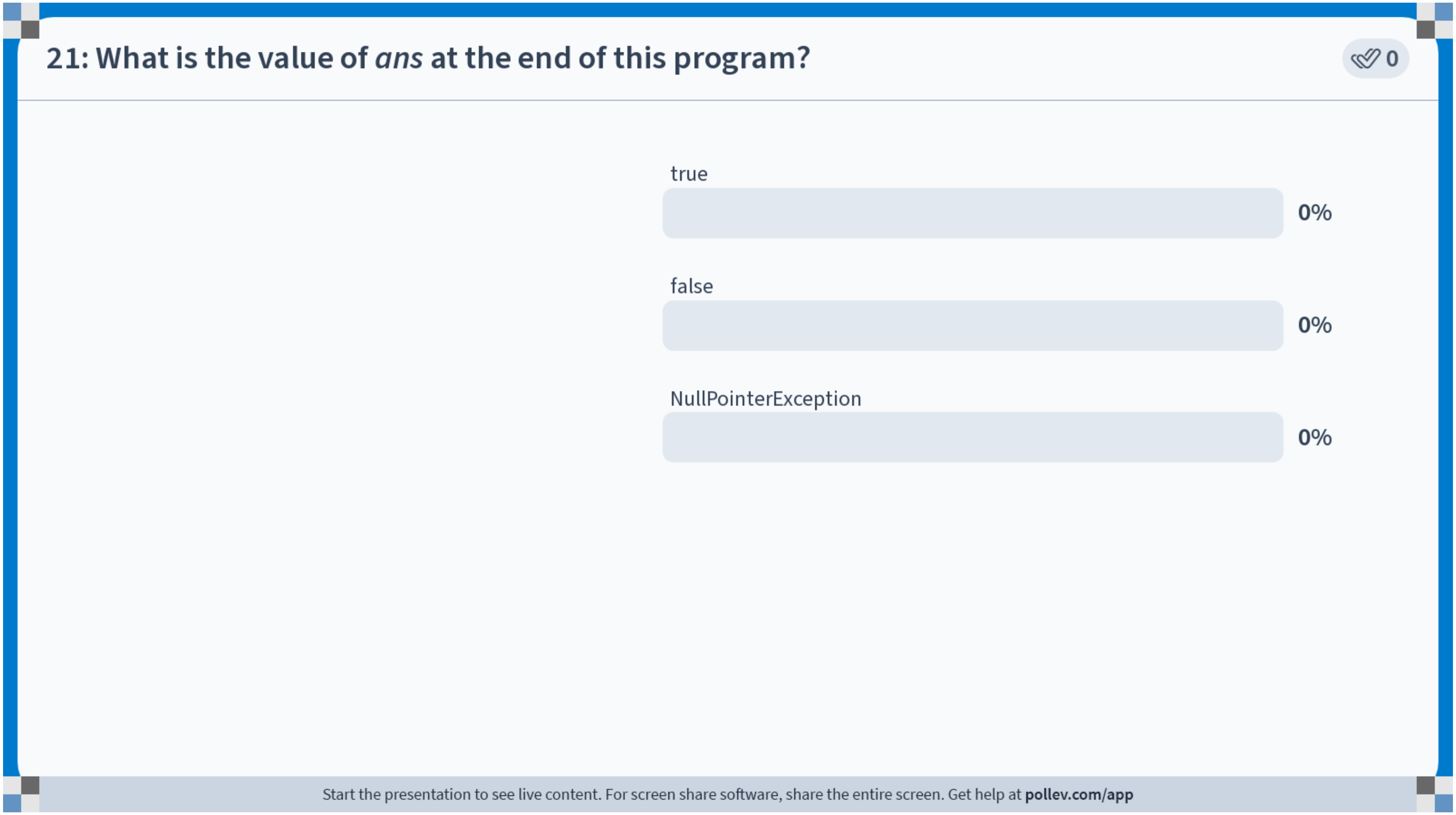
- Text in a String is immutable (like OCaml)
  - but variables that store strings are not
  - `String x = "OCaml";`
  - `String y = x;`
  - Immutability: can't do anything to `x` so that `y` changes
- The `.equals` method returns true when two strings contain the *same* sequence of characters

Aside: StringBuffers

- `StringBuffer` is a *mutable* Java String
- Alternative to "+" when constructing large strings

```
String s = "Hello";
for (int i=0; i<200; i++) {
  s = s + "!";
}
```

```
StringBuffer sb = new StringBuffer("Hello");
for (int i=0; i<200; i++) {
  sb.append("!");  // modify end of sb
}
String s = sb.toString();  // convert back to String
```



21: What is the value of *ans* at the end of this program?

✓ 0

true

0%

false

0%

NullPointerException

0%

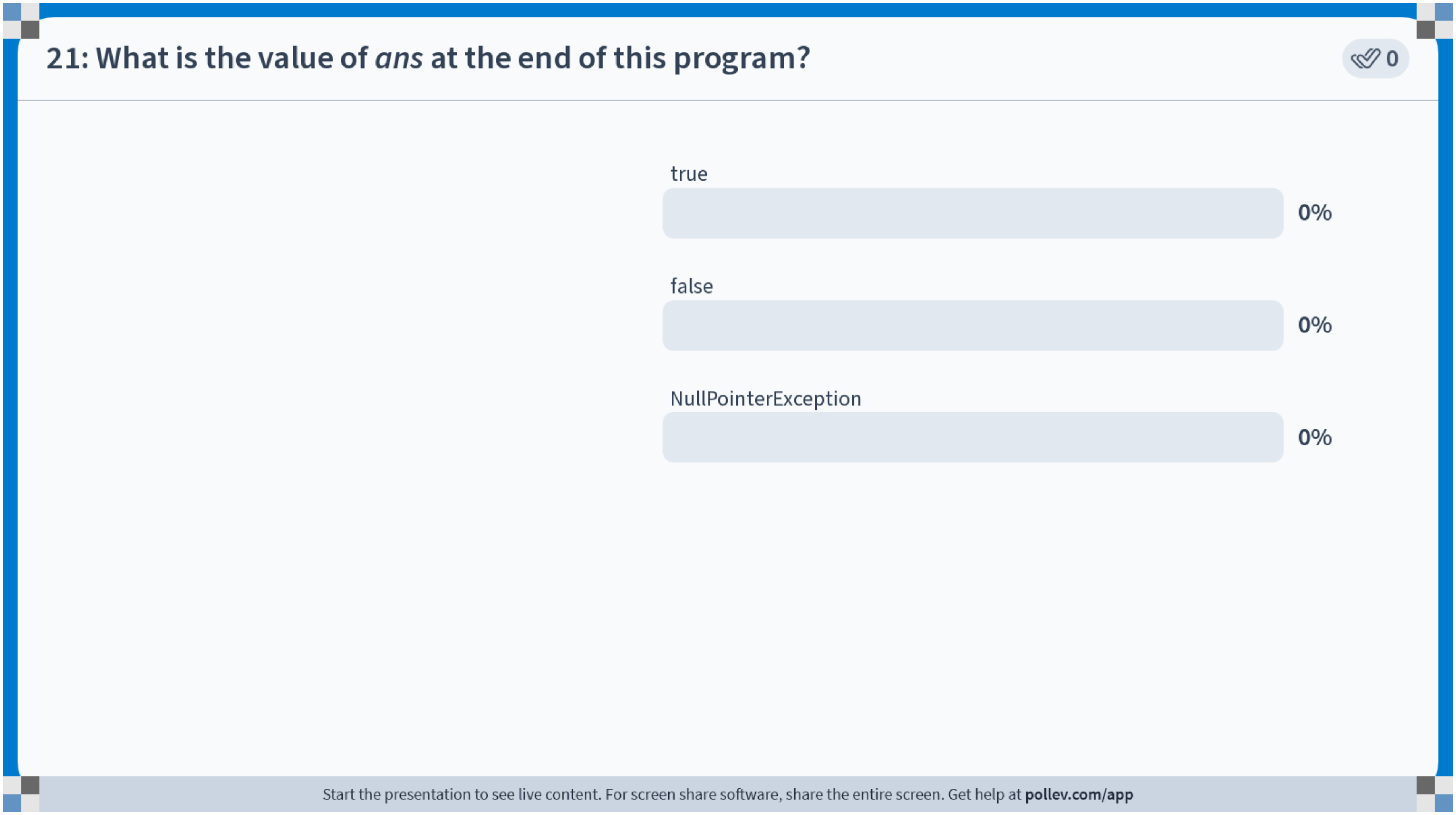
Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

What is the value of ans at the end of this program?

```
String x = "CIS 1200";
String z = "CIS 1200" ;
boolean ans = x.equals(z);
```

1. true
2. false
3. NullPointerException

Answer: true
This is the preferred method of comparing strings!



21: What is the value of *ans* at the end of this program?

✓ 0

true 0%

false 0%

NullPointerException 0%

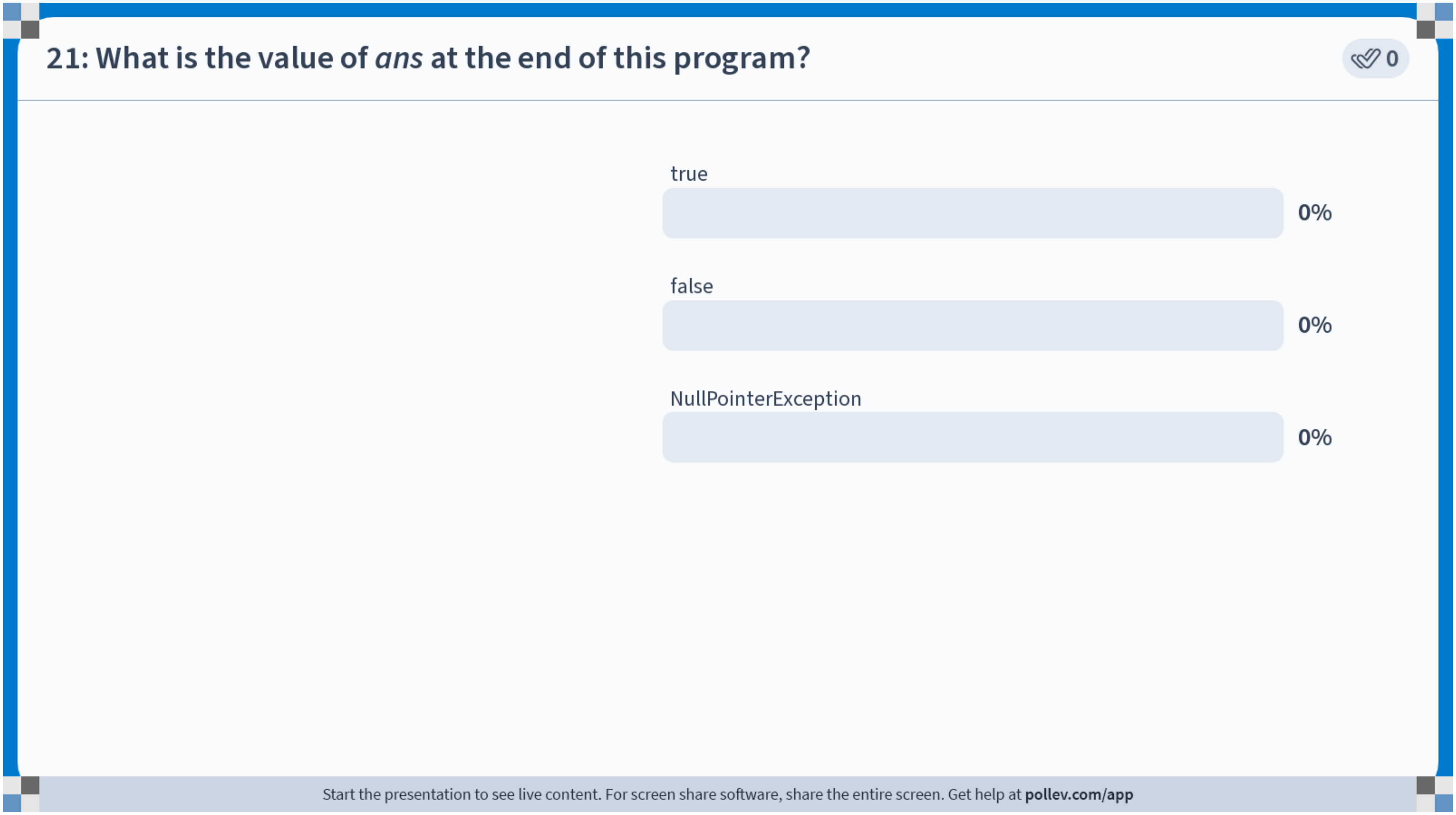
Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

What is the value of ans at the end of this program?

```
String x1 = "CIS ";
String x2 = "1200";
String x = x1 + x2;
String z = "CIS 1200";
boolean ans = (x == z);
```

1. `true`
2. `false`
3. `NullPointerException`

Answer: false
Even though x and z both contain the characters “CIS 1200”,
they are stored in two different locations in the heap.



21: What is the value of *ans* at the end of this program?

✓ 0

true

0%

false

0%

NullPointerException

0%

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

What is the value of ans at the end of this program?

```
String x = "CIS 1200";
String z = "CIS 1200";
boolean ans = (x == z);
```

```
1. true
2. false
3. NullPointerException
```

Answer: true(!)
Why? Since strings are immutable, two identical
strings that are known when the program is compiled can be aliased by the compiler (to save space).

Moral

Always use s1.equals(s2) to compare Strings!

Compare strings with respect to their content, not where they happen to be allocated in memory…

Object Oriented Programming

Preview: The OO Style

- Core ideas:
  - objects (state encapsulated with operations)
  - dynamic dispatch ("receiver" of method call determines behavior)
  - classes ("templates" for object creation)
  - subtyping (grouping object types by common functionality)
  - inheritance (creating new classes from existing ones)
- Good for:
  - GUIs
    - complex software systems that include many different implementations of the same "interface" (set of operations) with different behaviors
  - Simulations
    - designs with an explicit correspondence between "objects" in the computer and things in the real world
  - Games

encapsulated state

"Objects" in OCaml

```
(* The type of counter objects *)
type counter = {
    inc  : unit -> int;
    dec  : unit -> int;
}

(* Create a counter “object” *)
let new_counter () : counter =
  let r = {contents = 0} in
  {
    inc = (fun () ->
      r.contents <- r.contents + 1;
      r.contents);
    dec = (fun () ->
      r.contents <- r.contents - 1;
      r.contents)
 }
```

Why is this an object?

- *Encapsulated local state* only visible to the methods of the object
- Object is *defined by what it can do*—local state does not appear in the interface
- There is a way to *construct* new object values that behave similarly

OO terminology

- *Object*: a structured collection of encapsulated *fields* (aka *instance variables*) and *methods*
- *Class*: a template for creating objects
- The class of an object specifies…
  - the types and initial values of its local state (fields)
  - the set of operations that can be performed on the object (methods)
  - one or more *constructors*: create new objects by (1) allocating heap space, and (2) running code to initialize the object (optional, but default provided)
- Every (Java) object is an *instance* of some class
  - Instances are created by invoking a constructor with the new keyword

OO programming

OCaml **(part we've seen)**

- Explicitly create objects using a record of higher order functions and hidden state

- Flexibility through ***composition***: objects can only implement one interface

```
type button =
    widget *
    label_controller *
     notifier_controller
```

Java **(and Python, C, C++, C#)**

- Primitive notion of object creation (classes, with fields, methods and constructors)

- Flexibility through ***extension***: ***Subtyping*** allows related objects to share a common interface

```
class Button extends Widget {
  /* Button is a subtype
     of Widget */

}
```

Objects in Java

class declaration

```java
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

class name

instance variable

constructor

methods

object creation and use

```java
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter();

      System.out.println( c.inc() );

   }
}
```

constructor invocation

method call

Encapsulating local state

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

r is *private*

constructor and methods can refer to r

other parts of the program can only access public members

```
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter();

      System.out.println( c.inc() );

   }
}
```

method call

Encapsulating local state

- *Visibility modifiers* make the state local by controlling access
- Basically*:
  - public : accessible from anywhere in the program
  - private : only accessible inside the class
- Design pattern — first cut:
  - Make *all* fields private
  - Make constructors and non-helper methods public

*Java offers a couple of other protection levels — “protected” and “package protected” for structure larger code developments and libraries. The details are not important at this point.

Constructors with Parameters

```
public class Counter {

  private int r;

  public Counter (int r0) {
    r = r0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

  public int dec () {
    r = r - 1;
    return r;
  }
}
```

Constructor methods can take parameters

Constructor must have the same name as the class

object creation and use

```
public class Main {

 public static void
    main (String[] args) {

      Counter c = new Counter(3);

      System.out.println( c.inc() );

   }
}
```

constructor invocation

Creating Objects

- *Declare* a variable to hold a `Counter` object
  - Type of the object is the *name* of the class that creates it
- *Invoke* the constructor for `Counter` to create a `Counter` instance with keyword "new" and store it in the variable

```
Counter c = new Counter();
```

Creating Objects

- Every Java variable is mutable

```
Counter c = new Counter(2);
c = new Counter(4);
```

- A Java variable of *reference* type can also contain the special value “null”

```
Counter c = null;
```

☞ Remember!
Single = for assignment
Double == for reference equality testing

What is the value of ans at the end of this program?

```
Counter x;
x.inc();
int ans = x.inc();
```

1. 1
2. 2
3. 3
4. Raises `NullPointerException`

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

}
```

Answer: NullPointerException

What is the value of ans at the end of this program?

```
Counter x = new Counter();
x.inc();
Counter y = x;
y.inc();
int ans = x.inc();
```

```
public class Counter {

  private int r;

  public Counter () {
    r = 0;
  }

  public int inc () {
    r = r + 1;
    return r;
  }

}
```

1. 1
2. 2
3. 3
4. NullPointerException

Answer: 3