Programming Languages and Techniques (CIS120)

Lecture 29

Equality, Enums, and Iteration

Chapters 25 and 26

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)

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

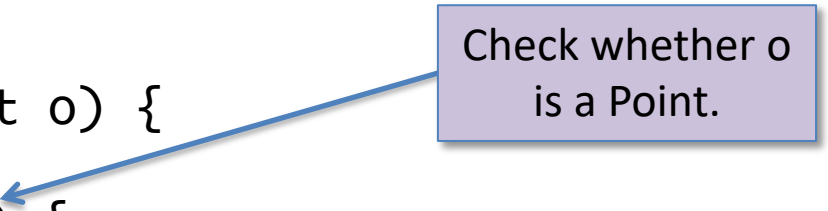
```
Point p = new Point(1,2);
Object o1 = p;
Object o2 = "hello";
System.out.println(p instanceof Point);
       // prints true
System.out.println(o1 instanceof Point);
       // prints true
System.out.println(o2 instanceof Point);
       // prints false
System.out.println(p instanceof Object);
    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

- `null` is *not* an instanceof any type
- 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.

```
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.

Whew. Are we done?

- If we never need to make any subclasses of Point, then yes, this works
- But if we do want to make subclasses of Point, then things get a bit trickier …

SEE CHAPTER 26 OF LECTURE NOTES

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
```

- 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

Broken Transitivity

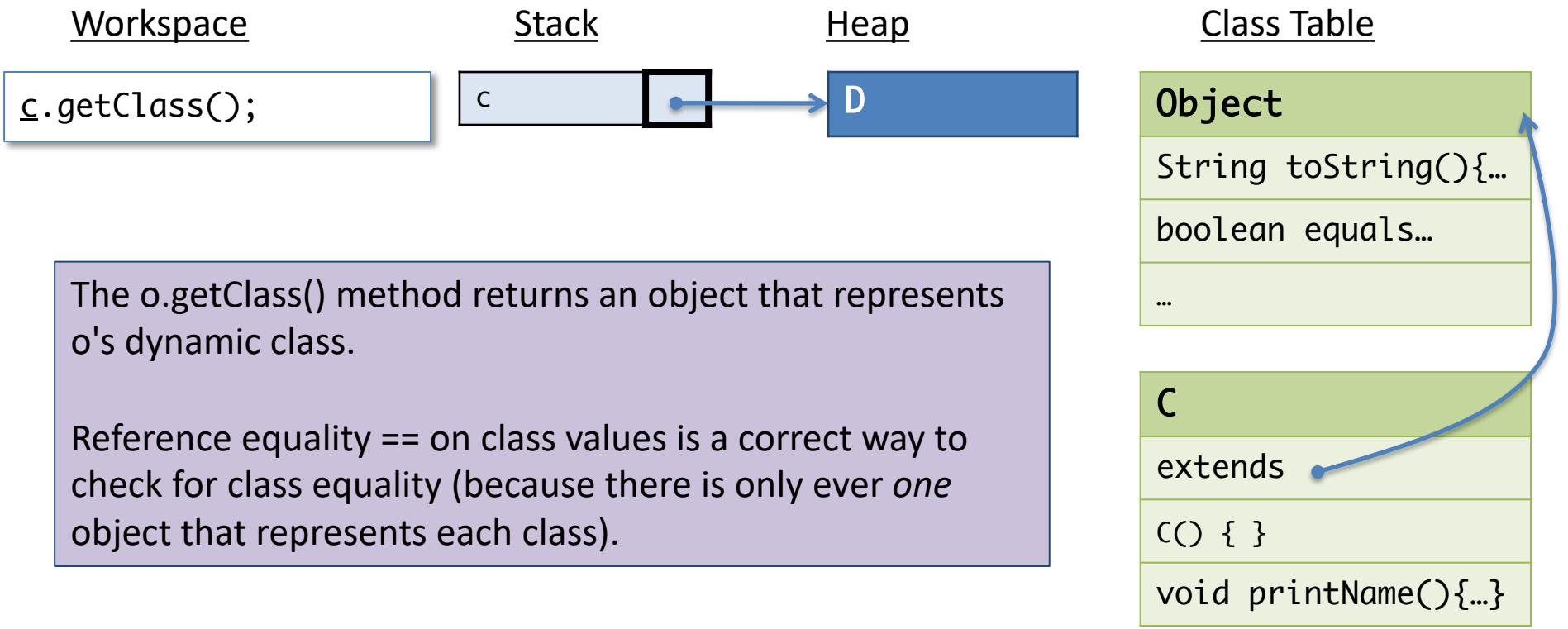
```
Point p = new Point(1,2);
ColoredPoint cp1 = new ColoredPoint(1,2,17);
ColoredPoint cp2 = new ColoredPoint(1,2,42);
System.out.println(p.equals(cp1));
    // prints true
System.out.println(cp1.equals(p));
    // prints true(!)
System.out.println(p.equals(cp2));
    // prints true
System.out.println(cp1.equals(cp2));
    // 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
  - But... be wary that auto-generated code may not do exactly what you want

One more gotcha: Equality and Hashing

- The hashCode method of Object is used by the HashSet and HashMap collections
  - It 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()`

- Forgetting to do this can lead to extremely puzzling bugs...
  - Do not use HashMap or HashSet without understanding hashing!

Enumerations

Enumerations (a.k.a. Enum Types)

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

```
public 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?

```
CommandType t = CommandType.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 ServerResponse {
    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;

    ServerResponse(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.

Iterating over collections

iterators, while, for, for-each loops

Iterator* and Iterable*

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

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

```
interface Iterator<E> {
    public boolean hasNext();
    public E next();
}
```

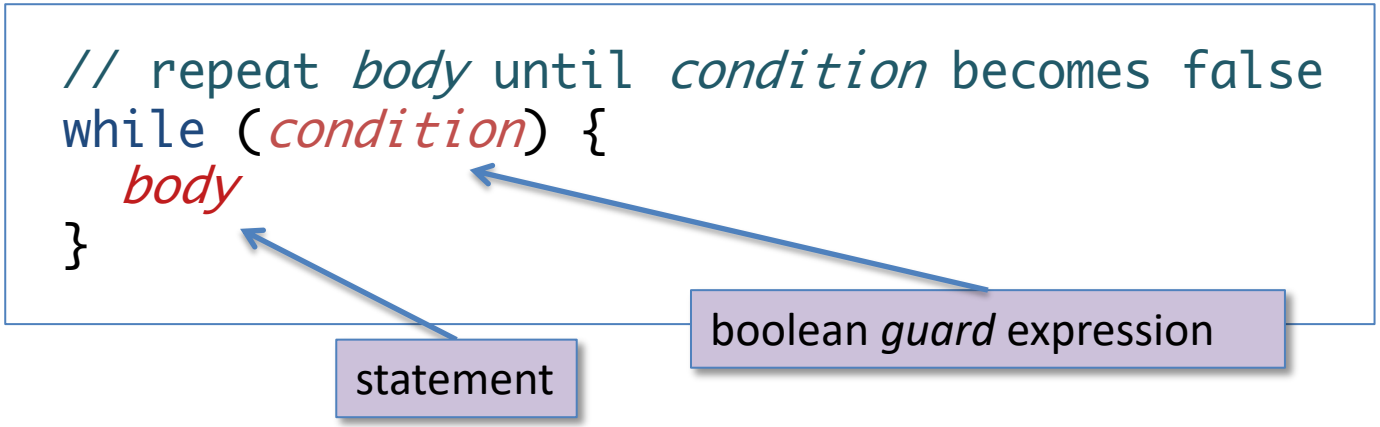
Challenge: given a List<Book> how would you add each book's data to a catalogue using an iterator?

* Required methods shown here. Both interfaces have a few optional methods as well.

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

Some Advice on Debugging

Use the Scientific Method

1. Make an observation / ask a question
   - One of my test cases fails!
   - Which assertion? What exception? What is the stack trace?
2. Formulate a hypothesis
   - Could I have passed null as bar to foo.munge(bar)?
3. Conduct an experiment
   - Modify the program to try to confirm / refute the hypothesis.
   - *Don't* make random changes!
   - You should try to predict the effects of your experiment
   - Re-run test cases
4. Analyze the results
   - Did the modified code behave as expected?
5. Draw conclusions / Report results
   - Create a new test case (if appropriate)

Observing Behavior

- Understand exceptions and the stack trace
  - They give you a lot of information
- If you are using Eclipse, it is worth taking a little time to learn how to use the debugger!
  - See Piazza for a Quick Start tutorial
- Simple print statements are also very effective!
  - Confirm or disprove hypothesis
  - e.g.: The code reached "HERE!" (or not)