Programming Languages and Techniques (CIS120)

Lecture 28
March 30, 2018

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

When a subclass replaces an inherited method by its own 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
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.
Overriding Example

Workspace

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

Stack

Heap

Class Table

<table>
<thead>
<tr>
<th>Class</th>
<th>Inheritance</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td></td>
<td><code>toString()</code>, <code>equals()</code></td>
</tr>
<tr>
<td>C</td>
<td>extends C()</td>
<td><code>{ }</code>, <code>void printName()</code></td>
</tr>
<tr>
<td>D</td>
<td>extends D()</td>
<td><code>{ }</code>, <code>void printName()</code></td>
</tr>
</tbody>
</table>
Overriding Example

Workspace

Stack

Heap

Class Table

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

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

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

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

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

Workspace

Stack

Heap

Class Table

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

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

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

The Object class implements the .equals method using reference equality (i.e. ==)
We want structural equality for Point in this example
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)
How to override equals
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());
    }
}
Gocha: overloading, vs. overriding

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!

```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!
```

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.
Properly overridden equals

```java
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;
}
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:

```
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`
    ```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`
Iterating over collections

iterators, while, for, for-each loops
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:

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

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;
}
```
For-each Loops (cont’d)

Another example:

```java
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);
}
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. NoSuchElementException
5. Something else

Answer: 5  NoSuchElementException