Accepting Reality: Full Java

Corky Cartwright

Stephen Wong

Department of Computer Science

Rice University

What are Language Levels Hiding?

- In principle, nothing…
  Java could have supported a notion of *immutable* classes with essentially the same semantics as the DrJava Functional Level. Scala appears to do this in “variant” classes (concrete classes in an immutable composite)
- But Java is what it is …
- Transforming DrJava IL code to full Java code:
  - Explicit constructors
  - Explicit accessors
  - Explicit overriding of equals
  - Explicit overriding of hashCode()
  - Explicit overriding of toString()

Explicit Constructors

- A constructor definition has the form:

```
<ClassName>(var1, ..., varn) {
  <optional supercall on superclass constructor>
  <code body that initializes instance fields of class>
}
```

- All fields not initialized in explicit constructors are set to the default value for their respective type: `0` for all primitive number/char types, `false` for `boolean` and `null` for all object (reference) types.
- Multiple constructors are permissible (static overloading).
- If no explicit constructors are provided, Java automatically generates a default 0-ary constructor with an empty body.
- A superclass call has the form

  `super(arg1, ..., argm);`

  where the arguments in the superclass call match the signature (parameter declarations) of one the superclass constructors. If a supercall does not appear as the first statement in a constructor, Java automatically generates a superclass call on a 0-ary constructor (which may cause a compiler error).

Explicit Accessors

- An accessor definition is an ordinary instance method definition of the form:
  `<accessorName>() { return <fieldName>; }`
- The choice of `<accessorName>` is arbitrary. I recommend using the corresponding `<fieldName>`. Another common convention is `get<fieldName>`.
- Instance fields should never be `public`.
- Multiple constructors are allowed (static overloading).
- Recall: if no explicit constructors are provided, Java automatically generates a default 0-ary constructor with an empty body.

Explicit Overriding of `equals`

- The `equals` method, which has signature,

  ```
  public boolean equals(Object other);
  ```

  is inherited in any program-defined class from its superclass. In `Object`, `equals` means object identity (same allocation using `new`. This default is almost never the proper definition for an immutable class, but it is usually the right definition for a mutable class.

- In the Java programming culture, the following rule is very widely taught: always override `hashCode`, which has signature:

  ```
  public int hashCode();
  ```

  when you override `equals`. Their meanings purportedly must preserve the following invariant:

  `a.equals(b)` → `a.hashCode() == b.hashCode()`

  This rule is compelling for immutable data but it makes no sense for mutable data. We will discuss this issue in more detail later in the course.

Explicit Overriding of `equals` cont.

- How should we write code to override `equals` an immutable class `C` with fields `f`, `g`, `h`? For the complete answer, look at the `.java` files generated by the DrJava language levels facility. A satisfactory answer in some contexts is the following:

```
public boolean equals(Object other) {
  C o = (C) other;
  return f.equals(other.f) &&
    g.equals(other.g) && h.equals(other.h);
```

- Note: if a field is of primitive type, the proper comparison operator is infix `==` .
- What is potentially wrong with this definition? What happens if we extend class `C`?
- What is fundamentally wrong with using the `==` operator instead of `equals` on object types? Not algebraic (mathematical) equality.

Explicit Overriding of **`hashCode`**

- For immutable classes, the preceding invariant linking **`equals`** and **`hashCode`** is important because hash tables will break if the invariant is violated.
- We will study hash tables later in the course.
- We defer discussing how to properly override **`hashCode`** until then.

Explicit Overriding of `toString`

- The default definition of `toString`, which has signature

  ```
  public String toString();
  ```

  is awful: `<className>@<hashCode>`.
- Why is `toString` important? This representation is used anytime that an object is printed, *e.g*. in many testing contexts.

For Next Class

- Homework for next Monday is posted on the wiki. It consists of doing HW6 (optional) in Java. We have provided you with a purely functional Scheme solution that you must translate to Java using stub code that we have provided.
- For this assignment, the functional language level is your friend.
- If any aspect of Java puzzles you (which is likely!), please ask a question directly of a course staff member or by sending mail to comp211.