Programming Languages and Techniques (CIS1200)

Lecture 28

Overriding, Equality

Chapter 26

Announcements (1)

- HW07: PennPals
  - Programming with Java Collections
  - Due Tuesday, November 12 at 11.59pm

- Midterm 2: Friday, November 15th
  - Similar to Midterm 1
  - Content: HW 4 – 6, Chapters 11-21 (Java Arrays) and Chapter 32 (Encapsulation) of lecture notes

Announcements (2)

- Midterm 2: Friday, November 15
  - Coverage: up to Monday, Oct. 28 (Chapters 11-21, 32)
  - During lecture (001 @ 10.15am, 002 @ noon)
    Last names: A – Z Meyerson Hall B1
  - 60 minutes; closed book, closed notes
  - Review Material
    - old exams on the web site ("schedule" tab)
  - Review Session
    - TBA

Review: TreeSet Demo

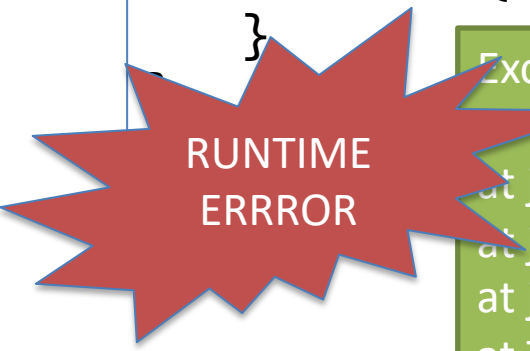
implement Comparable when using SortedSets and Sorted Maps

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<>();
        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?

Dynamic Method Overriding

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

28: What gets printed to the console?

0

I'm a C

0%

I'm a D

0%

NullPointerException

0%

NoSuchMethodException

0%

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

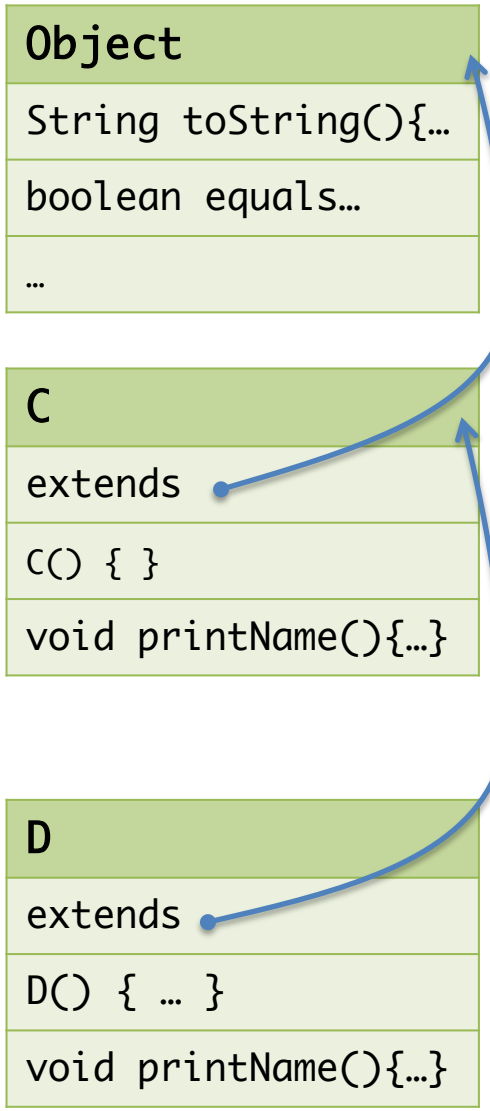
Workspace

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

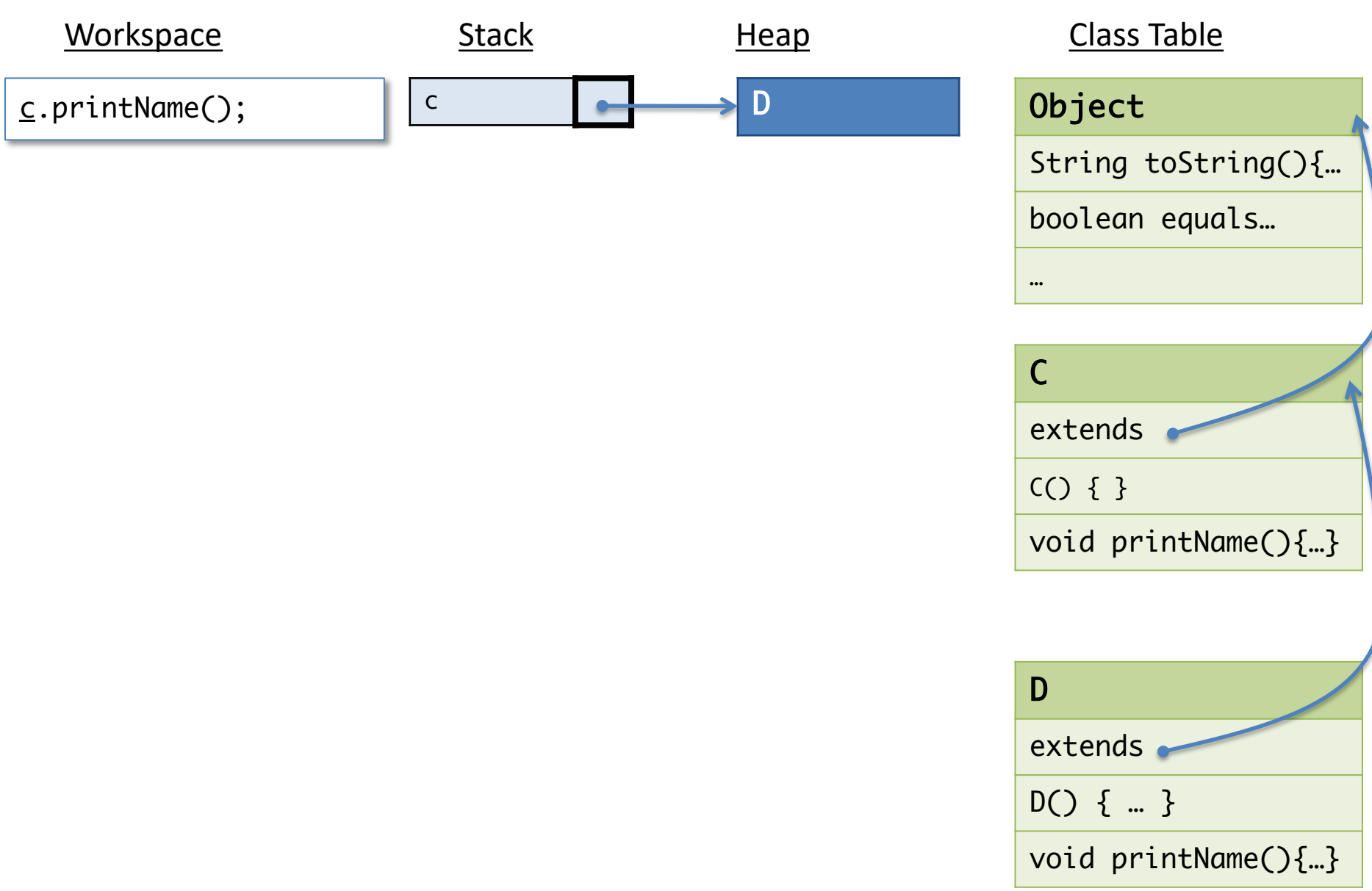
Stack

Heap

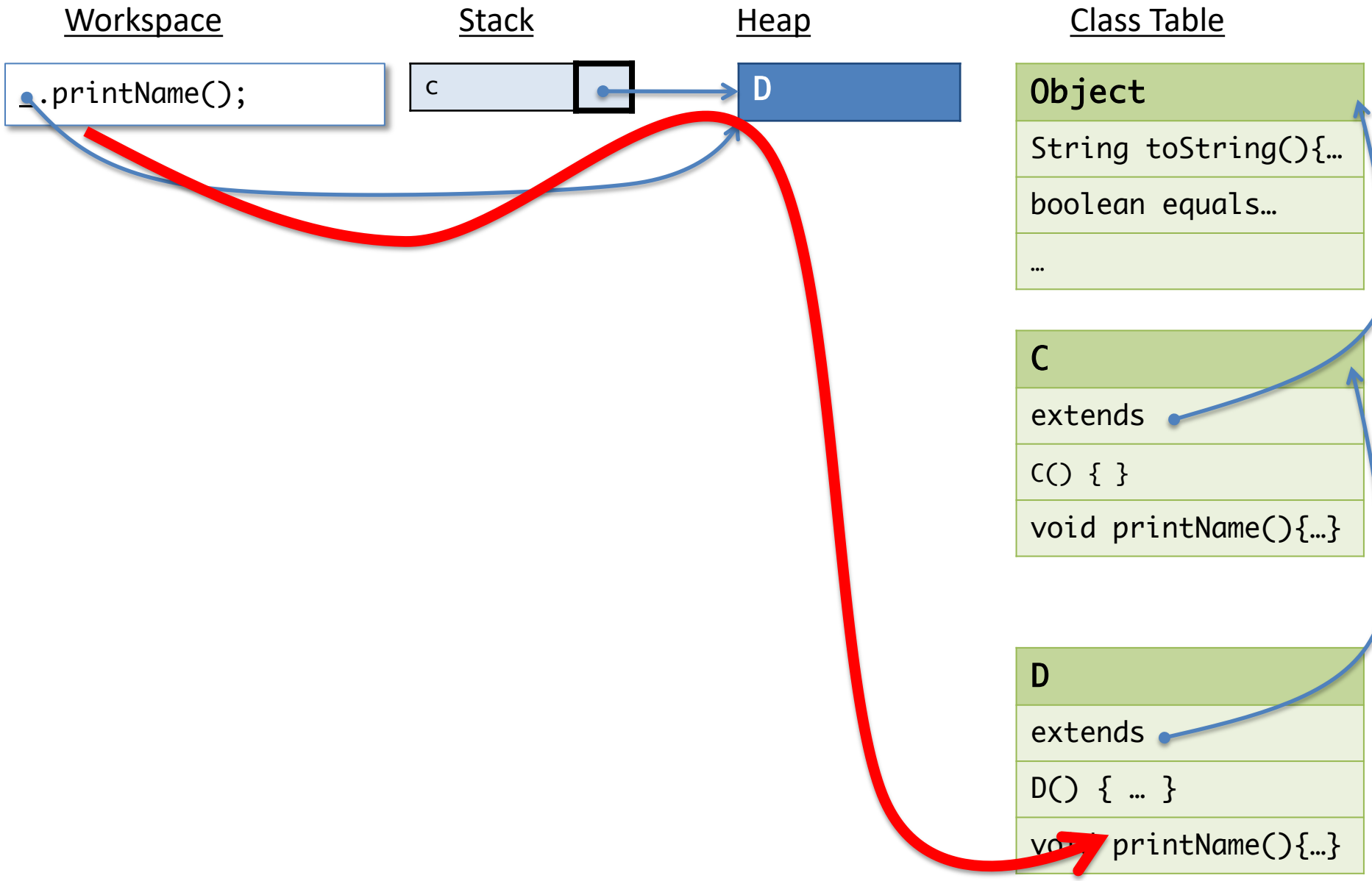
Class Table



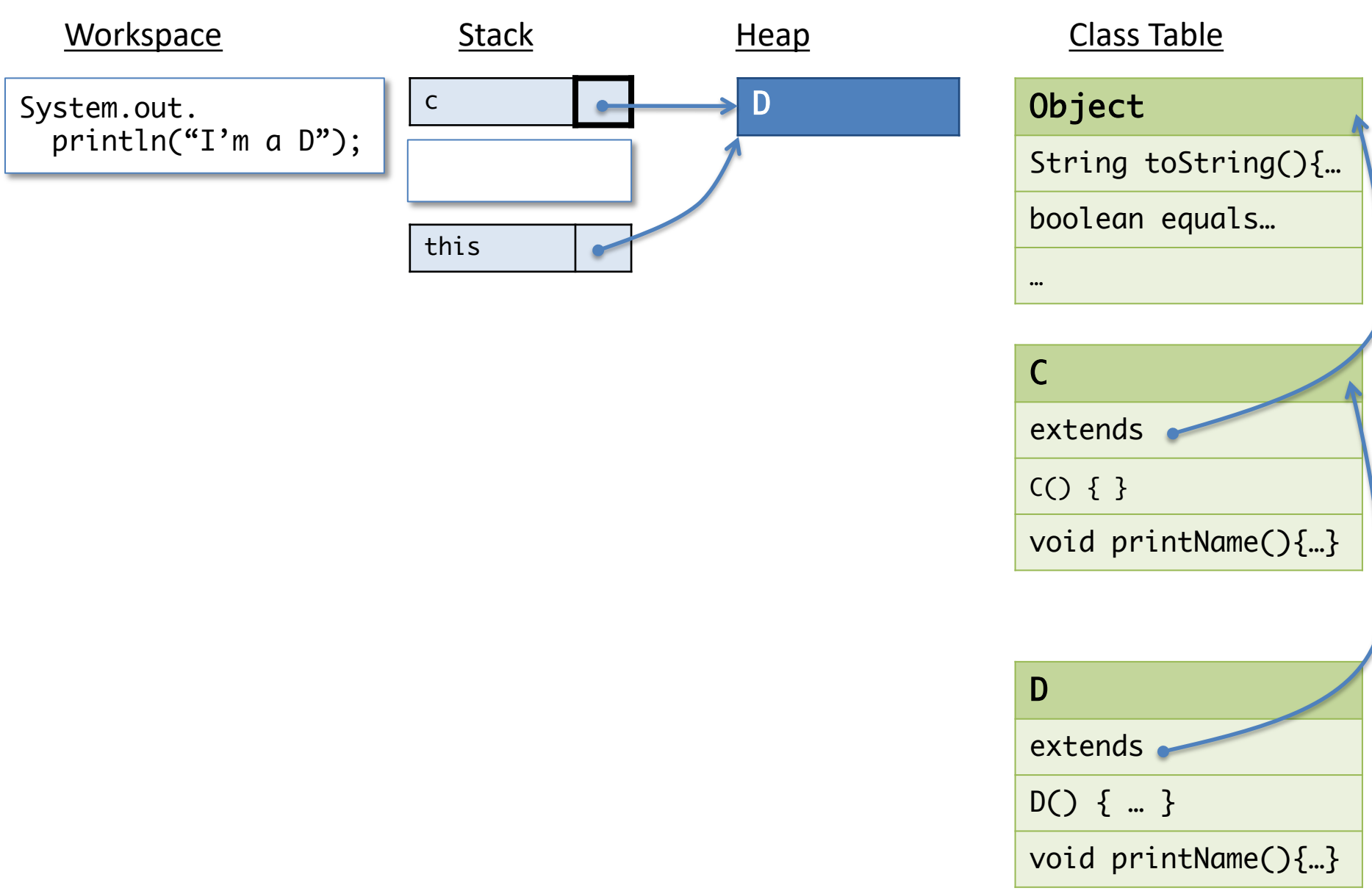
Overriding Example



Overriding Example



Overriding Example



28: What gets printed to the console?

0

I'm a C

0%

I'm a E

0%

NullPointerException

0%

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.

Equality

A case study in overriding

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

Answer: 2

Why?

When to override equals

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

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

- Whenever instances of a class might be used as *elements of a set* or as *keys in a map*
  - Because 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 the mutable internal state might change, the only sensible notion of equality is object identity
  - Also, 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
  - Can be appropriate when a subclass is implemented by adding only new methods, but not fields
  - Use with care: easy to get wrong

How to override equals

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.

Quoted from
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/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());
  }

}
```

Problems:

- equals is *overloaded* not *overridden* (bc. it has the wrong type!)
- if 'that' is null, equals throws an exception

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!
```

The type of equals as declared in Object is:

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

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

Both implementations are now available in class Point; the argument type selects which is used

Overriding equals, take two

Properly overridden equals

```
public class Point {
    …
    @Override
    public boolean equals(Object o) {
        // what do we do here???
    }
}
```

- Use the `@Override` annotation when you *intend* to override a method so that the compiler can warn you about accidental overloading
  - modern IDEs such as IntelliJ will automatically add/suggest these annotations

- 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

Correct Implementation: Point

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

Check whether obj is a Point

"*dynamic cast*" or "*type cast*" or "*downcast*" or "*coercion*"

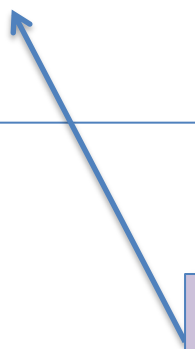
The class cast expression "(T)e" is a runtime test of the dynamic class of of e.
If T is not a subtype of the dynamic class, then a ClassCastException is thrown.
The static type of the expression "(T)e" is T.

Compatibility with compareTo

- For classes that implement the Comparable<E> interface, the equals and compareTo methods should agree:
  - o.compareTo(p) == 0 exactly when o.equals(p)

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass())
      return false;
    Point point = (Point) o;
    return (this.compareTo(point) == 0);

}
```



Implement equals by using compareTo.

Overriding Equality in Practice

- This is all a bit complicated!
- Fortunately, some tools (e.g. IntelliJ) can autogenerate equality methods of the kind we've developed here.
  - Just need to specify which fields should be taken into account.
  - (And you should know why some comparisons use == and some use .equals)

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
  - hashCode is used by the HashSet and HashMap collections

- Whenever you override equals, you should also override hashCode in a compatible way:
  - If `o1.equals(o2)`
    then `o1.hashCode() == o2.hashCode()`

- Forgetting to do this can lead to extremely puzzling bugs!

Equality and Subtypes

What's wrong with using instanceof in the equals method?

*See the 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

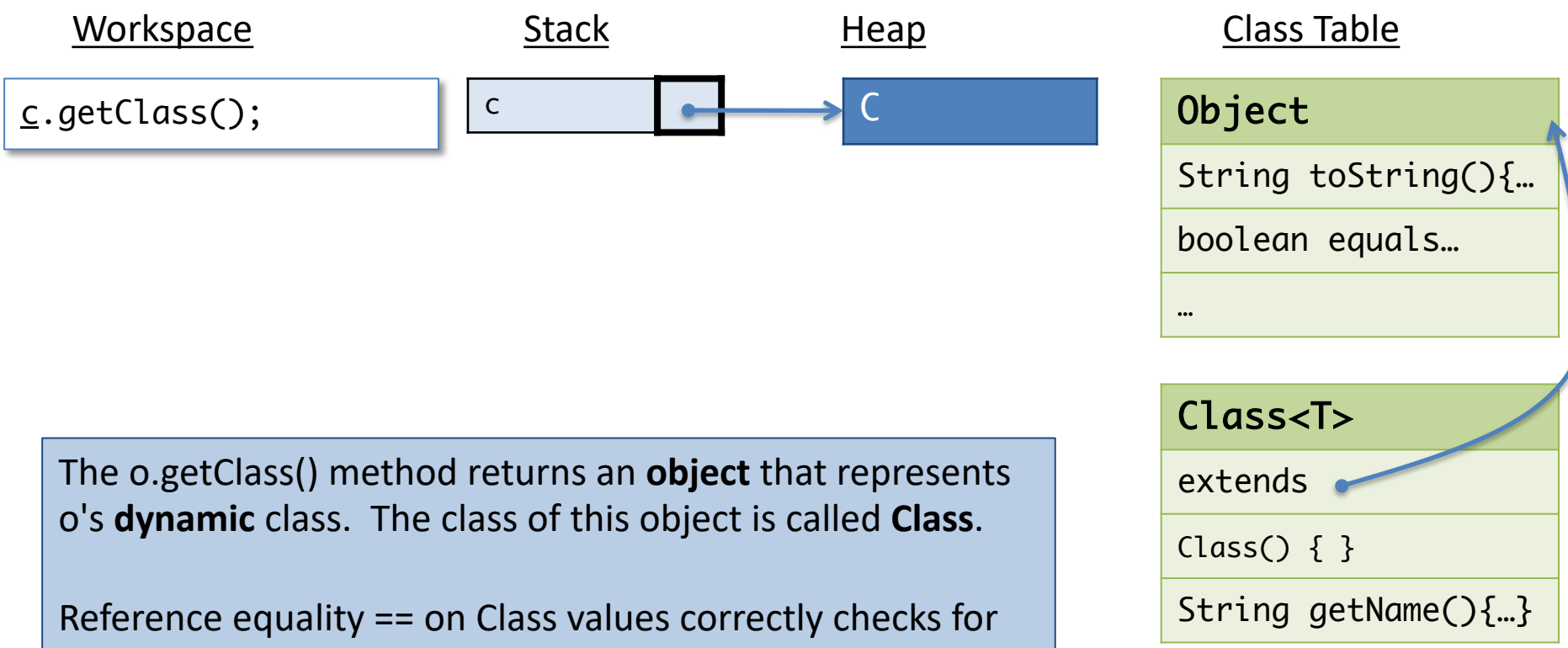
instanceof

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

```
Point p1 = new Point(1,2);
Point p2 = null;
Object o1 = p1;
Object o2 = "hello";
System.out.println(p1 instanceof Point);
       // prints true
System.out.println(o1 instanceof Point);
       // prints true
System.out.println(o2 instanceof Point);
       // prints false
System.out.println(p1 instanceof Object);
       // prints true
System.out.println(p2 instanceof Point);
       // prints false
```

Use `instanceof` judiciously – usually dynamic dispatch is better.

instanceof



The o.getClass() method returns an **object** that represents o's **dynamic** class. The class of this object is called **Class**.

Reference equality == on Class values correctly checks for class equality (i.e. there is only ever *one* object that represents each class).

(Bad!) implementation of equals

- We can test whether o is a Point using instanceof

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

Why do we require the classes of the objects to be exactly the same? What about subtypes?

Should equals use `instanceof`?

- It's tempting to use instanceof in this way, to change the argument of the equals method from Object to Point
- But instanceof only lets us ask about whether an object belongs to some subtype of a given class
- To correctly account for equality in the presence of subtyping, we need the classes of the two objects to match *exactly*

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) {
    if (o instanceof ColoredPoint that) {
      return (this.color == that.color &&
              super.equals(that));
    }
    return false;
  }
}
```

This 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?

- 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) {
    if (o instanceof ColoredPoint that) {
          return (this.color == that.color &&
                  super.equals(that));
        } else if (o instanceof Point p) {
          return super.equals(o);
        }
        return false;
  }
}
```

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

Does this really work?

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!

Correct Implementation: ColoredPoint

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

Check whether obj is a ColoredPoint

"*dynamic cast*" or "*type cast*" or "*downcast*" or "*coercion*"

Iterating over collections

iterators, while, for, for-each loops

Iterator and Iterable

```
interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void delete();   // optional
}
```

```
interface Iterable<E> {
    public Iterator<E> iterator();
}
```

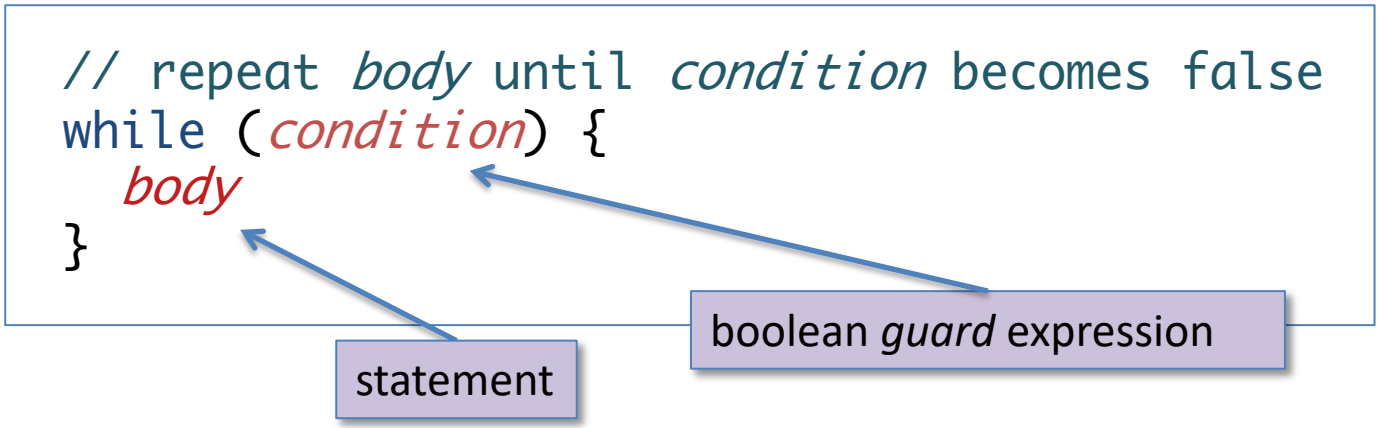
```
interface Collection<E> extends Iterable<E> …
```

Challenge: given a List<Book> object how would you add each book's data to a catalogue using an iterator?

While Loops

syntax:

```
// repeat body until condition becomes false
while (condition) {
   body
}
```



example:

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
Iterator<Book> iter = shelf.iterator();
while (iter.hasNext()) {
    Book book = iter.next();
    catalogue.addInfo(book);
    numBooks = numBooks+1;
}
```

For Loops

syntax:

```
for (init-stmt; condition; next-stmt) {
  body
}
```

equivalent while loop:

```
init-stmt;
while (condition) {
  body
  next-stmt;
}
```

```
List<Book> shelf = …  // create a list of Books

// iterate through the elements on the shelf
for (Iterator<Book> iter = shelf.iterator();
     iter.hasNext();) {
   Book book = iter.next();
   catalogue.addInfo(book);
   numBooks = numBooks+1;
}
```

For-each Loops

syntax:

```
// repeat body for each element in collection
for (type var : coll) {
  body
}
```

element type E

Array of E or instance of Iterable<E>

example:

```
List<Book> shelf = …  // create a list of books

// iterate through the elements on a shelf
for (Book book : shelf) {
    catalogue.addInfo(book);
    numBooks = numBooks+1;
}
```

For-each Loops (cont'd)

Another example:

```
int[] arr = …  // create an array of ints

// count the non-null elements of an array
for (int elt : arr) {
    if (elt != 0) cnt = cnt+1;
}
```

For-each can be used to iterate over arrays or any class that implements the `Iterable<E>` interface (notably `Collection<E>` and its subinterfaces).

Iterator example

```
public static void iteratorExample() {
    List<Integer> nums = new LinkedList<Integer>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int numElts = 0;
    int sumElts = 0;
    Iterator<Integer> iter =
        nums.iterator();
    while (iter.hasNext()) {
      Integer v = iter.next();
      sumElts = sumElts + v;
      numElts = numElts + 1;
    }

    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
  }
```

What is printed by iteratorExample()?

1. sumElts = 0 numElts = 0
2. sumElts = 3 numElts = 2
3. sumElts = 10 numElts = 3
4. NullPointerException
5. Something else

Answer: 3

For-each version

```
public static void forEachExample() {
    List<Integer> nums = new LinkedList<Integer>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int numElts = 0;
    int sumElts = 0;
    for (Integer v : nums) {
      sumElts = sumElts + v;
      numElts = numElts + 1;
    }

    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
  }
```

Another Iterator example

```
public static void nextNextExample() {
    List<Integer> nums = new LinkedList<Integer>();
    nums.add(1);
    nums.add(2);
    nums.add(7);

    int sumElts = 0;
    int numElts = 0;
    Iterator<Integer> iter =
        nums.iterator();
    while (iter.hasNext()) {
      Integer v = iter.next();
      sumElts = sumElts + v;
      v = iter.next();
      numElts = numElts + v;
    }
    System.out.println("sumElts = " + sumElts);
    System.out.println("numElts = " + numElts);
  }
```

What is printed by nextNextExample()?

1. sumElts = 0 numElts = 0
2. sumElts = 3 numElts = 2
3. sumElts = 8 numElts = 2
4. NullPointerException
5. Something else

Answer: 5 NoSuchElementException