Programming Languages and Techniques (CIS120)

Lecture 36

April 23, 2014

Overriding and Equality

Announcements

- HW 10 has a HARD deadline
  - You must submit by midnight, April 30th
  - Demo your project to your TA during reading days
- Friday's lecture is a BONUS lecture
  - Not directly related to game project or Java libraries
  - No clicker quizzes
  - Fun!
- Senior project demos this morning. Check them out after class!

Clicker quiz

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public int getY() { return y; }
}

// somewhere in main…
List<Point> l = new LinkedList<Point>();
l.add(new Point(1,2));
System.out.println(l.contains(new Point(1,2)));
```

What gets printed to the console?

1. true
2. false

Method Overriding

A Subclass can *Override* its Parent

```
public class C {
  public void printName() { System.out.println(“I’m a C”); }
}

public class D extends C {
  public void printName() { System.out.println(“I’m a D”); }
}

C c = new D();
c.printName();    // what gets printed?
```

What gets printed to the console?

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

A Subclass can *Override* its Parent

```
public class C {
  public void printName() { System.out.println(“I’m a C”); }
}

public class D extends C {
  public void printName() { System.out.println(“I’m a D”); }
}

C c = new D();
c.printName();   // what gets printed?
```

- 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

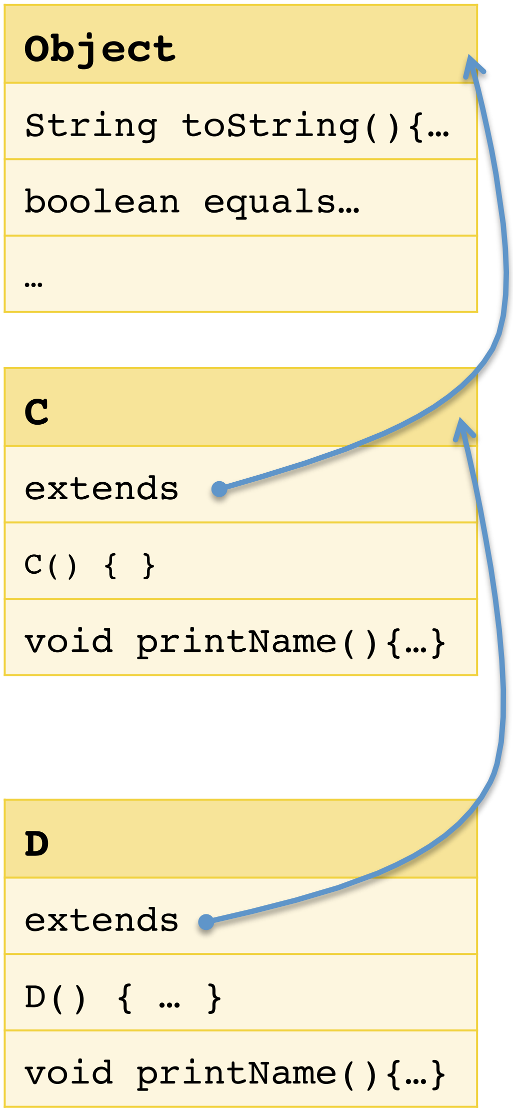
Workspace

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

Stack

Heap

Class Table



Object

String toString(){…

boolean equals…

…

C

extends

C() { }

void printName(){…}

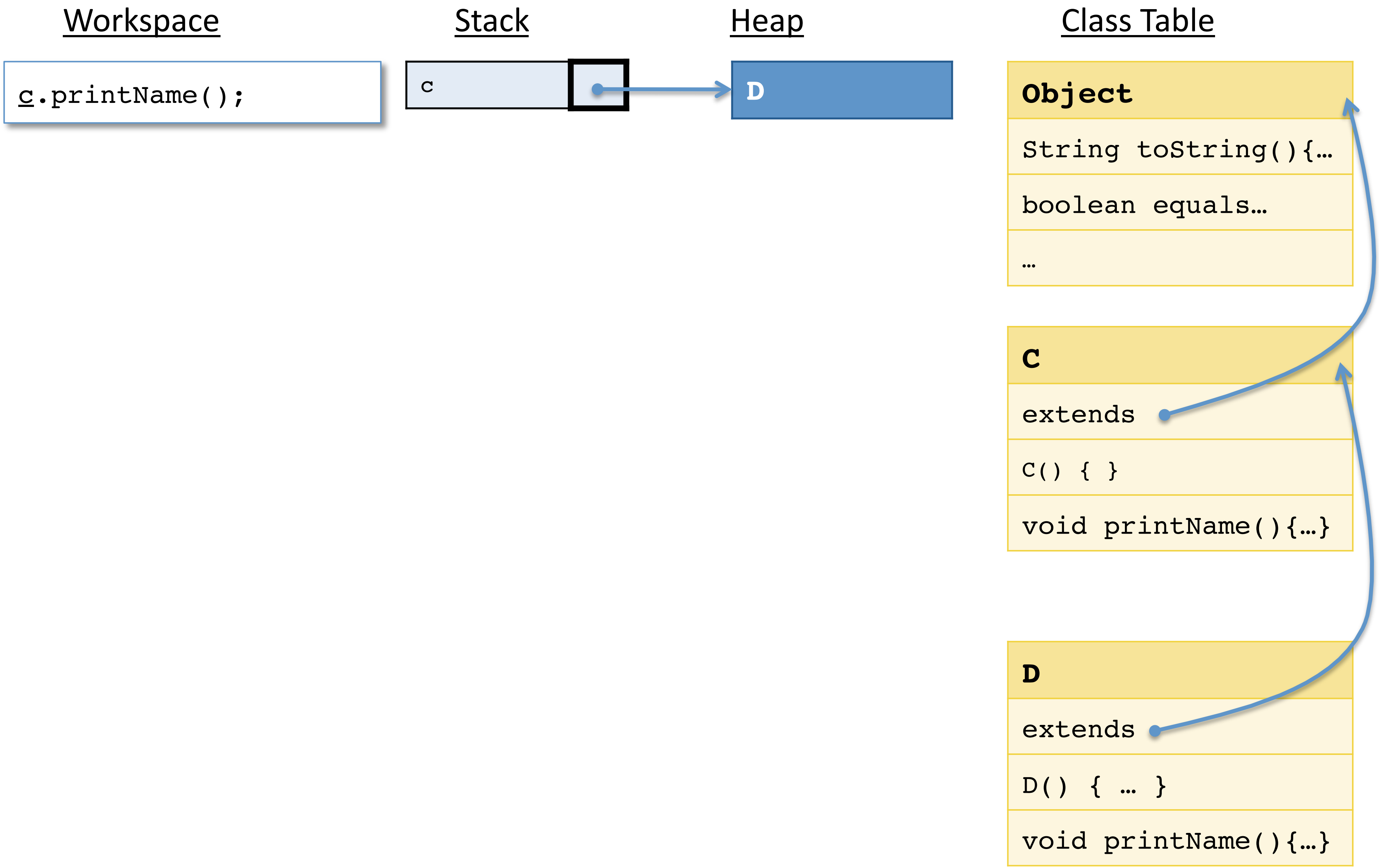
D

extends

D() { … }

void printName(){…}

Overriding Example



Workspace

```
c.printName();
```

Stack

c

Heap

D

Class Table

Object

String toString(){…

boolean equals…

…

C

extends

C() { }

void printName(){…}

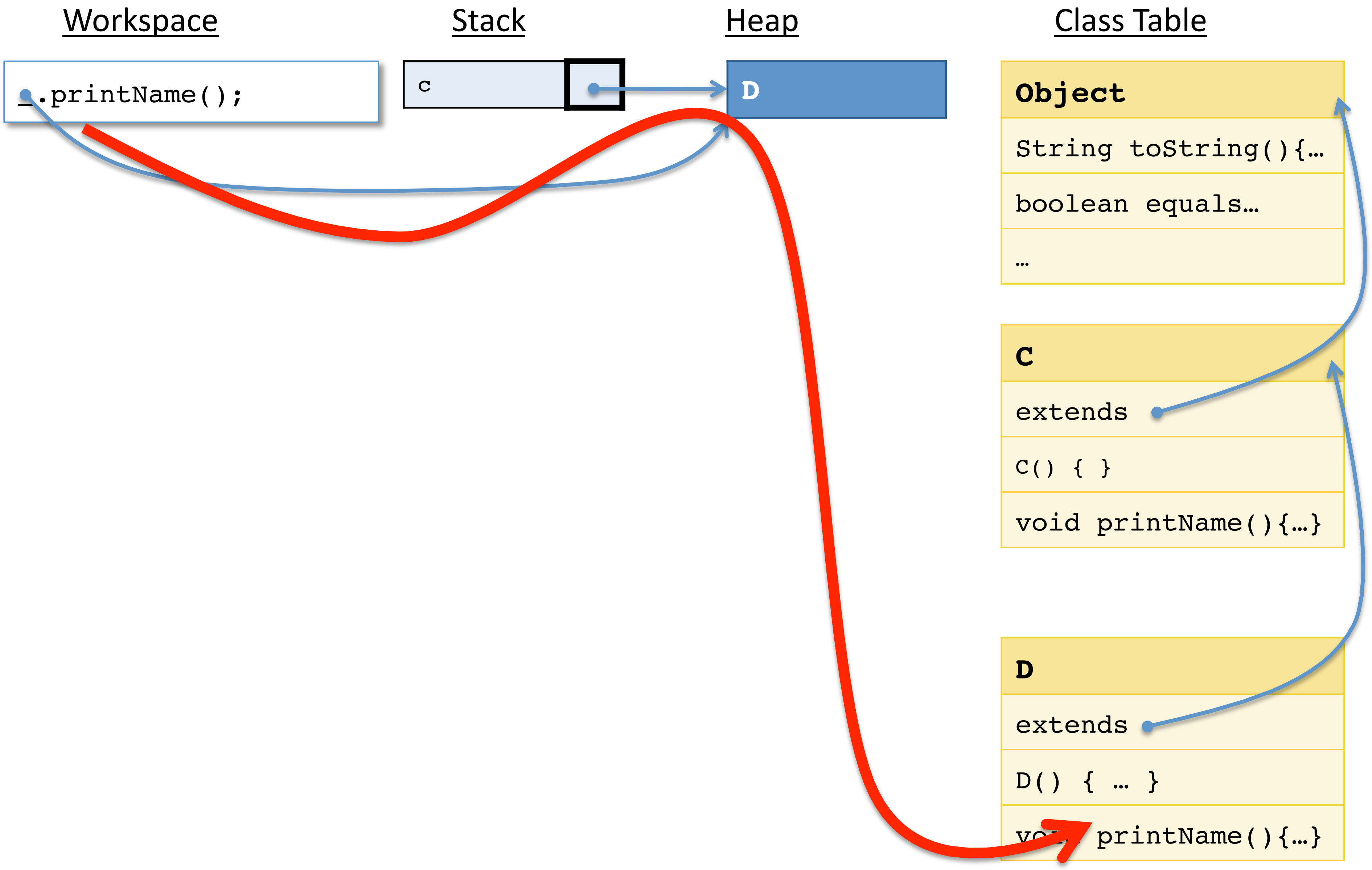
D

extends

D() { … }

void printName(){…}

Overriding Example



Workspace

_.printName();

Stack

c

Heap

D

Class Table

Object

String toString(){…

boolean equals…

…

C

extends

C() { }

void printName(){…}

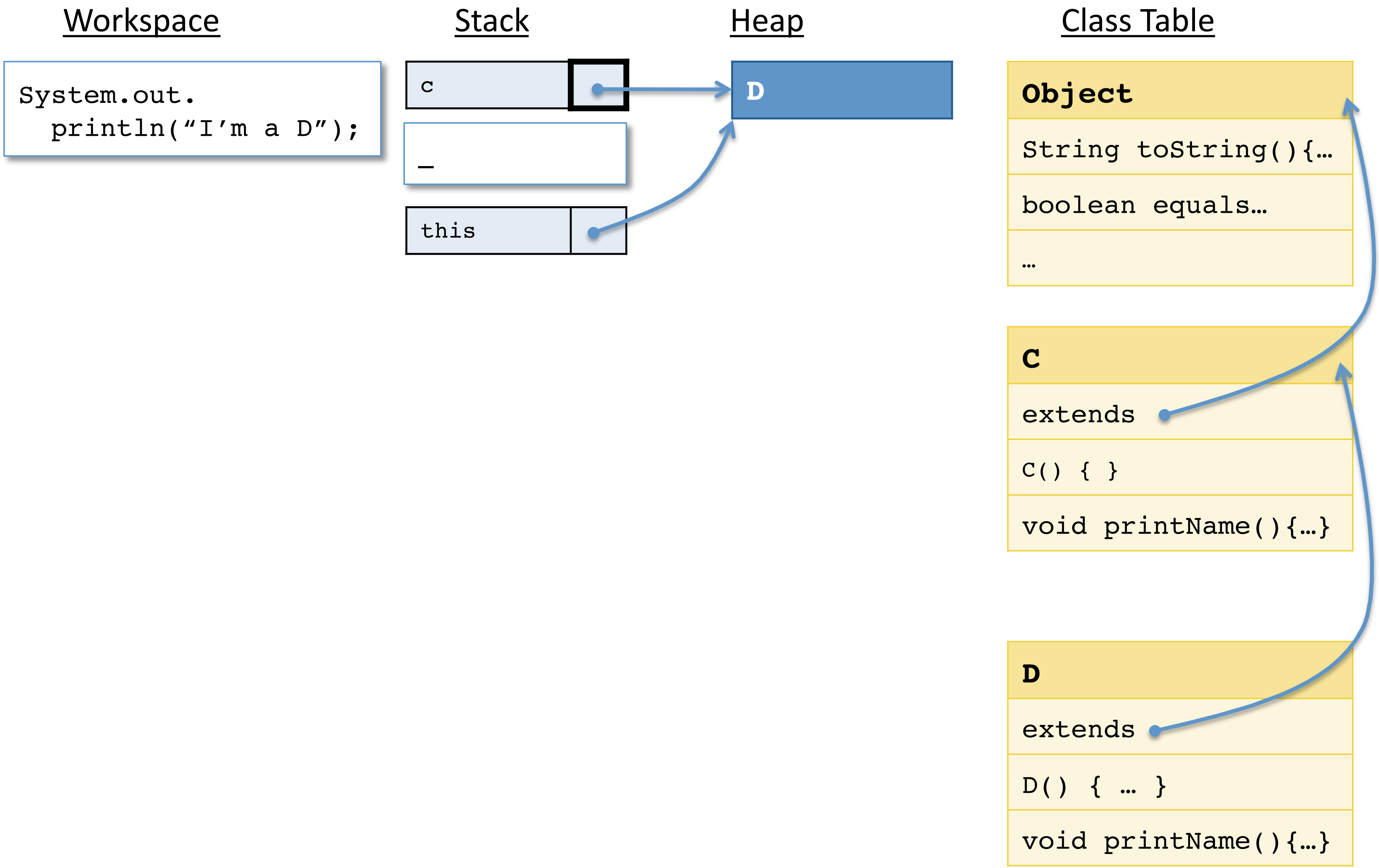
D

extends

D() { … }

void printName(){…}

Overriding Example



Workspace

```
System.out.
  println(“I’m a D”);
```

Stack

c

_

this

Heap

D

Class Table

Object

String toString(){…

boolean equals…

…

C

extends

C() { }

void printName(){…}

D

extends

D() { … }

void 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. I’m an E
4. NullPointerException

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 wrote D might not be aware of the implications of changing `getName`.

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

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

When To Override?

- Only override methods when the parent class is *designed* specifically to support such modifications:
  - If the library designer specifically describes the behavioral contract that the parent methods assume about overridden methods (e.g. equals, paintComponent)
  - If you're writing the code for both the parent and child class (and will maintain control of both parts as the software evolves) it might be OK to overrride.
  - Either way: document the design
  - Use the `@Override` annotation to mark intentional overriding

- Look for other means of achieving the desired outcome:
  - Use composition & delegation (i.e. wrapper objects) rather than overriding

How to prevent overriding

- By default, methods can be overridden in subclasses.
- The `final` modifier changes that.
- Final methods *cannot* be overridden in subclasses
  - Prevents subclasses from changing the “behavioral contract” between methods by overriding
  - `static final` methods cannot be hidden

- Similar, but not the same as final fields and local variables:
  - Act like the immutable name bindings in OCaml
  - Must be initialized (either by a static initializer or in the constructor) and cannot thereafter be modified.
  - `static final` fields are useful for defining constants (e.g. `Math.PI`)

Case study: Equality

Motivating example

```java
public class Point {
   private final int x;
   private final int y;
   public Point(int x, int y) { this.x = x; this.y = y; }
   public int getX() { return x; }
   public int getY() { return y; }
}

// somewhere in main…
List<Point> l = new LinkedList<Point>();
l.add(new Point(1,2));
System.out.println(l.contains(new Point(1,2)));
```

We *must* override equals in the Point class to get the desired behavior.

When to override equals

- In classes that represent immutable *values*
  - String already overrides equals
  - Our Point class is a good candidate

- When there is a “logical” notion of equality
  - The collections library overrides equality for Sets
    (e.g. two sets are equal if and only if they contain equal elements)

- Whenever instances of a class might need to serve as *elements of a set* or as *keys in a map*
  - The collections library uses `equals` internally to define set membership and key lookup
  - (This is the problem with the example code)

When *not* to override equals

- When each instance of a class is inherently unique
  - *Often* the case for mutable objects (since its state might change, the only sensible notion of equality is identity)
  - Classes that represent "active" entities rather than data (e.g. threads, gui components, etc.)

- When a superclass already overrides equals and provides the correct functionality.
  - Usually the case when a subclass adds only new methods, not fields

How to override equals

*See the very nicely written article “How to write an Equality Method in Java” by Oderski, Spoon, and Venners (June 1, 2009) at http://www.artima.com/lejava/articles/equality.html

The contract for equals

- The equals method implements an *equivalence relation* on non-null objects.
- It is *reflexive*:
  - for any non-null reference value x, x.equals(x) should return true
- It is *symmetric*:
  - for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- It is *transitive*:
  - for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent:
  - for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified
- For any non-null reference x, x.equals(null) should return false.

Directly from: http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html#equals(java.lang.Object)

First attempt

```
public class Point {
  private final int x;
  private final int y;
  public Point(int x, int y) {this.x = x; this.y = y;}
  public int getX() { return x; }
  public int getY() { return y; }
  public boolean equals(Point that) {
    return (this.getX() == that.getX() &&
            this.getY() == that.getY());
  }
}
```

Gocha: *overloading*, vs. *overriding*

```
public class Point {
  …
  // overloaded, not overridden
  public boolean equals(Point that) {
    return (this.getX() == that.getX() &&
                this.getY() == that.getY());
  }
}
Point p1 = new Point(1,2);
Point p2 = new Point(1,2);
Object o = p2;
System.out.println(p1.equals(o));
// prints false!
System.out.println(p1.equals(p2));
// prints true!
```

The type of equals as declared in Object is:

```
 public boolean equals(Object o)
```

The implementation above takes a Point *not* an Object!

Overriding equals, take two

Properly overridden equals

```
public class Point {
   …
   @Override
   public boolean equals(Object o) {
       if (o == null) { return false; }
      // what do we do here???
   }
}
```

- Start with the null check. Why can we immediately return false?

- Use the `@Override` annotation when you *intend* to override a method so that the compiler can warn you about accidental overloading.

- Now what? How do we know whether the o is even a Point?
  - We need a way to check the *dynamic* type of an object.

instanceof

- The `instanceof` operator tests the *dynamic* type of any object

```
Point p = new Point(1,2);
Object o1 = p;
Object o2 = "hello";
System.out.println(p instanceof Point);
       // prints true
System.out.println(o1 instanceof Point);
       // prints true
System.out.println(o2 instanceof Point);
       // prints false
System.out.println(p instanceof Object);
       // prints true
System.out.println(null instanceof Object);
       // prints false
```

What gets printed? (1=true, 2=false)

- In the case of equals, instanceof is appropriate because the method behavior depends on the dynamic types of *two* objects: o1.equals(o2)
- But… use `instanceof` judiciously – usually dynamic dispatch is better.

Type Casts

- We can test whether o is a Point using instanceof

```
@Override
public boolean equals(Object o) {
  if (o == null) { return false; }
  if (!(o instanceof Point)) { return false; }
   // o is a point - how do we treat it as such?


   …
}
```

- Use a type *cast*: `(Point) o`
  - At compile time: the expression `(Point) o` has type `Point`.
  - At runtime: check whether the dynamic type of o is a subtype of `Point`, if so evaluate to o, otherwise raise a `ClassCastException`
  - As with instanceof, use casts judiciously – i.e. almost never

Refining the `equals` implementation

```
@Override
public boolean equals(Object o) {
   if (o == null) { return false; }
   if (!(o instanceof Point)) { return false; }
   Point that = (Point) o;
   if (x != that.x) { return false; }
   if (y != that.y) { return false; }
   return true;
}
```

This cast is guaranteed to succeed.

One more addition

```
@Override
public boolean equals(Object o) {
    if (this == o) { return true; }
    if (o == null) { return false; }
    if (!(o instanceof Point)) { return false; }
    Point that = (Point) o;
    if (x != that.x) { return false; }
    if (y != that.y) { return false; }
    return true;
}
```

An optimization for when the argument is an alias.

Now the example code from the slide 3 will behave as expected.
But… are we done? Does this implementation satisfy the contract?