Programming Languages and Techniques (CIS120)

Lecture 27

Generics and Collections

Chapters 25 and 26

Announcements

- Java Programming (Pennstagram)
  - Tuesday, November 5 at 11:59:59pm

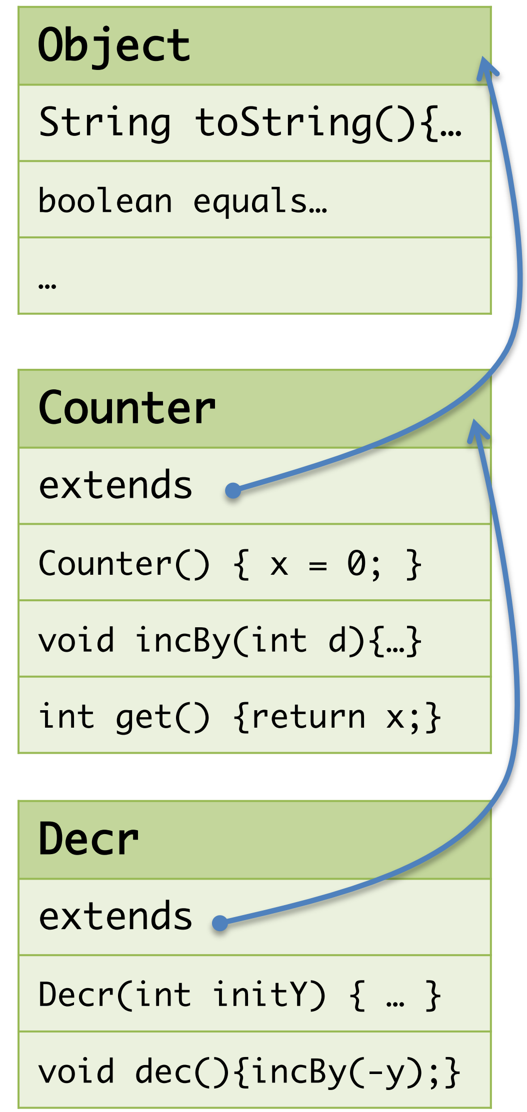
- Upcoming: Midterm 2
  - Friday, November 8th in class
  - Coverage: mutable state, queues, deques, GUI, Java material up to Friday (simple inheritance, "this")
  - Chapters 11-24
- Exam Logistics:
  - Last Names A – M   go to Leidy Labs 10 (here)
  - Last Names N – Z   go to College Hall 200   (COLL 200)

- Midterm Review Session:
  - Wednesday, November 6th 6:00-8:00pm in Towne 100
  - RSVP on Piazza

ASM refinement: The Class Table

```
public class Counter {
  private int x;
  public Counter () { x = 0; }
  public void incBy(int d) { x = x + d; }
  public int get() { return x; }
}

public class Decr extends Counter {
  private int y;
  public Decr (int initY) { y = initY; }
  public void dec() { incBy(-y); }
}
```

Class Table



The class table contains:

- the code for each method,
- references to each class's parent, and
- the class's static members.

Static members and the Java ASM

Static Members

- Classes in Java can also act as *containers* for code and data.
- The modifier `static` means that the field or method is associated with the class and *not* instances of the class.

You can do a static assignment to initialize a static field.

```
class C {
  public static int x = 23;
  public static int someMethod(int y) { return C.x + y; }
  public static void main(String args[]) {
    …
  }
}

 // Elsewhere:
C.x = C.x + 1;
C.someMethod(17);
```

Access to the static member uses the class name `C.x` or `C.foo()`

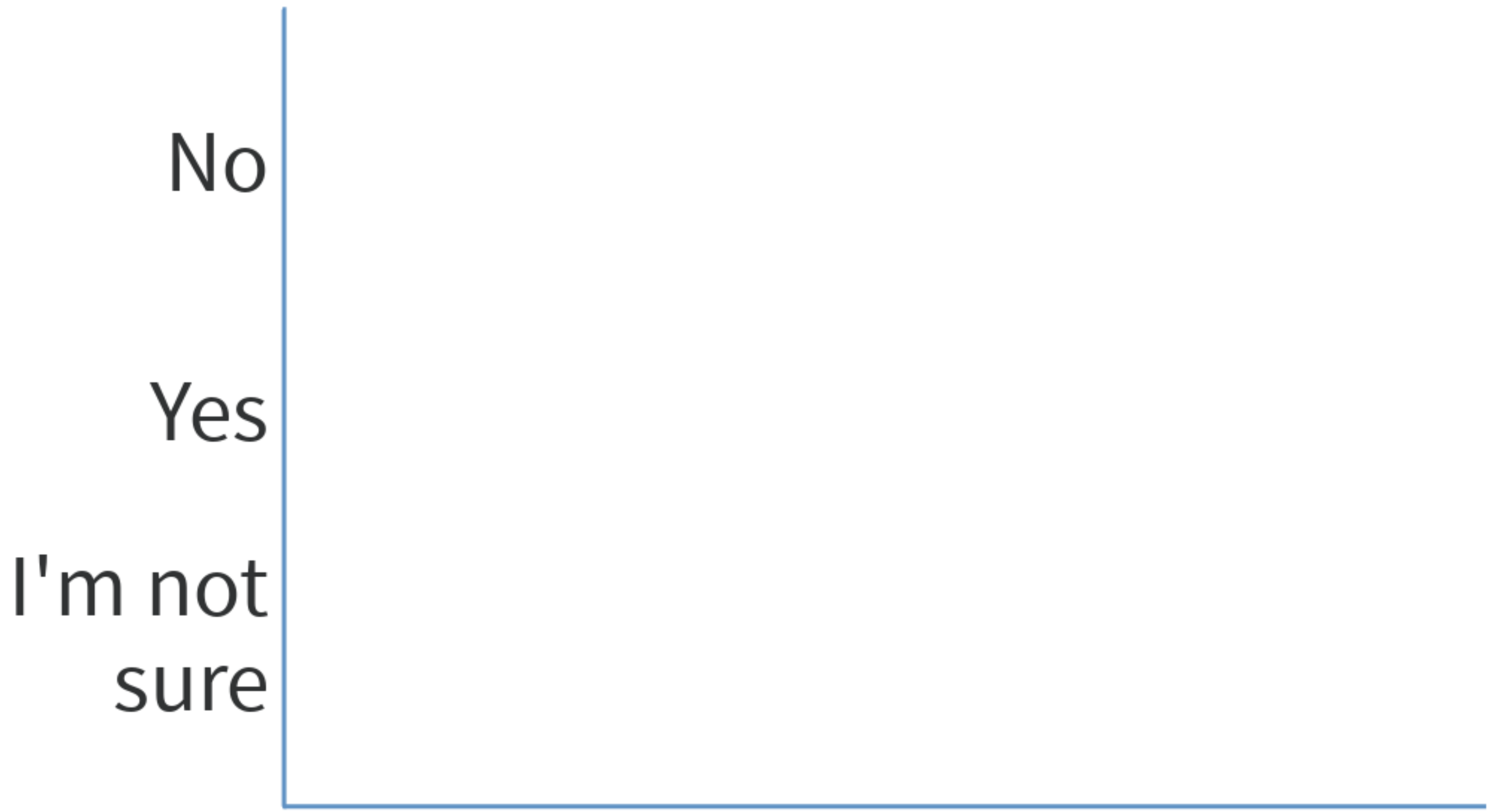
Class Table Associated with C

- The class table entry for `C` has a field slot for `x`.
- Updates to `C.x` modify the contents of this slot: `C.x = 17;`

C	
`extends Object`	
`static x`	23
`static int someMethod(int y) { return x + y; }`	
`static void main(String args[]) {…}`	

- A static field is a *global* variable
  - There is only one heap location for it (in the class table)
  - Modifications to such a field are visible everywhere the field is
    - if the field is public, this means *everywhere*
  - Use with care!

Based on your understanding of *this*, is it possible to refer to *this* in a static method?



Static Methods (Details)

- Static methods do *not* have access to a `this` pointer
  - Why? There isn't an instance to dispatch through!
  - Therefore, static methods may only directly call other static methods.
  - Similarly, static methods can only directly read/write static fields.
  - Of course a static method can create instance of objects (via `new`) and then invoke methods on those objects.

- Gotcha: It is possible (but confusing) to invoke a static method as though it belongs to an object instance.
  - e.g. `o.someMethod(17)` where `someMethod` is static

Java Generics

Subtype Polymorphism

vs.

Parametric Polymorphism

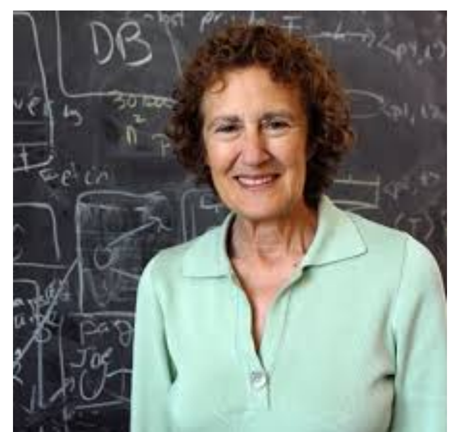
Review: Subtype Polymorphism*

- Main idea:

> Anywhere an object of type A is needed, an object that is a subtype of A can be provided.

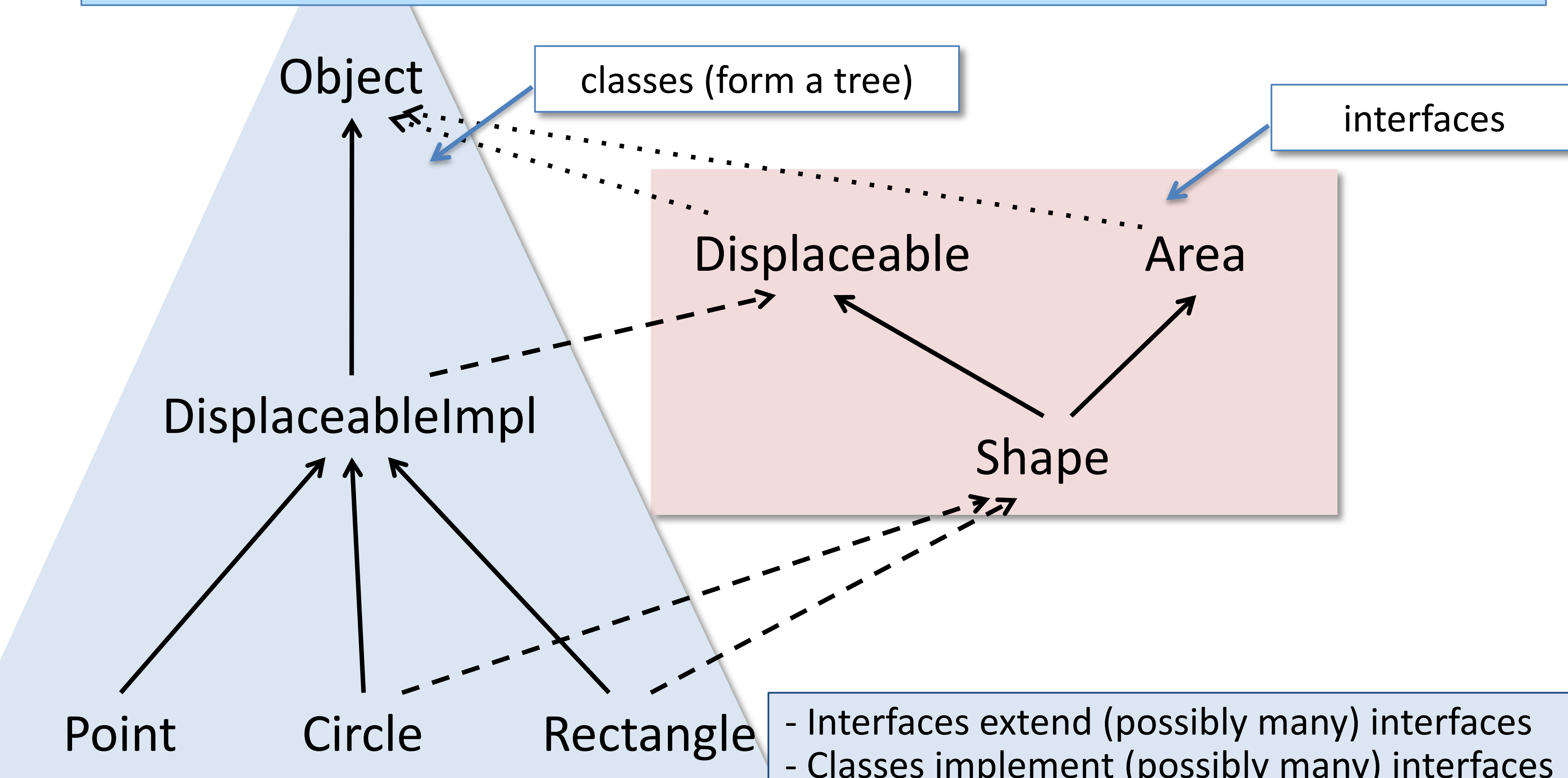
- If B is a subtype of A, it provides all of A's (public) methods.

- More generally: a B should "behave the same" as an A
  - see: *Liskov Substitution Principle*
  - named for Turing Award winner MIT Professor, Barbara Liskov

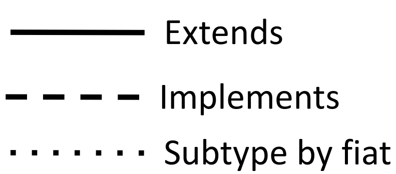


*polymorphism = many shapes

Recap: Subtyping



- Interfaces extend (possibly many) interfaces
- Classes implement (possibly many) interfaces
- Classes (except Object) extend exactly one other class (Object by default)
- Interface types (and arrays) are subtypes "by fiat" of Object



Is subtype polymorphism enough?

Mutable Queue Interface in OCaml

```
module type QUEUE =
sig
  (* type of the data structure *)
  type 'a queue

  (* Make a new, empty queue *)
  val create : unit -> 'a queue

  (* Add a value to the end of the queue *)
  val enq : 'a -> 'a queue -> unit

  (* Remove the front value and return it (if any) *)
  val deq : 'a queue -> ‘a

  (* Determine if the queue is empty *)
  val is_empty : ‘a queue -> bool
end
```

How can we translate this interface to Java?

Java Interface using Subtyping

```ocaml
module type QUEUE =
sig

  type 'a queue

  val create : unit -> 'a queue

  val enq : 'a -> 'a queue -> unit

  val deq : 'a queue -> 'a

  val is_empty : 'a queue -> bool

end
```

OCaml

```java
interface ObjQueue {



   // no constructors
   // in an interface

    public void enq(Object elt);

    public Object deq();

    public boolean isEmpty();

}
```

Java

Subtype Polymorphism

```
interface ObjQueue {
   public void enq(Object elt);
   public Object deq();
   public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
__A__  x = q.deq();
```

What type can we write for A?

1. String
2. Object
3. ObjQueue
4. None of the above

ANSWER: Object

Subtype Polymorphism

```
interface ObjQueue {
   public void enq(Object elt);
   public Object deq();
   public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
Object  x = q.deq();
System.out.println(x.trim());
```

trim is a method of the String class (removes extra spaces)

← Does this line type check

1. Yes
2. No
3. It depends

ANSWER: No

Subtype Polymorphism

```
interface ObjQueue {
   public void enq(Object elt);
   public Object deq();
   public boolean isEmpty();
}
```

```
ObjQueue q = …;

q.enq(" CIS 120 ");
Object  x = q.deq();
//System.out.println(x.trim());
q.enq(new Point(0.0,0.0));
___B___  y = q.deq();
```

What type for B?

1. Point
2. Object
3. ObjQueue
4. None of the above

ANSWER: Object

Parametric Polymorphism (a.k.a. Generics)

- Main idea:

Parameterize a type (i.e. interface or class) by another type.

```
public interface Queue<E> {
  public void enq(E o);
  public E deq();
  public boolean isEmpty();
}
```

- The implementation of a parametric polymorphic interface cannot depend on the implementation details of the parameter.
  - the implementation of enq cannot invoke any methods on 'o' (except those inherited from Object)
  - i.e. the only thing we know about E is that it is a subtype of Object

Generics (Parametric Polymorphism)

```
public interface Queue<E> {
  public void enq(E o);
  public E deq();
  public boolean isEmpty();
  …
}
```

```
Queue<String> q = …;

q.enq(" CIS 120 ");
String x = q.deq();                  // What type of x?    String
System.out.println(x.trim());        // Is this valid?     Yes!
q.enq(new Point(0.0,0.0));           // Is this valid?     No!
```

Subtyping and Generics

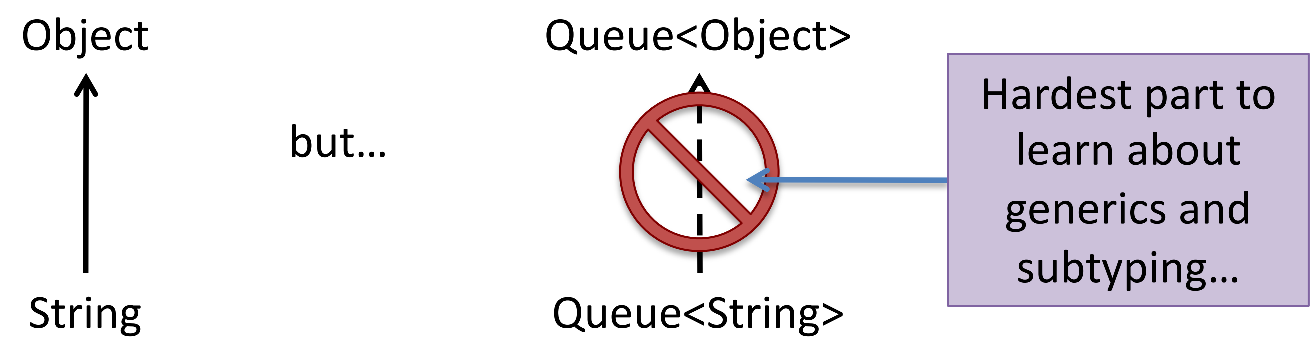
Subtyping and Generics*

```
Queue<String> qs = new QueueImpl<String>();
Queue<Object> qo = qs;

qo.enq(new Object());
String s = qs.deq();
```

Ok? Sure!
Ok? Let's see…

Ok? I guess
Ok? Noooo!

- Java generics are *invariant*:
  - Subtyping of *arguments* to generic types does not imply subtyping between the instantiations:

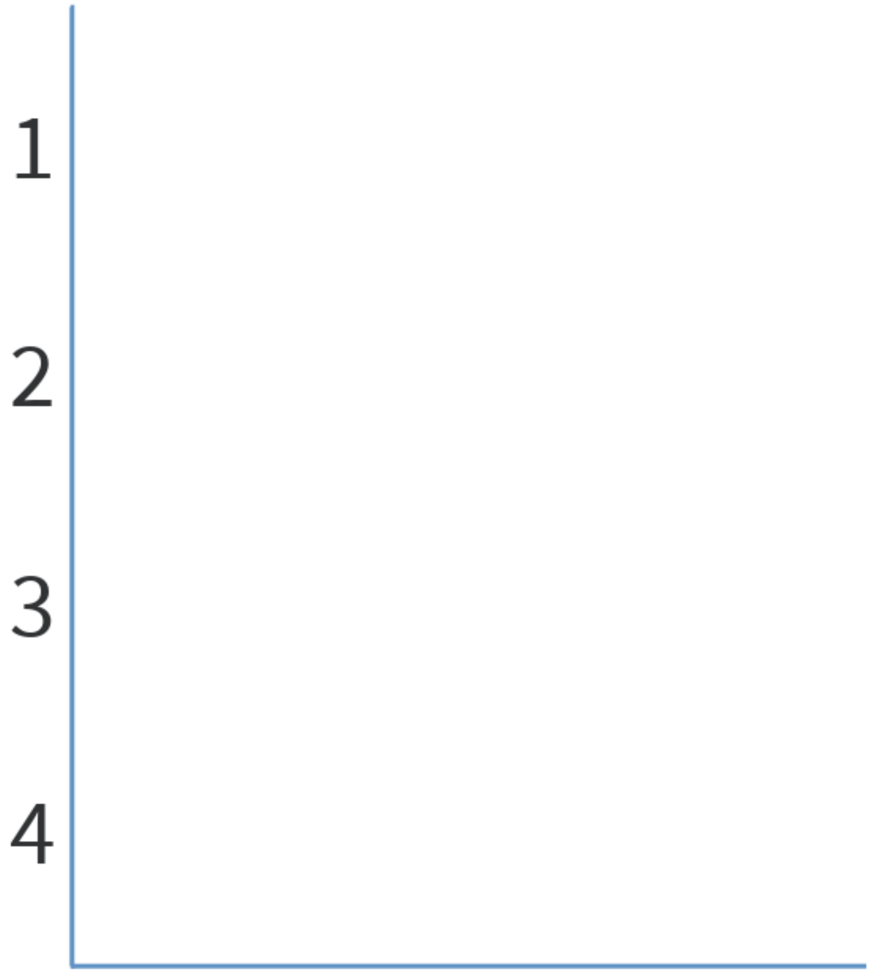


* Subtyping and generics interact in other ways too. Java supports *bounded polymorphism* and *wildcard types*, but those are beyond the scope of CIS 120.

Subtyping with Generics

Which of these are true, assuming that class QueueImpl<E> implements interface Queue<E>?

1. QueueImpl<Queue<String>> is a subtype of Queue<Queue<String>>
2. Queue<QueueImpl<String>> is a subtype of Queue<Queue<String>>
3. Both
4. Neither



Subtyping and Generics

Which of these are true, assuming that class QueueImpl<E> implements interface Queue<E>?

1. QueueImpl<Queue<String>> is a subtype of Queue<Queue<String>>
2. Queue<QueueImpl<String>> is a subtype of Queue<Queue<String>>
3. Both
4. Neither

Answer: 1

Other subtleties with Generics

- Unlike OCaml, Java classes and methods can be generic only with respect to *reference* types.
  - Not possible to do: Queue<int>
  - Must instead do: Queue<Integer>

- Java Arrays cannot be generic!
  - Not possible to do:

```
class C<E> {
    E[] genericArray;
    public C() {
        genericArray = new E[];
    }
}
```

  - There are various (hacky) workarounds that involve typecasts or reflection.

The Java Collections Library

A case study in subtyping and generics...

that is also very useful...

(But many pitfalls and Java idiosyncrasies!)

Java Packages

- Java code can be organized into *packages* that provide namespace management.
  - Somewhat like OCaml's modules
  - Packages contain groups of related classes and interfaces.
  - Packages are organized hierarchically in a way that mimics the file system's directory structure.
- A .java file can *import* (parts of) packages that it needs access to:

```
import org.junit.Test;    // just the JUnit Test class
import java.util.*;       // everything in java.util
```

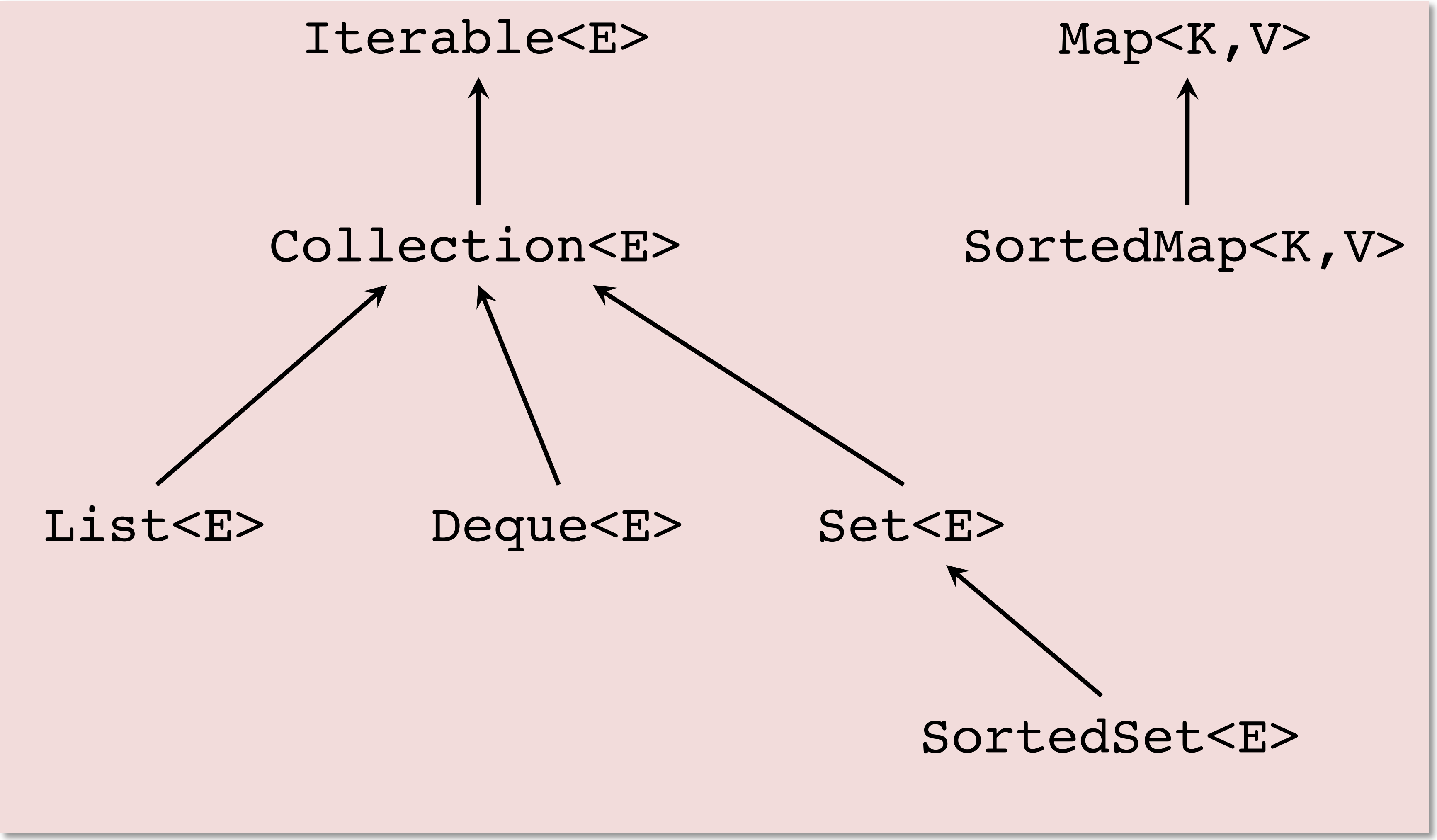
- Important packages:
  - java.lang, java.io, java.util, java.math, org.junit
- See documentation at:
  http://docs.oracle.com/javase/8/docs/api/
- You should read this documentation in preparation for HW 7

Reading Java Docs

java.util

https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html

Interfaces* of the Collections Library



*not all of them!

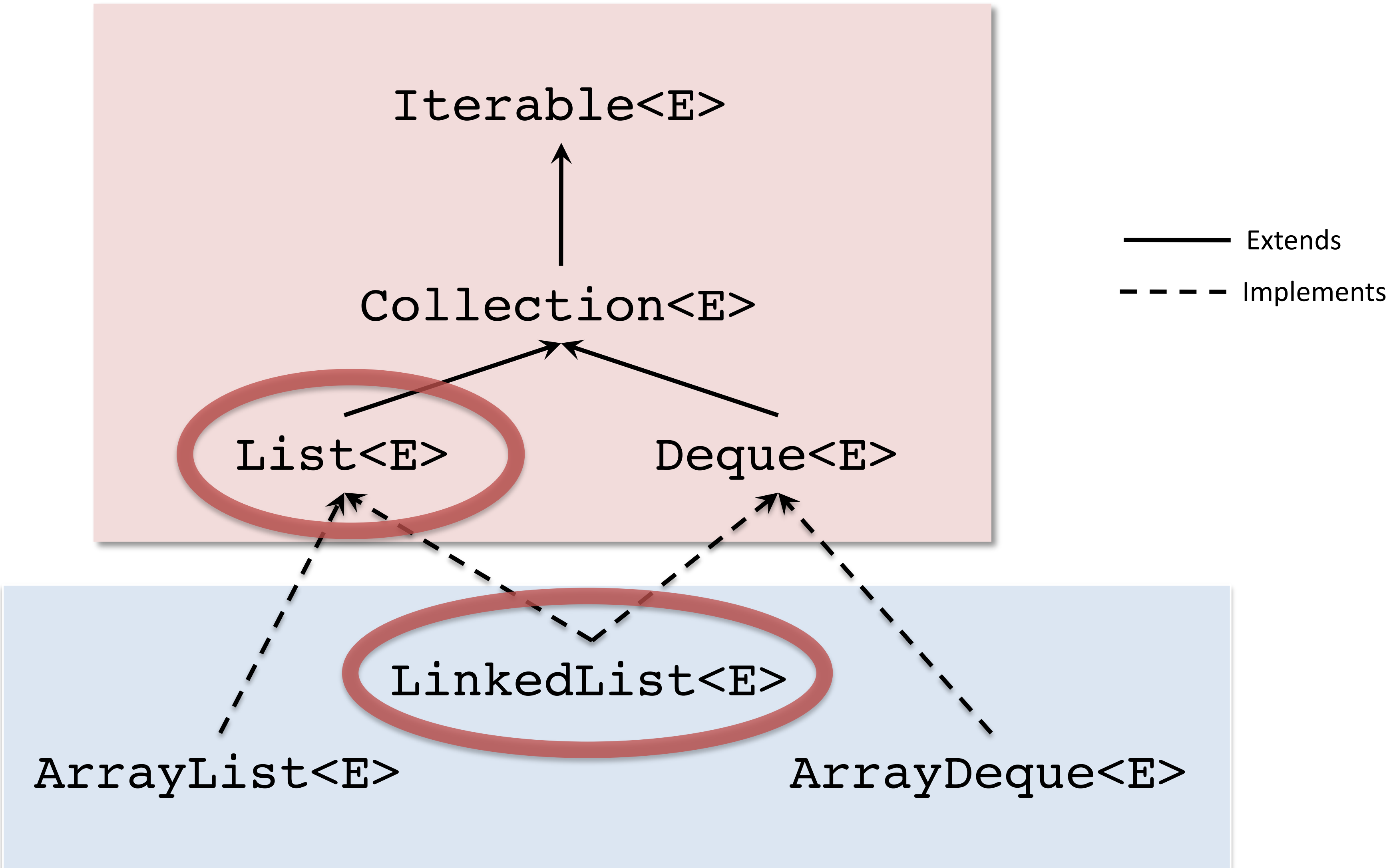
Collection<E> Interface (Excerpt)

```
interface Collection<E> extends Iterable<E> {
  // basic operations
  int size();
  boolean isEmpty();
  boolean add(E o);
  boolean remove(Object o);     // why not E?*
  boolean contains(Object o);

  // bulk operations
  …
}
```

- We've already seen a similar interface in the OCaml part of the course.
- Most collections are designed to be *mutable* (like queues)

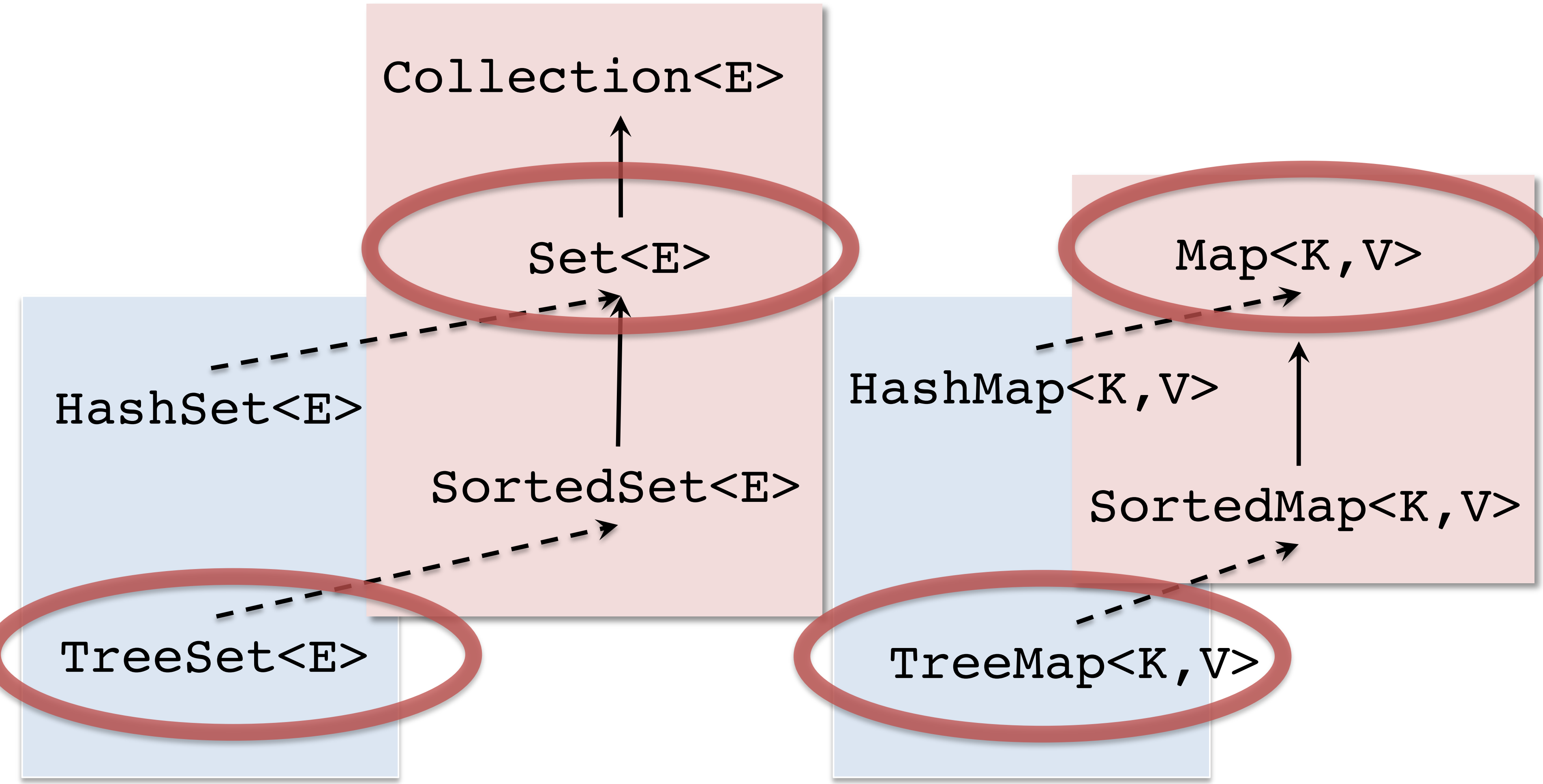
* Why not E? Internally, collections use the `equals` method to check for equality – membership is determined by o.equals, which does not have to be false for objects of different types. Most applications only store and remove one type of element in a collection, in which case this subtlety never becomes an issue.

Sequences



Iterable<E>
Collection<E>
List<E>
Deque<E>
LinkedList<E>
ArrayList<E>
ArrayDeque<E>
Extends
Implements

Sets and Maps*



*Read javadocs before instantiating these classes! There are some important details to be aware of to use them correctly.

TreeSet Demo

implement Comparable when using SortedSets and Sorted Maps

Implement Comparable when using SortedSets and Sorted Maps.

See TreeSetExample.java and Point.java

TREESET DEMO

Buggy Use of TreeSet implementation

```java
import java.util.*;

class Point {
    private final int x, y;
    public Point(int x0, int y0) { x = x0; y = y0; }
    public int getX(){ return x; }
    public int getY(){ return y; }
}

public class TreeSetDemo {
    public static void main(String[] args) {
        Set<Point> s = new TreeSet<Point>();
        s.add(new Point(1,1));
    }
}
```

RUNTIME
ERRROR

Exception in thread "main" java.lang.ClassCastException:
Point cannot be cast to java.base/java.lang.Comparable
at java.base/java.util.TreeMap.compare(TreeMap.java:1291)
at java.base/java.util.TreeMap.put(TreeMap.java:536)
at java.base/java.util.TreeSet.add(TreeSet.java:255)
at TreeSetDemo.main(TreeSetDemo.java:14)

A Crucial Detail of `TreeSet`

Constructor Detail

TreeSet

public TreeSet()

Constructs a new, empty tree set, sorted according to the natural ordering of its elements. All elements inserted into the set must implement the `Comparable` interface. Furthermore, all such elements must be mutually comparable: `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the set. ...

The Interface `Comparable`

`public interface Comparable<T>`

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*. ...

Methods of `Comparable`

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
`int`	`compareTo(T o)` Compares this object with the specified object for order.

Method Detail

compareTo

`int compareTo(T o)`

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all x and y. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 &&`

Adding Comparable to Point

```
import java.util.*;

class Point implements Comparable<Point> {
    private final int x, y;
    public Point(int x0, int y0) { x = x0; y = y0; }
    public int getX(){ return x; }
    public int getY(){ return y; }

    public int compareTo(Point o) {
        if (this.x < o.x) {
            return -1;
        } else if (this.x > o.x) {
            return 1;
        } else if (this.y < o.y) {
            return -1;
        } else if (this.y > o.y) {
            return 1;
        }
        return 0;
    }
}
```

```
Point p1 = new Point(0,1);
Point p2 = new Point(0,2);
p1.compareTo(p2);  // -1
p2.compareTo(p1);  // 1
p1.compareTo(p1);  // 0
```

Digging Deeper into Comparable

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. *In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals method.*

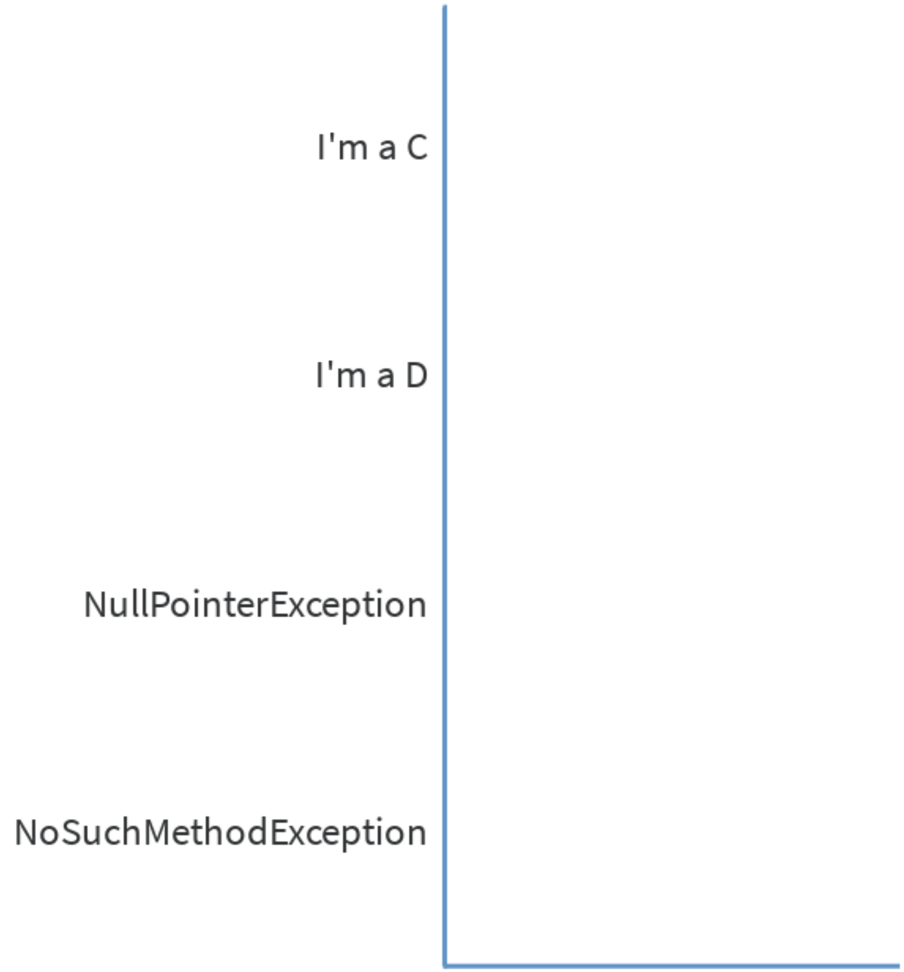
How do we change the definition of equals?

Method Overriding

When a subclass replaces an inherited method with its own re-definition…

What gets printed to the console?

```
class C {
        public void printName() { System.out.println("I'm a
C"); }
}

class D extends C {
        public void printName() { System.out.println("I'm a
D"); }
}

// somewhere in main
C c = new D();
c.printName();
```



A Subclass can *Override* its Parent

```java
class C {
      public void printName() { System.out.println("I'm a C");
}
}

class D extends C {
      public void printName() { System.out.println("I'm a D");
}
}

// somewhere in main
C c = new D();
c.printName();
```

What gets printed to the console?

1. I'm a C
2. I'm a D
3. NullPointerException
4. NoSuchMethodException

Answer: I'm a D

A Subclass can *Override* its Parent

```
class C {
  public void printName() { System.out.println("I'm a C"); }
}

class D extends C {
  public void printName() { System.out.println("I'm a D"); }
}

// somewhere in main
C c = new D();
c.printName();
```

- Our ASM model for dynamic dispatch already explains what will happen when we run this code.
- Useful for changing the default behavior of classes.
- But… can be confusing and difficult to reason about if not used carefully.

Overriding Example

Workspace | Stack | Heap | Class Table

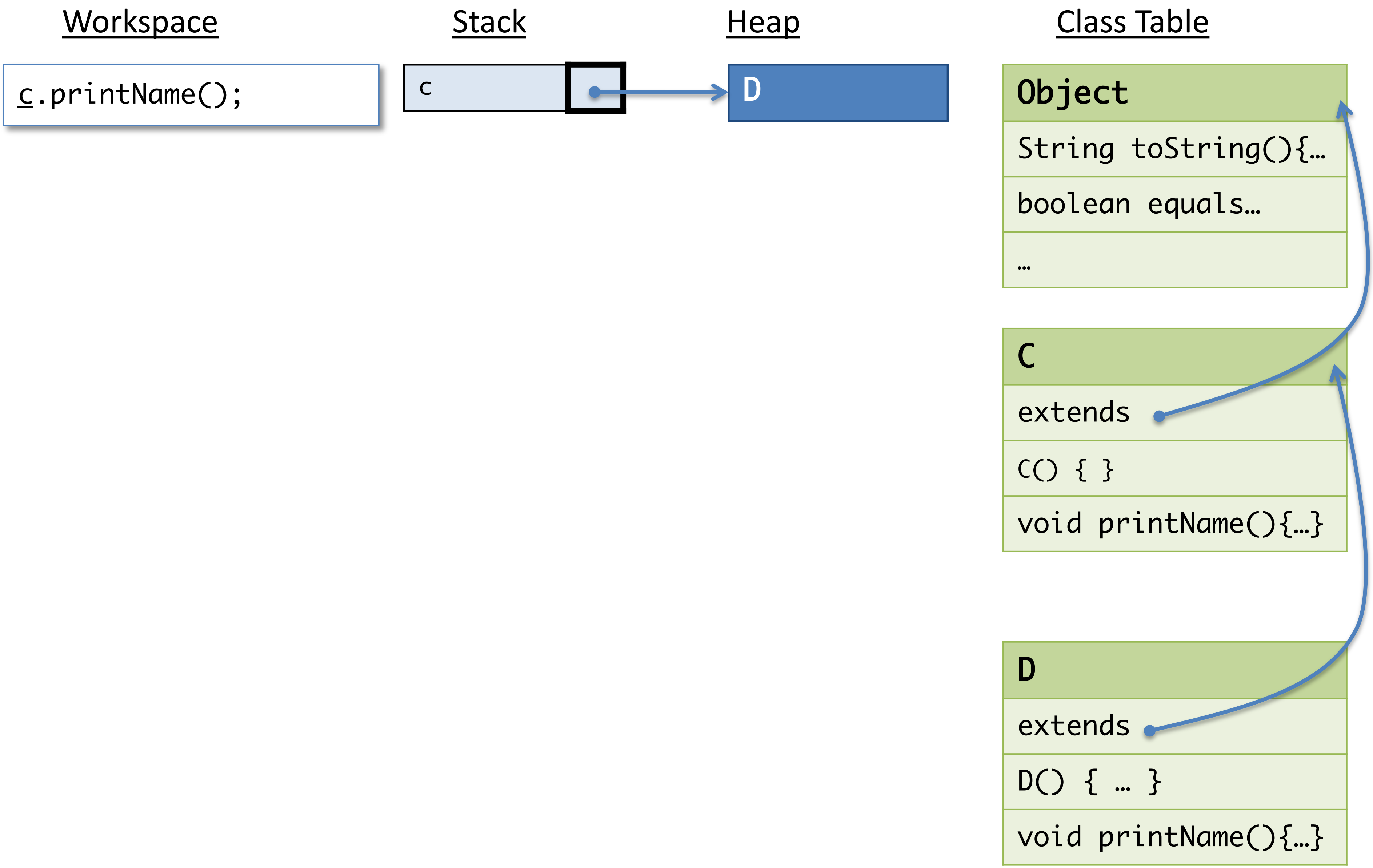
```
C c = new D();
c.printName();›
```

Object
- String toString(){…
- boolean equals…
- …

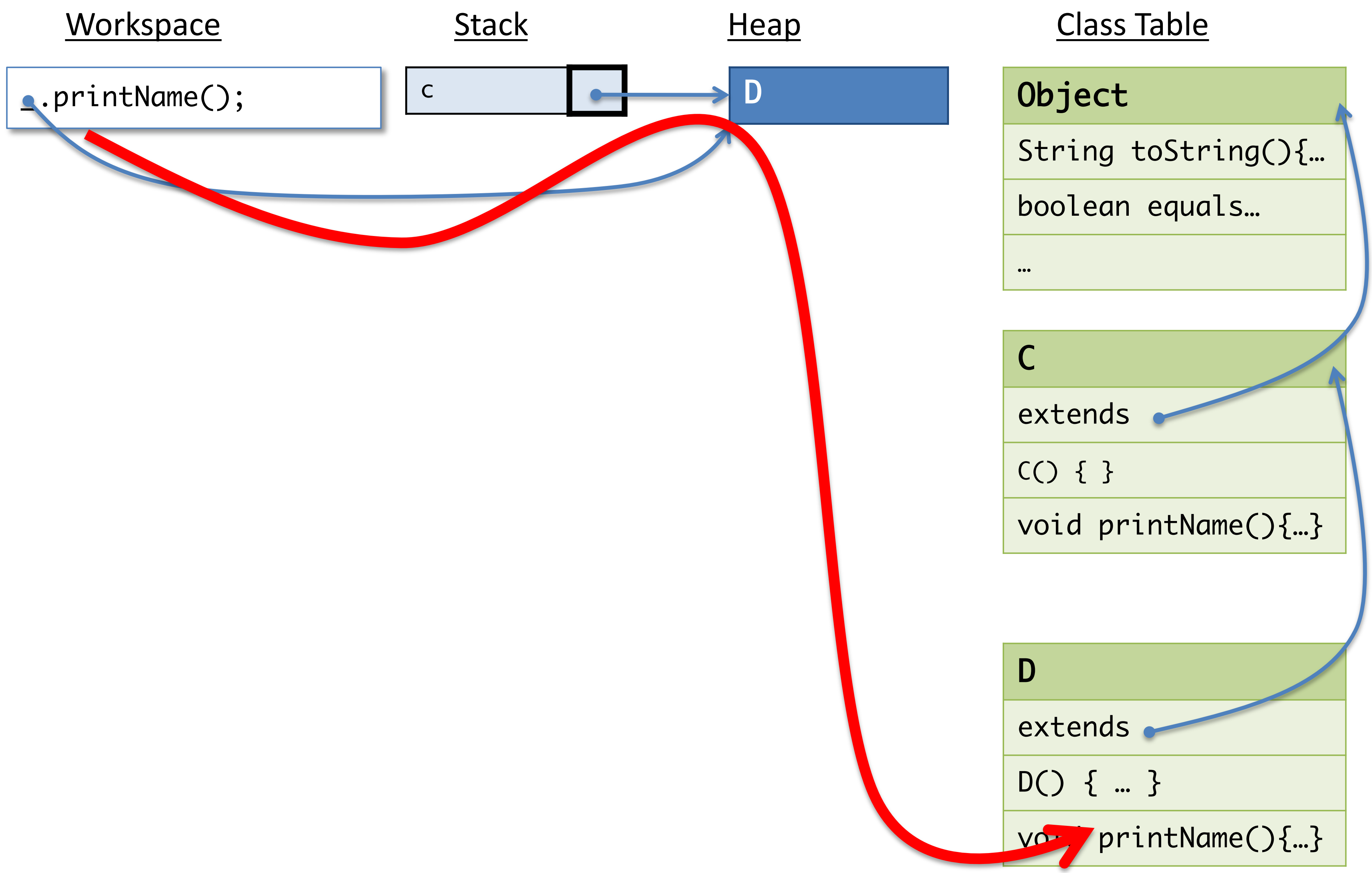
C
- extends
- C() { }
- void printName(){…}

D
- extends
- D() { … }
- void printName(){…}

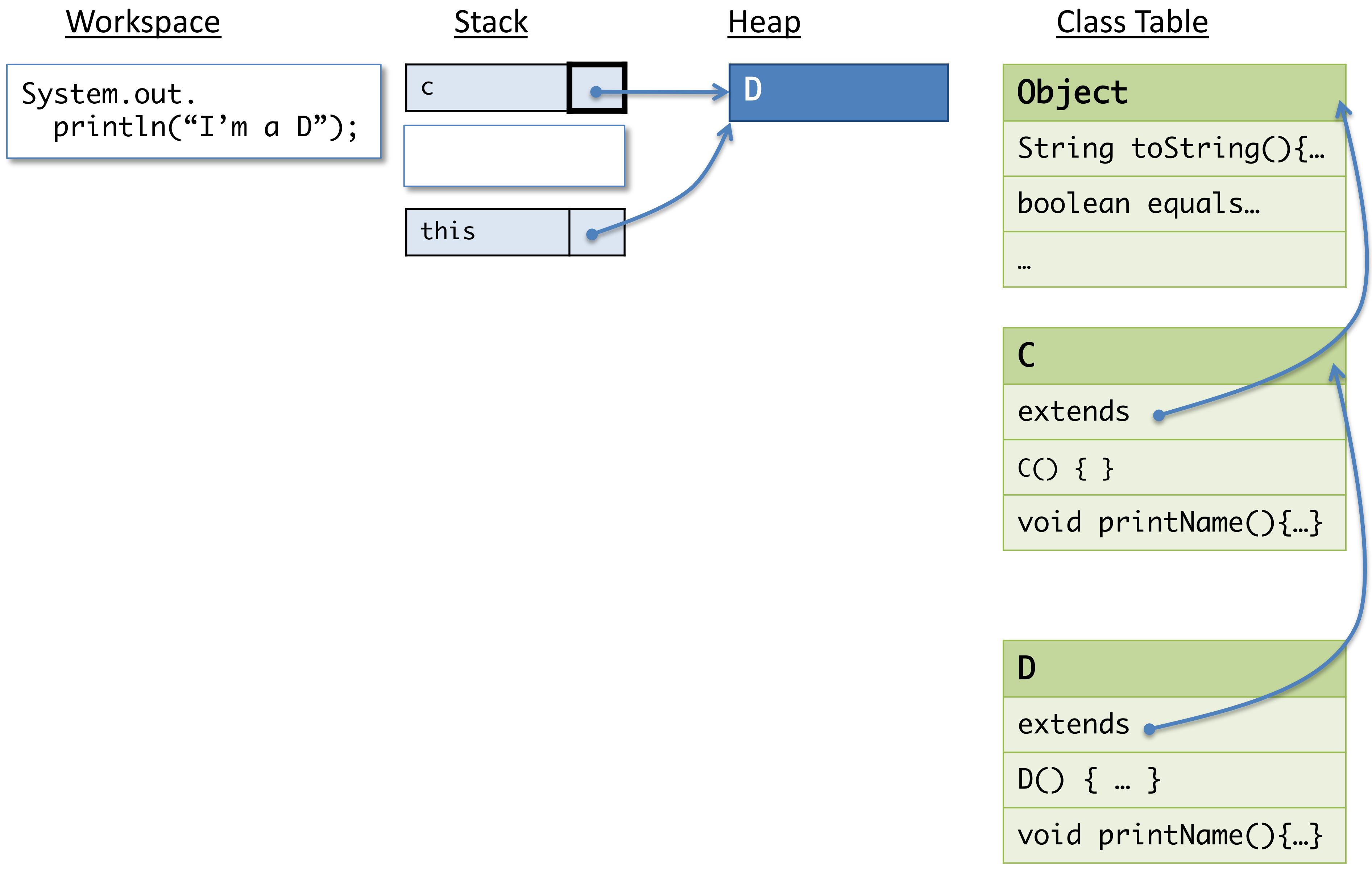
Overriding Example



Overriding Example

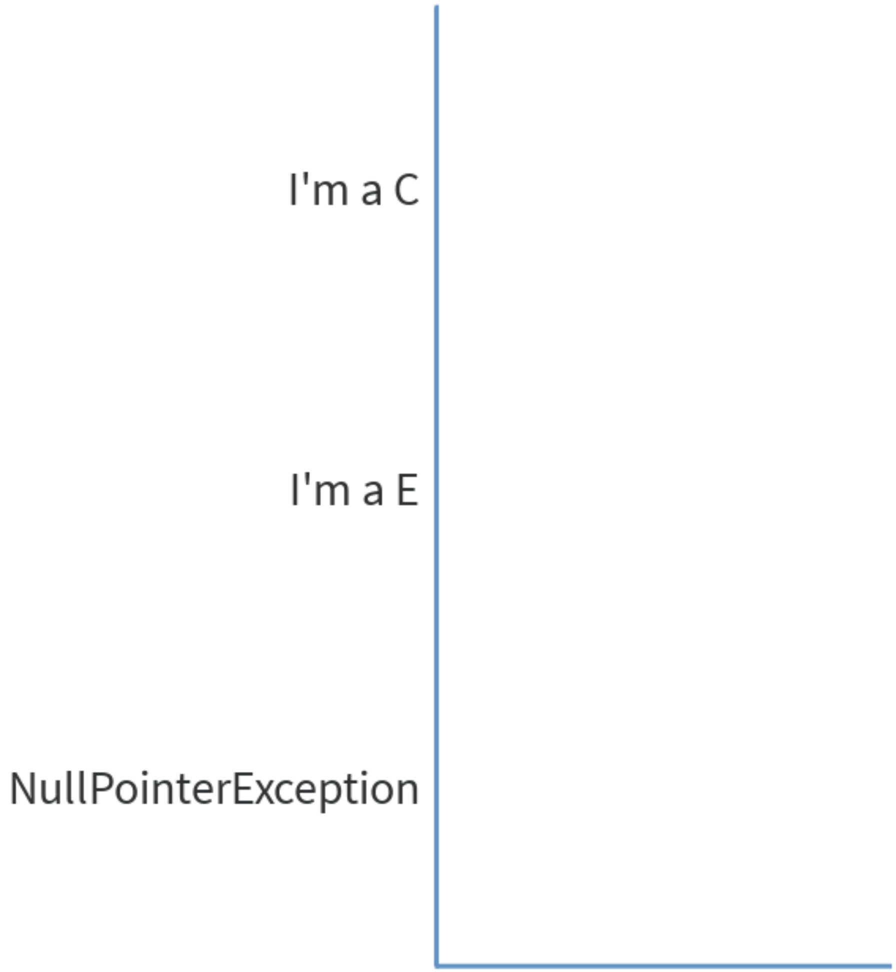


Overriding Example



What gets printed to the console?

```java
class C {

  public void printName() {
    System.out.println("I'm a " + getName());
  }

  public String getName() {
    return "C";
  }
}

class E extends C {

  public String getName() {
    return "E";
  }
}

// in main
C c = new E();
c.printName();
```



Difficulty with Overriding

```
class C {

  public void printName() {
    System.out.println("I'm a " + getName());
  }

  public String getName() {
    return "C";
  }
}

class E extends C {

  public String getName() {
    return "E";
  }
}

// in main
C c = new E();
c.printName();
```

What gets printed to the console?

1. I'm a C
2. I'm a E
3. NullPointerException

Answer: I'm a E

Difficulty with Overriding

```
class C {

  public void printName() {
    System.out.println("I'm a " + getName());
  }

  public String getName() {
    return "C";
  }
}

class E extends C {

  public String getName() {
    return "E";
  }
}

// in main
C c = new E();
c.printName();
```

The C class might be in another package, or a library...

Whoever writes E might not be aware of the implications of changing getName.

Overriding the getName method causes the behavior of printName to change!

- Overriding can break invariants/abstractions relied upon by the superclass.