Programming Languages and Techniques (CIS120)

Lecture 28
November 7, 2018

Overriding Methods, Equality, Enums, Iterators
Chapters 25 and 26
Announcements

• HW7: Chat Server
  – Available on Codio / Instructions on the web site
  – Due Tuesday, November 13\textsuperscript{th} at 11:59pm

• \textbf{Midterm 2 is this Friday, in class}
  • Last names A – F  Stiteler Hall B21
  • Last names G – S  Leidy Labs 10 (here)
  • Last names T – Z  Fagin 118

• \textbf{Review Session: Wednesday November 7\textsuperscript{th} at 6:00pm, Towne 100}

• Coverage:
  – Mutable state (in OCaml and Java)
  – Objects (in OCaml and Java)
  – ASM (in OCaml and Java)
  – Reactive programming (in Ocaml)
  – Arrays (in Java)
  – Subtyping, Simple Extension, Dynamic Dispatch (in Java)

• Makeup exam request form: on the course web pages
Method Overriding

When a subclass replaces an inherited method by its own definition
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
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();
```

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

Class Table

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

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

D
- extends
- D() { … }
- void printName(){…}
c.printName();

\[\text{Class Table}\]

**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(){
...
...

C

extends

C()

{} {}

void printName(){...}

D

extends

D()

{ ... }

void printName(){...}
System.out.println("I’m a D");

C
extends C()
{
}
void printName(){...}

D
extends D()
{
...
}
void printName(){...}

Class Table

Object
String toString(){...}
boolean equals...
...

C
extends
C()
{
}
void printName(){...}

D
extends
D()
{
...
}
void printName(){...}
```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
try {
    C c = new E();
    c.printName();
} catch (NullPointerException e) {
    System.out.println("I'm a E");
}
```
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
```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();
```

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

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

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

Whoever wrote `E` might not be aware of the implications of changing `getName`. 
Case study: Equality

An example that motivates overriding
Consider this 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)));
```

What gets printed to the console?

1. true
2. false

Why?

Answer: False
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)). ...
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 is implemented by adding only new methods, but not fields
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.

Directly from: [http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html - equals(java.lang.Object)]
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*

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

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

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

```java
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
```

But... use `instanceof` judiciously – usually dynamic dispatch is better.
Type Casts

• We can test whether \( o \) is a Point using `instanceof`

```java
@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;
}
```

• 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

```java
@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.

What about subtypes?
Suppose we extend Point like this:

```java
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;
    }
}
```

This version of `equals` is suitably modified to check the color field too.

Keyword `super` is used to invoke overridden methods.
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?

What gets printed? (1=true, 2=false)
Suppose Points *can* equal ColoredPoints

```java
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.

Does this really work? (1=yes, 2=no)
Broken Transitivity

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

```java
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(!!)
```
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?

```java
C extends C()
{
    void printName(){…}
}
Object
String toString(){…}
boolean equals…
...
D

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

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

• 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.
@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 that checks if obj is a subclass of Point (We know it won't fail.)
Equality and Hashing

- Whenever you override equals you **must also** override hashCode in a compatible way
  - If \( o1 \text{.equals}(o2) \) then
    \[
    o1\text{.hashCode()} == o2\text{.hashCode()}
    \]
  - hashCode is used by the HashSet and HashMap collections

- Forgetting to do this can lead to extremely puzzling bugs!
Overriding Equality in Practice

• Some tools (e.g. Eclipse) can autogenerate equality methods of the kind we developed.
  – But you need to specify which fields should be taken into account.
  – and you should know why some comparisons use == and some use .equals
Enumerations
Enumerations (a.k.a. Enum Types)

- Java supports *enumerated* type constructors
  - Intended to represent constant data values
- Example (from PennPals HW):

```java
public enum ServerError {
    OKAY(200),
    INVALID_NAME(401),
    NO_SUCH_CHANNEL(402),
    NO_SUCH_USER(403),
    ...
    // The integer associated with this enum value
    private final int value;
    ServerError(int value) {
        this.value = value;
    }
    public int getCode() {
        return value;
    }
}
```
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
  – Not pattern matching! (Cannot bind subcomponents of an Enum)

• The default keyword specifies the “catch all” case
What will be printed by the following program?

```java
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!
• **GOTCHA**: By default, each branch will “fall through” into the next, so that code prints:

```java
Got CREATE!
Got MESG!
Got NICK!
```

default

• Use an explicit **break** to avoid fallthrough:

```java
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`
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();
}

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

syntax:

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

example:

```java
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:**

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

- **Array of E or instance of Iterable<E>**
- **element type E**

**example:**

```java
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;
}
```
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).
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
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);
}
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