Enumerations, Overriding Methods, Equality
Chapters 25 and 26
Announcements

• HW7: Chat Server
  – Due Thursday, April 11th at 11:59pm

• HW8: Spellcheck
  – Due Thursday, April 18th at 11:59pm
LIFE AFTER CIS@PENN

APRIL 11, 2019
6PM-9PM

SINGH CENTER FOR NANOTECHNOLOGY

> AN ALUMNI PANEL FOLLOWED BY NETWORKING RECEPTION
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
  - 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?

```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 actually prints:

Got CREATE!
Got MESG!
Got NICK!

default

• Use an explicit `break` statement to avoid fall-through:

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

```java
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.
When a subclass replaces an inherited method with its own re-definition...
What gets printed to the console?

```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();

I'm a C
I'm a D

NullPointerException

NoSuchMethodException

Start the presentation to see live content. Still no live content? Install the app or get help at PollEv.com/app
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. NullPointerExce$$exception
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();

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

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

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

Workspace

Stack

Heap

Class Table

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

```
D
extends D()
{
…
}
void printName(){…}
```

```
Object
String toString(){…
boolean equals...
…
```

```
c.printName();
```
Override Example

Workspace

Stack

Heap

Class Table

Object

String toString(){...

boolean equals...

...

C

extends

c()

{}{}

void printName(){...}

D

extends

d()

{}{}

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

**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() {...}`
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();
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
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 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

```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 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
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, calling x.equals(null) should return false.

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:
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.
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??
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... important to 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;
}
```

- 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.
@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;
}
What about Subtyping?
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;
    }
}
```

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

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

```
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;
    }
}
```

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

```java
C extends C()
{
}
void printName(){…}
```

```
String toString(){…
…
```

```
boolean equals…
```

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).

```
Stack

Heap

Workspace

Class Table
```
Overriding equals, take two
@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);
}
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!