Announcements

• HW 09 due Tuesday at midnight

• More information about exam 2 available on Friday
Unfinished Business

Histogram.java and WordScanner.java
• Write a command-line program that, given a filename for a text file as input, calculates the frequencies (i.e. number of occurrences) of each distinct word of the file. The program should then print the frequency distribution to the console as a sequence of “word: freq” pairs (one per line).

Histogram result:
The : 1
Write : 1
a : 4
as : 2
calculates : 1
command : 1
console : 1
distinct : 1
distribution : 1
e : 1
each : 1
file : 2
filename : 1
for : 1
freq : 1
frequencies : 1
frequency : 1
given : 1
i : 1
input : 1
line : 2
number : 1
occurrences : 1
of : 4
one : 1
pairs : 1
per : 1
print : 1
program : 2
sequence : 1
should : 1
text : 1
that : 1
the : 4
then : 1
to : 1
word : 2
Method Overriding
A Subclass can **Override** its Parent

```java
public class C {
    public void printName() { System.out.println("I’m a C"); }
}

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

C c = new D();
c.printName(); // what gets printed?
```

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

### Workspace

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Class Table

<table>
<thead>
<tr>
<th>Object</th>
<th>String toString() { ... }</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>boolean equals...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>extends</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C() { }</td>
</tr>
<tr>
<td></td>
<td>void printName() { ... }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>extends</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>D() { ... }</td>
</tr>
<tr>
<td></td>
<td>void printName() { ... }</td>
</tr>
</tbody>
</table>
Overriding Example

Workspace

Stack

Heap

Class Table

```
c.printName();
```

```
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

```
printName();
```

```
C
```

```
D
```

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

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

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

Workspace

System.out.
println("I’m a D");

Stack

Heap

Class Table

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

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

D
extends 
D() { ... }
void printName() { ... }
Dangers of Overriding

```java
public class C {
    Exam exam2 = ...
    public void printTest() {
        if (onDate("March 29th")) {
            System.out.println("as scheduled");
        } else { System.out.println("postponed"); }
    }
    public boolean onDate(String s) {
        return exam2.date().equals(s);
    }
}

public class D extends C {
    Exam final = ...
    public boolean onDate(String s) {
        return final.date().equals(s);
    }
}

C c = new D();
c.printTest(); // what gets printed?
```

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

Whoever wrote D might not be aware of the implications of changing `onDate`.

Overriding the method can cause the behavior of `printTest` to change!
- Overriding can break invariants/abstractions relied upon by the superclass.
When To Override?

• Only override methods when the parent class is designed specifically to support such modifications:
  – If you’re writing the code for both the parent and child class (and will maintain control of both parts as the software evolves) it might be OK to override.
  – If the library designer specifically describes the behavioral contract that the parent methods assume about overridden methods (and the child follows that contract).
  – Either way: document the design.
  – Use the @Override annotation to mark intentional overriding.

• Look for other means of achieving the desired outcome:
  – Use composition & delegation (i.e. wrapper objects) rather than overriding.
The final modifier

• By default, fields and local variables are mutable and methods can be overridden*.
• The final modifier changes that.
• Final fields and local variables:
  – Must be initialized (either by a static initializer or in the constructor) and cannot thereafter be modified.
  – Act like the immutable name bindings in OCaml
  – static final fields are useful for defining constants (e.g. Math.PI)

• Final methods cannot be overridden in subclasses.
  – Also useful in combination with static
  – Prevents subclasses from changing the “behavioral contract” between methods by overriding.

*Technically, fields can also be re-declared in a subclass (i.e. C has field x and D extends C and also declares a field x, not even necessarily of the same type!). Don’t do this! But be aware that you can introduce bugs by inadvertently using this “feature”.
When to override: Equality
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 is printed to the terminal? Why?
Equality*

1. Identity vs. Equality
2. Pitfalls with overriding equals
3. Recipe for overriding equals

*See the very 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
Identity vs. Equality

• Object *identity* is “pointer equality” a.k.a. “reference equality”
  – Indicates where in the heap the object is located
  – Tested using ==

• Object *equality* is “value”, “logical”, “structural” or “deep” equality
  – Indicates when two objects are “the same” as values
  – Tested using the `equals` method inherited from `Object`

• In Java, the default implementation of `equals` is ==
  – In this case, instances are equal only to themselves

• Classes *can* override the default implementation to provide a different “structural” notion of equality.
  – e.g. String tests for identical sequences of characters.
Logical Equality

• What does it mean for two things to be equal?
  – “that depends on what your definition of is is”
  – In what way is the equality being used?

• Answer 1: Mutable objects are (usually) only equal to themselves
  – Why?

• Answer 2: Two immutable objects (of the same type) are equal if their corresponding fields are equal
  – What if there are “unimportant” fields?
  – What if the objects are of different types?

• What is a reasonable definition of equality?
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.