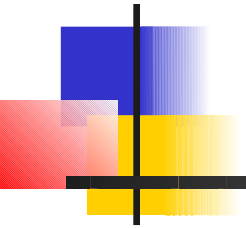
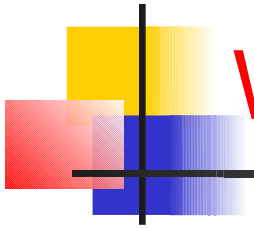


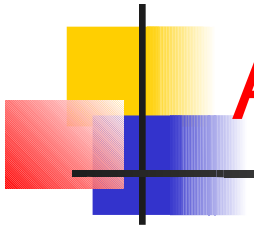
Coping with Reality: Full Java

Corky Cartwright
Department of Computer Science
Rice University



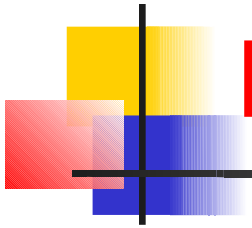
What is Hidden by Language Levels?

- In principle, nothing …
  Java could support the notion of *immutable* classes with essentially the same semantics as the DrJava Intermediate Level.
- But Java is what it is …
- Transforming DrJava IL code to full Java code:
  [Reference: Notes on OO Design, Ch. ??]
  - Explicit constructors
  - Explicit accessors
  - Explicit overriding of equals
  - Explicit overriding of hashCode()
  - Explicit overriding of toString()



Aside: Distinctions Among Equals

- In computing with mutable objects, several different notions of equality are important.
- The `equals` method is the notion of equality that the author defined on objects of the receiver's class. By default (if not overridden), the Java `equals` method behaves exactly like `==` (described below) except when the receiver is `null`. (Every Java object belongs to multiple types but only one class.)
- Java also supports the infix operator `==` which is defined on primitive values (like `int`s) as well as objects. What does `==` check? On objects, whether or not its two arguments are identical objects (same `new` allocation) or both `null`. On primitive values, whether the two values are equal. You cannot compare primitive values and objects (their types are incompatible).
- Where does this distinction bite? When using `==` on objects when `equals` is meant. Java tries to help programmers avoid these bugs on `String`s (by interning all constant `String`s). Demo. See OO Design Notes, ??
- In the Language Levels (immutable) Java subset, we only use `==` on primitive values.

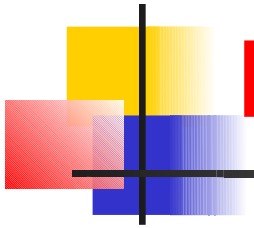


Explicit Constructors

- A constructor definition has the form:

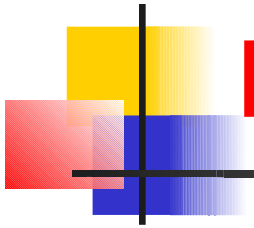
```
<ClassName>(arg1, …, argn) {
  <optional supercall on superclass constructor>
  <code body that initializes instance fields of class>
}
```

- 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.



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`**.



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.

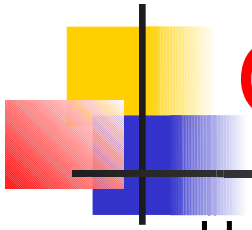
- In the Java programming culture, the following rule is very widely taught: always override `hashCode`, which has signature:

  ```
  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. Why? You should never hash on mutable data using `hashCode`. The Java libraries include `IdentityHashMap` (which hashes on the object address) for this purpose.



Overriding of `equals` cont.

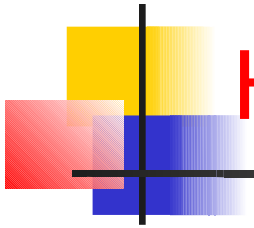
- How should we write code to override `equals` an immutable class `C` with fields `f`, `g`, `h`? For the complete answer, look at the `.java` files generated by the DrJava language levels facility. A satisfactory answer in some contexts is the following:
  - ```
    public boolean equals(Object other) {
      return (other instanceof C) && 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 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 stock invariant linking **`equals`** and **`hashCode`** is critical because hash tables will break if the invariant is violated.

What is a hash table?

- This data structure is provided in several flavors by the **`java.util`** library.
- A hash function maps allocated objects to an **`int`**. Good hash functions almost always map unequal objects to unequal values. The Java **`Object`** class includes the method **`hashCode`**, which computes a value (typically the address of the object when **`hashCode`** is first called) that is different for nearly all objects.
- Hash tables use an object's **`hashCode`** to determine where to place the object (which slot) in an array (the contents of the table). Each slot really corresponds to a short list {typically length 0 or 1) of objects, which must be searched when looking up an object in the hash table. Since two distinct objects can (rarely) have the same **`hashCode`**, hash tables ultimately use the **`equals`** method to determine when objects are distinct (in searching the list of object mapped to the same hash table slot).
- If **`equalsHashMap`** is overridden in a class **`C`**, **`equal`** but different objects (allocations) of class **`C`** may be assigned different **`hashCode`**s, which breaks hash tables (look-ups can fail!), which must map **`equal`** objects to the same hash table slot.



Hash table classes in Java libraries

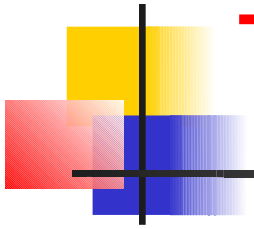
In **`java.util`**, the Java classes **`HashSet<A>`** and **`HashMap<A,B>`** use hash tables to implement sets of type **`A`** and maps from type **`A`** to type **`B`**, respectively. They work just like our **`TreeSet<A>`** and **`TreeMap<A,B>`** classes, except that they do not support operations that depend on an ordering relation on **`A`** (**`Comparable<A>`**).

Exercise: given our **`OOTreeMap<A,B>`**, write **`OOTreeSet<A>`**.

Observation: hash tables provide an efficient implementation of sets and maps even when there is no ordering on the key type.

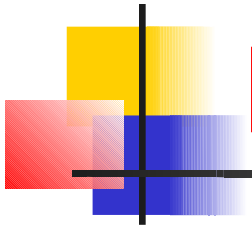
Explicit Overriding of `toString`

- 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 and debugging contexts.
- Should you routinely override `toString`?
  - For data classes, I say yes, because you never know when you will need to print an object when debugging. In addition, it is often more convenient to compare the string representations of objects in testing than it is to test for equality (which mandates overriding `equals`).
  - I recommend against overriding `equals` for mutable data classes. Why? Because it is misleading. There Is no sensible notion of equality on mutable objects (other than `==` which agrees with default definition of `equals)` that works in hash tables (`java.util.IdentityHashMap` uses `==` instead of `equals`). When you write tests using string representations, you realize that you are observing the attributes of an object, not checking for fundamental equality. (Remember to use `toString` explicitly in your tests; otherwise you may get default `equals`. JUnit should generically type the method `assertEquals` but it does not; it will willingly compare a `String` with some other `Object` type.



The Nitty Gritty in HW10

- The assignment is straightforward provided:
  - You are comfortable with full Java.
  - You are comfortable writing visitor classes including anonymous visitor classes
  - You are comfortable using BiList iterators (which are an improvement over the iterators built-in to `java.util`.
  - You can imitate the two forms of tests given the same Junit test suite.
- If you are confused, read Ch. 1 in our Notes on OO Design carefully (particularly 1.10 – 1.13). If you are confused about basic Java operations like `==`, you should read all of Ch.1 carefully and do the interactive finger exercises in DrJava.



For Next Class

- Homework due Friday (but you now have a total of 12 slip days). You need an essentially working HW10 by then even if you plan to use a slip day or two because you need to get started on HW11 which will be posted on Friday. HW12 will be posted a week from Friday.
- Note: Exam2 will be given during our final exam slot on the morning of April 30. Due to honor code issues, take home exams will not be given in Comp 211 for the forseeable future.