Equality and Copying
Context

- Although much of our discussion will use the OpenList class as an example, the concepts extend to a wide variety of other classes.
Equality

Suppose we want to test two OpenLists for equality.

From earlier discussions we could use the following rules, if in rex:

- \( \text{equals}([ ], [ ]) \Rightarrow 1; \)
- \( \text{equals}([ ], _) \Rightarrow 0; \)
- \( \text{equals}(_, [ ]) \Rightarrow 0; \)
- \( \text{equals}([A \mid L], [B \mid M]) \Rightarrow A == B \&\& \text{equals}(L, M); \)
Transcription to Java

- We could easily transcribe those rules using recursion.
- So let’s do it with iteration instead.
public static boolean equals(OpenList L1, OpenList L2) {
    while( L1.nonEmpty() && L2.nonEmpty() ) {
        if( !(L1.first().equals(L2.first())) ) {
            return false;
        }
        L1 = L1.rest();
        L2 = L2.rest();
    }
    return L1.isEmpty() && L2.isEmpty();
}
Comments

● Our version of equals assumes that elements are to be compared using `.equals`, not `==`.

● Does it matter? YES!
  ● `==` compares references only
  ● `.equals` compares content
  ● equal reference `implies` equal content, but NOT conversely
In order to properly over-ride the .equals that is found in the base class Object, we need the following form:

```
equals(Object Ob)
```

rather than the form

```
equals(OpenList L)
```

Otherwise, when comparing Objects that happen to be OpenLists, the wrong equals will get used, and we will end up with ==, which is not what we want.
Nomenclature

- `==` can be said to provide a "shallow" test for equality.

- The intent of `.equals` is to provide a "deep" test for equality.

- What actually happens is up to the programmer or library designer.

- More on this later.
Is there ever really a need to copy an OpenList for sake of just having a copy?
OpenList elements

- Even if OpenList structure is immutable,
- OpenList elements can be any Object, immutable or mutable.
- We can’t stop this, and sometimes it might be useful.
Choices in copying

- Suppose we want to copy an OpenList that contains some mutable Objects.
- There may be preferences involved:
  - We want to make new copies of the mutable objects.
- vs.
  - We want to keep the mutable objects the same in both the original and the copy.
Cloning

- A reasonable convention is that if a Class wants objects to be copiable, it provides a clone() method do so, and conversely.

- The class of things being copied should implement the Cloneable interface.
Recall the Interface Idea

- **Obligations:**
  - A class implementing the Cloneable interface must:
    - Declare that it is implementing the interface.
    - Provide a clone() method.

- **Privileges:**
  - Objects in a class implementing the Cloneable interface can be passed to any method declaring the argument type as Cloneable.
OpenLists of Cloneable’s

- Assume for the moment that we are only going to construct OpenLists out of Cloneable objects.

- Then we can make a clone of an OpenList itself as follows:
Cloning an OpenList: 1st approx.

```java
public Object clone()
{
    if( isEmpty() )
    {
        return nil;
    }
    return cons(first().clone(), (OpenList)rest().clone());
}
```

- Notes:
  - `Object` rather than `OpenList` is the **required** return type for `clone()`.
  - `Object` is not publicly cloneable, so `first().clone()` is illegal.
Fixing the Illegality

Regarding the first() of a list, we’d like to:

- Return a clone if the Object is cloneable.
- Return the next best thing if not:
  - i.e. return the object itself.

This will help make our definition work.

We still need to be careful when using it.
public Object clone()
{
  if( isEmpty() )
  {
    return nil;
  }
  return cons(maybeClone(first()),
              (OpenList)rest().clone());
}
Java Difficulty

- Java provides a way to test if something is cloneable:
  - `Ob instanceof Cloneable`

- Unfortunately, Java provides **no** nice way to **cast** an object known to be Cloneable such the clone() method can be applied. The target of a cast is a class, not an interface.
Object maybeClone(Object Ob)
{
    if( Ob instanceof Cloneable )
        return ((Cloneable)Ob).clone();
    else
        return Ob;
}
A Way I Devised: using java.lang.reflect magic

class Copier
{
    static Class[] noArgs = new Class[0];
    static Object[] args = new Object[0];

    static Object maybeClone(Object ob)
    {
        if( ob instanceof Cloneable )
        {
            try
            {
                // If Cloneable, clone it.

                return ob.getClass().getMethod("clone", noArgs).invoke(ob, args);
            }
            catch( Exception e)
            {
                
            }
        }
        return ob; // Otherwise just return the argument.
    }
}
Possible Approaches to Copying

- `clone()` method: returns copy of `this`

- `copy constructor`: constructs a copy from original

- static `copy method`: copies argument

- Only the first of these really extends robustly across the Java language. The others can be used on an ad hoc basis.
Copy constructor

- **Add constructor**
  
  ```
  MyClass(MyClass orig)
  {
    ...
  }
  ```

- **Use constructor**
  
  ```
  MyClass newObj = new MyClass(orig);
  ```
static copy method

- Add method
  ```java
  static MyClass copy(MyClass orig)
  {
    ...
  }
  ```

- Use method
  ```java
  MyClass newObj = MyClass.copy(orig);
  ```
Think of object as a tree
- The object is the root
- The components are the children.

Deep copy: Copy all the way down to the leaves.

Shallow copy: Copy references to components only.

A spectrum of copies lies in between deep and shallow.
class MyClass
{
    OpenList List1;
    OpenList List2;

    MyClass(MyClass orig)
    {
        List1 = orig.List1;
        List2 = orig.List2;
    }
}

This copy shares the lists with the original.
Deep Copy

- class MyClass
  {
  OpenList List1;
  OpenList List2;
  }

MyClass(MyClass orig)
{
  List1 = orig.List1.clone();
  List2 = orig.List2.clone();
}

- This copy has copies of the original lists, provided that clone() makes such copies.
Equality Revisited

- Defining `equals()` is at programmer’s discretion
- By analogy with copying:
  - Equality checking can be deep, shallow, or in-between.
- **Semantic** equality may be taken into account:
  - e.g. allowing an integer value to be *equal* to a floating value.
Interning principle: Making == function as equals

- Suppose we could guarantee that two objects are semantically equal only if they are the same object.

- Then computing equals would be very fast: only need to compare references.

- Such a guarantee can be made if we intern all of the objects in the class.

- It is common to do this for strings.
Interning principle

- To *intern* objects in a class, we need a way to get to all objects in that class that were ever created.

- The client does not use the constructor, but rather uses a *factory method* that returns objects.

- The factory method checks to see whether the prescribed object has already been created
  - If so, it returns the existing object's reference.
  - If not, it creates a new object (using the constructor) and returns the reference to that object.
Interning Example: an Interning cons

```java
OpenList cons(Object F, OpenList R) {
    OpenList found = find(F, R); // exists already?
    if( found == null ) {
        found = new OpenList(new Cell(F, R));
        remember(found);
    }
    return found;
}
```

Clearly, sharing is intended.

find and remember must be implemented.
Implementing *find* and *remember*

- We must store every OpenList ever produced (sharing tails, etc.)
- We must be able to find a list with equal firsts and rests.
- A naïve implementation would store an internal list of all OpenLists (not itself a OpenList) and search the list with *find*.
- This process can become slow.
Faster
Implementation of find and remember

- Use the concept of hashing.
- Hashing computes a numeric signature for the proposed new item.
- It then uses the signature to access an array (called a hash table) of “equivalence classes” of items.
- Each equivalence class ideally has a relatively small number of items in it.
- The only searching needed is that of searching the small equivalence class, not the whole universe.
How Java Helps

- Java provides method `hashCode()` of `Object`.
- This gives a (generally large) integer value for any object.
- By dividing the hash code by the hash table size, equivalence classes are thereby formed.
- All objects with the same hash code modulo the table size are considered equivalent.
- Java also provides a class `HashSet` that can be used to implement find and remember.
Further Investigation

- Examine these classes/interfaces on the Java API web page:
  - `Exception`
  - `Object`
  - `Cloneable`
  - `HashSet`
  - `Collection`
  - `HashMap`