Programming Languages and Techniques (CIS120)

Lecture 28

Overriding Methods, Equality, Enumerations

Chapters 25 and 26

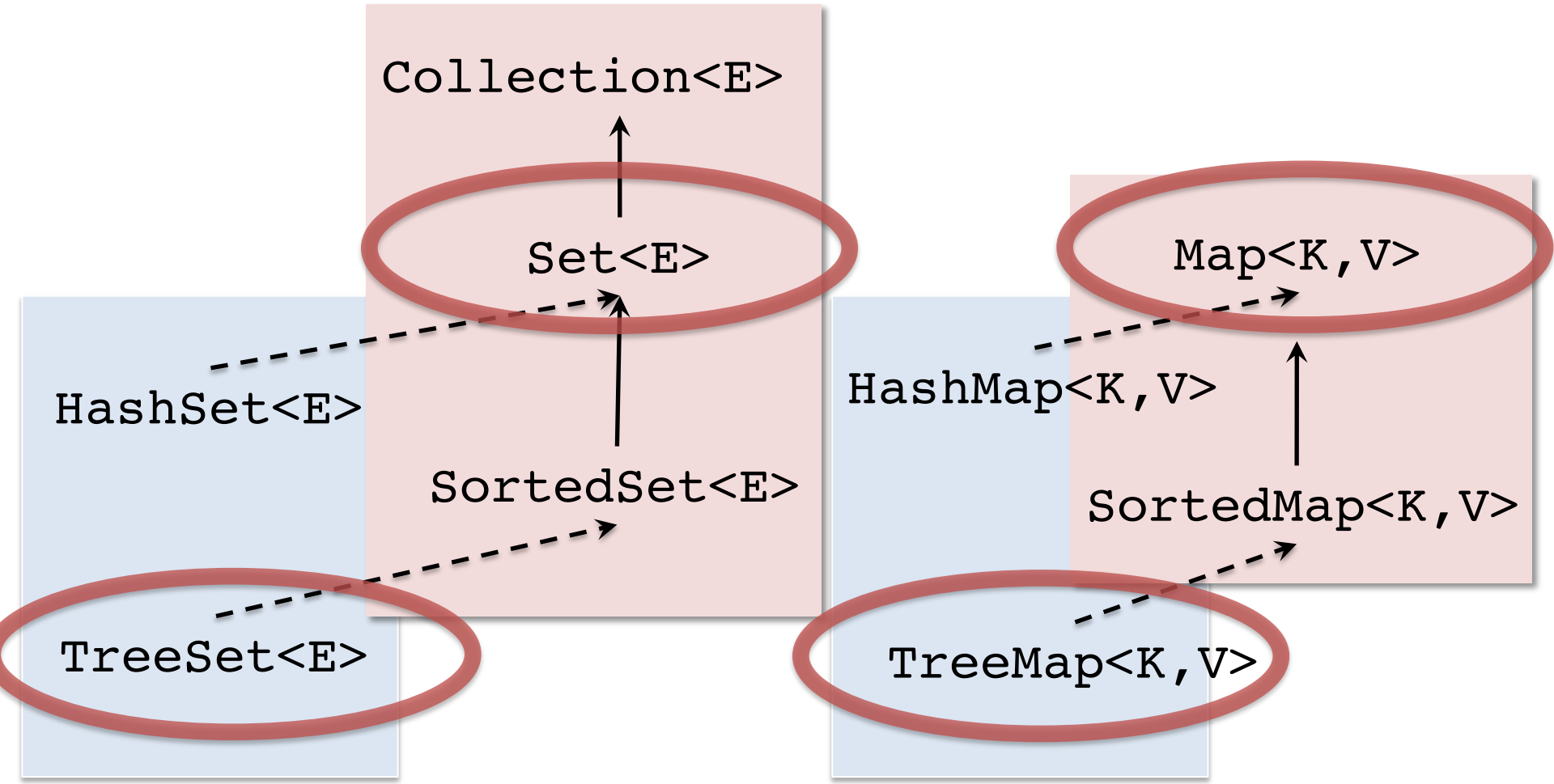
The Java Collections Library

A case study in subtyping and generics...

that is also very useful...

(But many pitfalls and Java idiosyncrasies!)

Sets and Maps*



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

TreeSet Demo

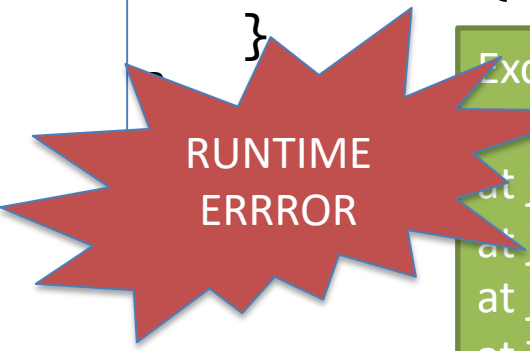
implement Comparable when using SortedSets and Sorted Maps

Buggy Use of TreeSet implementation

```
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));
    }
```




```
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…

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();
```

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

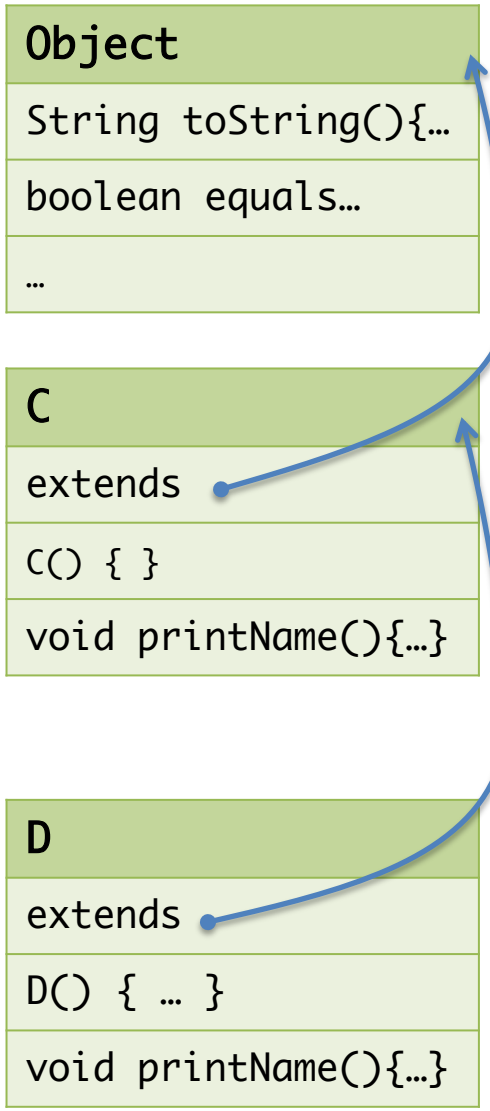
Workspace

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

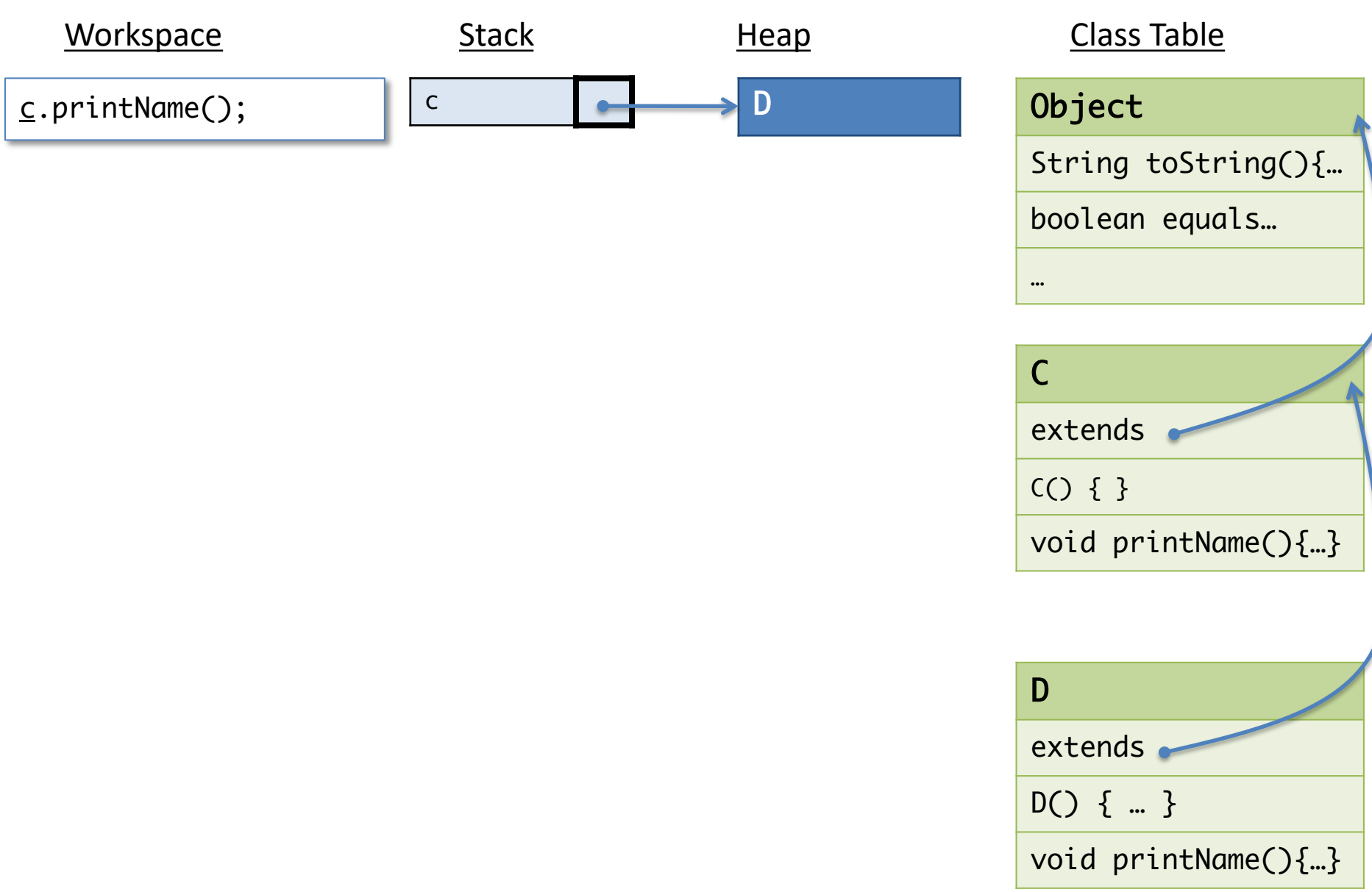
Stack

Heap

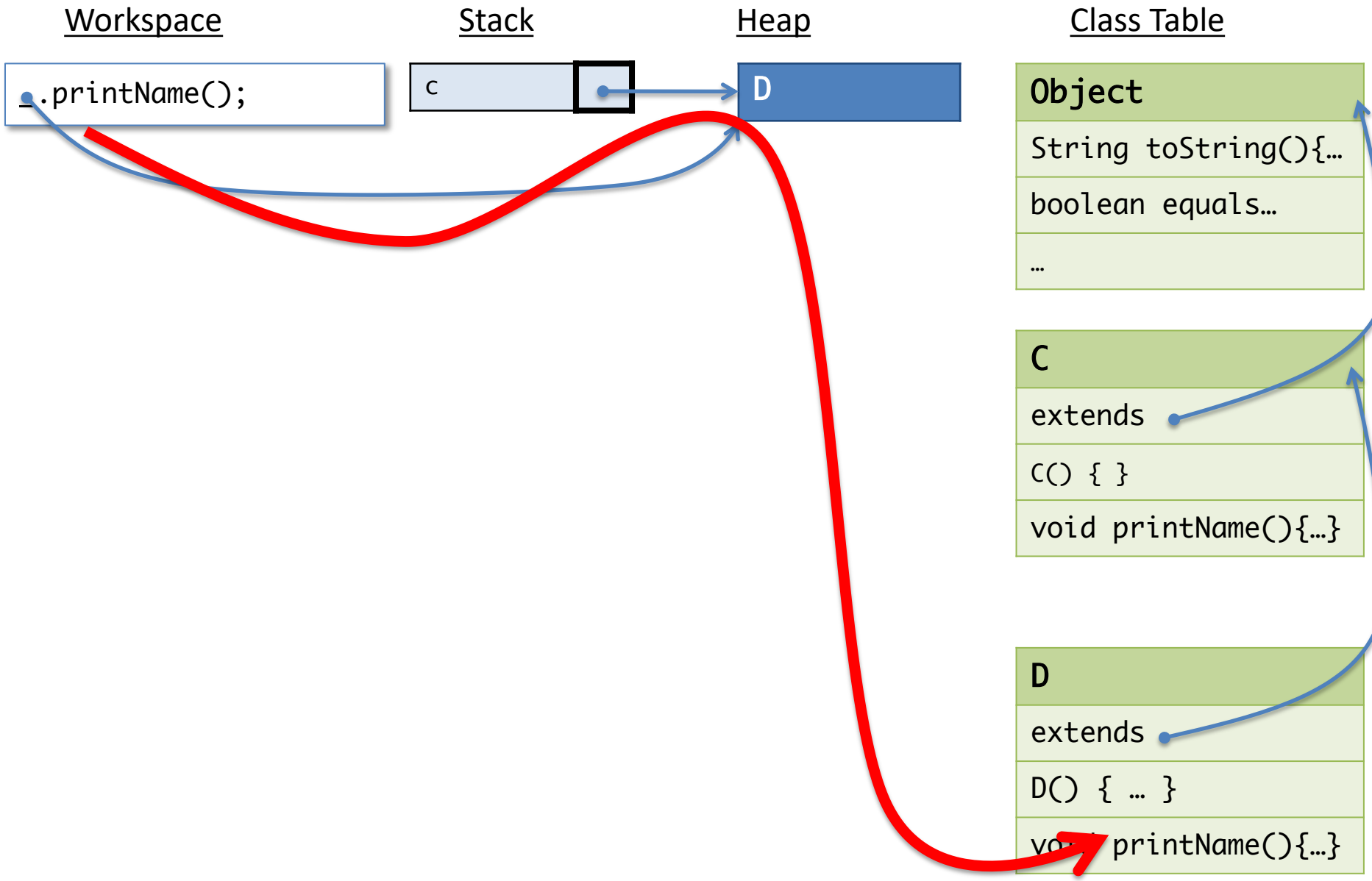
Class Table



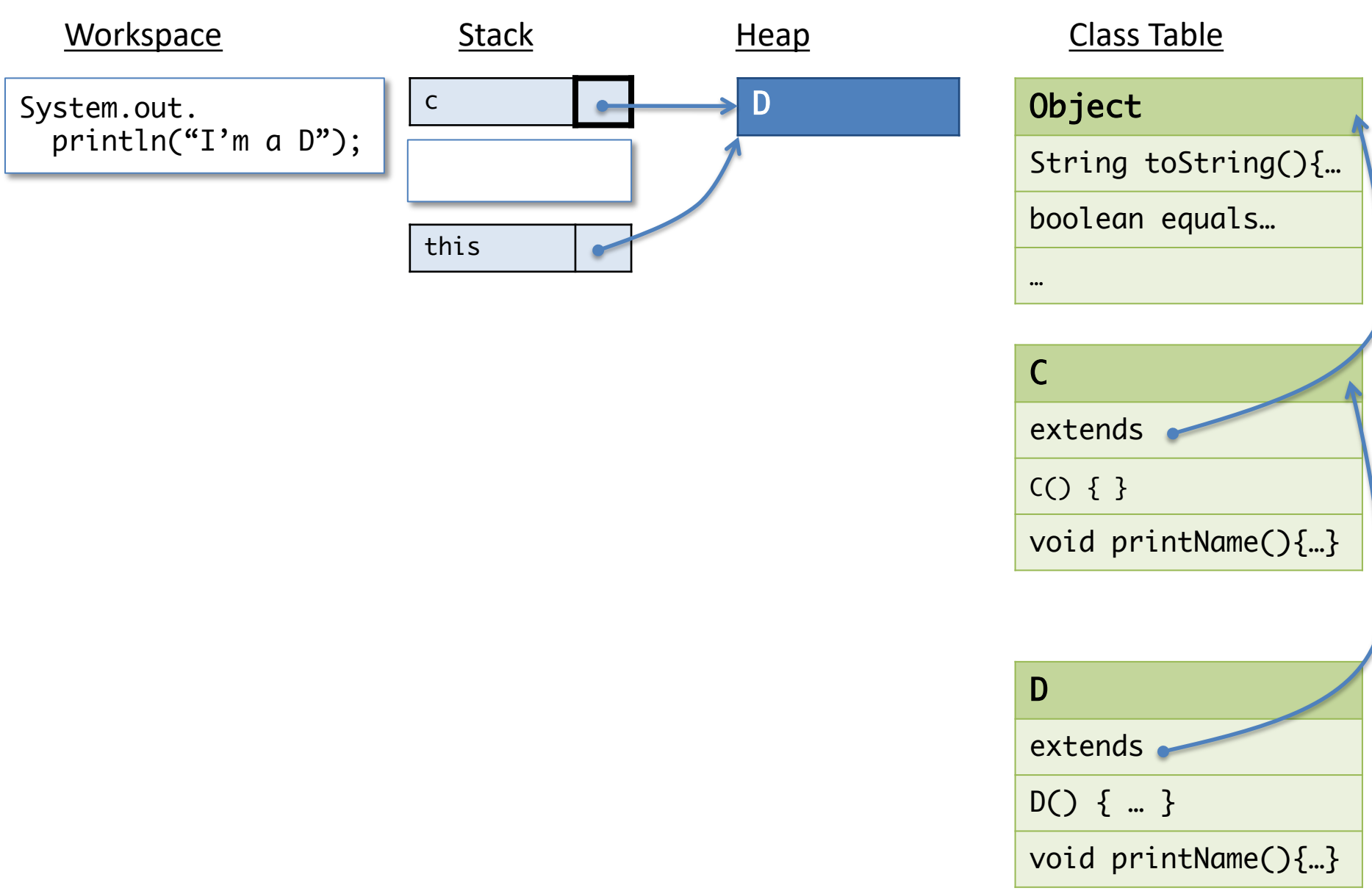
Overriding Example



Overriding Example



Overriding Example



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.

Case study: Equality

A common, but tricky, situation where overriding is needed

Consider this example

```
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

Why?

Answer: False

From Java API:

public interface **Collection<E>** extends Iterable<E>

…
Many methods in Collections Framework interfaces are defined in terms of the equals method. For example, the specification for the contains(Object o) method says: "returns true if and only if this collection contains at least one element e such that (o==null ? e==null : o.equals(e)). …

The `Object` class implements the `.equals` method using *reference* equality (i.e. ==).
We want *structural* equality for `Points` in this example.

When to override equals

- In classes that represent immutable *values*
  - `String` overrides equals for this reason
  - Our `Point` class is another 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 their 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` with the desired functionality.
  - Usually the case when a subclass is implemented by adding only new methods, *but not fields*

How to override equals

with some gotcha's and pitfalls along the way

The contract for equals

- The equals method implements an *equivalence relation* on non-null objects. *Assuming x, y, and z, are all not null:*
- *reflexive*: x.equals(x) == true
- *symmetric*: x.equals(y) == y.equals(x)
- *transitive*:

  if x.equals(y) == true and y.equals(z) == true
  then x.equals(z) == true.
- *consistent*:

  multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in comparisons on the object is modified

- x.equals(null) == false

Directly from: http://docs.oracle.com/javase/8/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());
  }
}
```

Gotcha: *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!
```

Overloading is when there are multiple methods in a class with the same name that take arguments of different types. Java uses the *static type* of the argument to determine which method to invoke.

The type of equals as declared in Object is:

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

The implementation above takes a Point, *not* an Object, so there are two different equals methods in Point!

A Useful Sanity Check

```
public class Point {
  …
  @Override
  public boolean equals(Point that) {
    return (this.getX() == that.getX() &&
            this.getY() == that.getY());
  }
}
```

Optional declaration documents programmer's intent that this method overrides another one

Compilation will yield an error in this case (because this method does *not* override anything from the superclass)

Adding `@Override` here will alert us that there is a problem. Now, how do we fix it??

instanceof

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

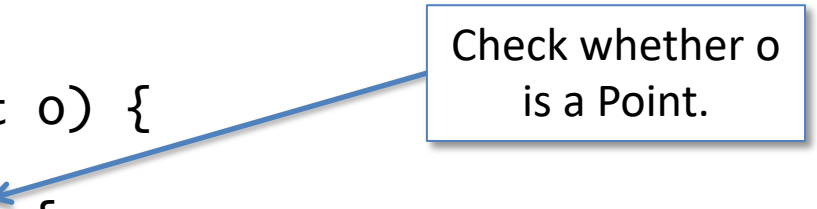
```
Point p = new Point(1,2);
Object o1 = p;
Object o2 = "hello";
System.out.println(p instanceof Point);
       // prints true
System.out.println(o1 instanceof Point);
       // prints true
System.out.println(o2 instanceof Point);
       // prints false
System.out.println(p instanceof Object);
    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

- `null` is *not* an instanceof any class
- But… important to 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) {
  boolean result = false;
     if (o instanceof Point) {
        // o is a point - how do we treat it as such?
     }
   return result;
}
```



- Answer: 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. Instead use generics.

Refining the equals implementation

```
@Override
public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof Point) {
        Point that = (Point) o;
        result = (this.getX() == that.getX() &&
                  this.getY() == that.getY());
    }
    return result;
}
```

This cast is guaranteed to succeed.

Whew. Are we done?

- If we never need to make any subclasses of Point, then yes, this works
  - In particular, this idiom is good enough for the Chat Server homework assignment
- But if we do want to make subclasses of Point, then things get a bit trickier …

```
final class Foo {
        final int x;

        public Foo(int x) {
                this.x = x;
        }

        @Override
        public boolean equals(Object o) {
                if (o instanceof Foo) {
                        return this.x == ((Foo)o).x;
                }
                return false;
        }
}
```

Is this a correct overriding of equals?

1. Yes
2. No

Answer: Yes

- the class is final (cannot be extended)
- the field is immutable, so structural equality is warranted
- equals has the right type and is implemented correctly.

What about Subtyping?

Suppose we extend Point like this…

```
public class ColoredPoint extends Point {
  private final int color;
  public ColoredPoint(int x, int y, int color) {
    super(x,y);
    this.color = color;
  }

  @Override
  public boolean equals(Object o) {
    boolean result = false;
    if (o instanceof ColoredPoint) {
      ColoredPoint that = (ColoredPoint) o;
      result = (this.color == that.color &&
                super.equals(that));
    }
    return result;
  }
}
```

New version of equals is suitably modified to check the color field too

Keyword `super` is used to invoke overridden methods

Broken Symmetry

```
Point p = new Point(1,2);
ColoredPoint cp = new ColoredPoint(1,2,17);
System.out.println(p.equals(cp));
    // prints true
System.out.println(cp.equals(p));
    // prints false
```

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

- The problem arises because we mixed Points and ColoredPoints, but ColoredPoints have more data that allows for finer distinctions.
- Should a Point *ever* be equal to a ColoredPoint?

Suppose Points *can* equal ColoredPoints

```
public class ColoredPoint extends Point {
  …
  public boolean equals(Object o) {
       boolean result = false;
       if (o instanceof ColoredPoint) {
          ColoredPoint that = (ColoredPoint) o;
          result = (this.color == that.color &&
                    super.equals(that));
       } else if (o instanceof Point) {
          result = super.equals(o);
       }
       return result;
  }
}
```

I.e., we repair the symmetry violation by checking for Point explicitly

Now are we good? (1=yes, 2=no)

Broken Transitivity

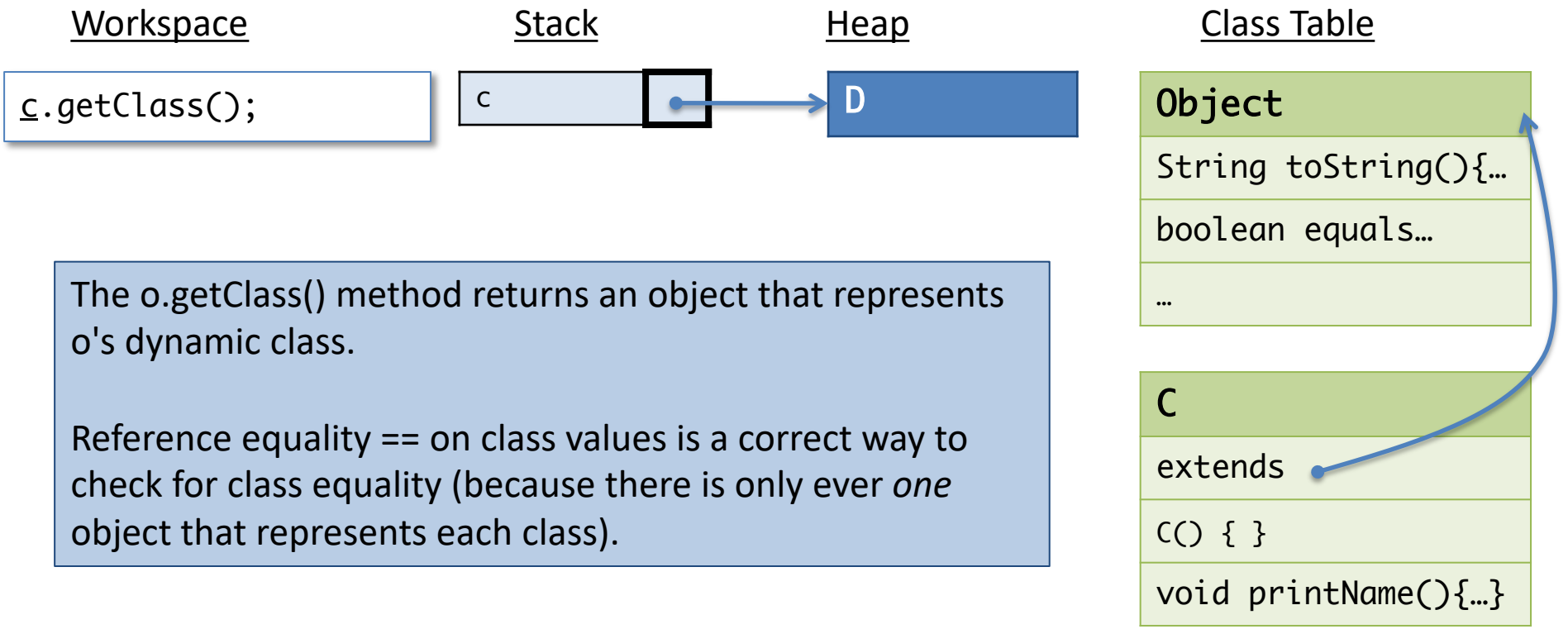
```
Point p = new Point(1,2);
ColoredPoint cp1 = new ColoredPoint(1,2,17);
ColoredPoint cp2 = new ColoredPoint(1,2,42);
System.out.println(p.equals(cp1));
    // prints true
System.out.println(cp1.equals(p));
    // prints true(!)
System.out.println(p.equals(cp2));
    // prints true
System.out.println(cp1.equals(cp2));
    // prints false(!!)
```

- We fixed symmetry, but broke transitivity!
- Should a Point *ever* be equal to a ColoredPoint?

No!

Should equality use `instanceof`?

- To correctly account for subtyping, we need the classes of the two objects to match *exactly*.
- `instanceof` only lets us ask about the subtype relation
- How do we access the dynamic class?



The o.getClass() method returns an object that represents o's dynamic class.

Reference equality == on class values is a correct way to check for class equality (because there is only ever *one* object that represents each class).

Overriding equals, take two

Correct Implementation (for Point)

```
@Override
public boolean equals(Object obj) {
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Point other = (Point) obj;
    return (x == other.x && y == other.y);
}
```

Check whether obj is a Point.

Dynamic cast that checks if obj is a subclass of Point (We know it won't fail.)

Overriding Equality in Practice

- This is all a bit complicated!
- Fortunately, some tools (e.g. Eclipse) can autogenerate equality methods of the kind we developed.
  - Just need to specify which fields should be taken into account.

One more gotcha: Equality and Hashing

- The hashCode method in the class Object is supposed to return an integer value that "summarizes" the entire contents of an object
- Whenever you override equals you should also override hashCode in a compatible way
  - If `o1.equals(o2)` then `o1.hashCode() == o2.hashCode()`
  - hashCode is used by the HashSet and HashMap collections
- Forgetting to do this can lead to extremely puzzling bugs!

Enumerations

Enumerations (a.k.a. Enum Types)

- Java supports *enumerated* type constructors
  - Intended to represent constant data values

```
private enum CommandType {
    CREATE, INVITE, JOIN, KICK, LEAVE, MESG, NICK
  }
```

- Intuitively similar to a simple usage of OCaml datatypes
  - …but each language provides extra bells and whistles that the other does not

Using Enums: Switch

```
// Use of 'enum' in CommandParser.java (PennPals HW)
CommandType t = …

switch (t) {
 case CREATE : System.out.println("Got CREATE!"); break;
 case MESG   : System.out.println("Got MESG!"); break;
 default     : System.out.println("default");
}
```

- Multi-way branch, similar to OCaml's match
  - Works for: primitive data 'int', 'byte', 'char', *etc.*, plus Enum types and String
  - Not as powerful as OCaml pattern matching! (Cannot bind "arguments" of an Enum)
- The `default` keyword specifies a "catch all" (wildcard) case

What will be printed by the following program?

```
Command.Type t = Command.Type.CREATE;

switch (t) {
  case CREATE : System.out.println("Got CREATE!");
  case MESG   : System.out.println("Got MESG!");
  case NICK   : System.out.println("Got NICK!");
  default     : System.out.println("default");
}
```

1. Got CREATE!
2. Got MESG!
3. Got NICK!
4. default
5. something else

Answer: 5 something else!

break

- GOTCHA: By default, each branch will "fall through" into the next, so that code actually prints:

```
Got CREATE!
Got MESG!
Got NICK!
default
```

- Use an explicit **break** statement to avoid fall-through:

```
switch (t) {
case CREATE : System.out.println("Got CREATE!");
              break;
case MESG   : System.out.println("Got MESG!");
              break;
case NICK   : System.out.println("Got NICK!");
              break;
default: System.out.println("default");
}
```

Enums are Classes

- Enums are a convenient way of defining a class along with some standard static methods
  - `valueOf` : converts a `String` to an Enum

    Command.Type c = Command.Type.valueOf ("CONNECT");
  - `values`: returns an `Array` of all the enumerated constants

    Command.Type[] varr = Command.Type.values();

- Implicitly extend class java.lang.Enum
- Can include specialized constructors, fields and methods
  - Example: ServerError

- See Java manual for more

A Useful Trick

```
public enum ServerError {
    OKAY(200),
    INVALID_NAME(401),
    NO_SUCH_CHANNEL(402),
    NO_SUCH_USER(403),
    USER_NOT_IN_CHANNEL(404),
    USER_NOT_OWNER(406),
    …

    private final int value;

    ServerError(int value) {
        this.value = value;
    }

    public int getCode() {
        return value;
    }
}
```

Elements of the enum can be declared along with “parameters”

When the object representing each element is created, the associated parameters are passed to the constructor method.