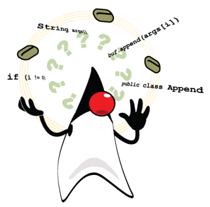

Java Programming 2 – Lecture #14 – Jeremy.Singer@glasgow.ac.uk

Immutable Data Types

Once an immutable object[1] has been constructed, its internal state cannot be modified. Examples from the Java standard libraries include `String` and primitive wrapper classes like `Integer`. Operations on `String` objects like concatenation actually return newly constructed `String` objects and leave the original objects unmodified.[2] There are several key benefits with immutability:

- Immutable objects can be safely shared between threads or data structures.
- Deduplication optimizations can save memory. Two immutable objects with the same field values are effectively indistinguishable and can be mapped onto the same object at runtime.
- Immutable objects are ideal lookup values (keys) in `Map` data structures like hashtables.

Implementing Immutability

To define an immutable data type, all the instance fields need to be `private` and have associated getter methods but no setters. The constructor must set up all the internal state for the object, which cannot be subsequently modified. If any of the instance fields refer to mutable objects, then the associated getter should return a reference to a copy, rather than the original field. Look at the `Person` example below – the names field refers to an `ArrayList`, which is a mutable object so the underlying reference should not be returned directly.

```
public class Person {
  private ArrayList<String> names;

  // check out this interesting varargs syntax!
  public Person(String… names) {
    this.names = new ArrayList<String>();
    for (String name: names) {
      this.names.add(name);
    }
  }
  // returns a copy of the names list, not the
  // underlying reference
  public List<String> getNames() {
    return (List)(this.names.clone());
  }
}
```

[1] See http://www.javapractices.com/topic/TopicAction.do?Id=29 for full details on immutability

[2] This means the use of repeated `String` concatenation to build up a compound `String` is highly inefficient – it is better to use mutable objects like `StringBuffer` – see later.

Copying Objects

There are two ways to create copies of existing objects: copy constructors or clone methods. A copy constructor[3] for a class takes a single parameter, which has the same type as the class. The constructor simply copies the values of the fields from the supplied object into the new object. See the `Pair` example below.

```
public class Pair<T> {
  private T first;
  private T second;
  // copy constructor
  public Pair(Pair<T> other) {
    this.first = other.first;
    this.second = other.second;
  }
}
```

The `clone()` method is a general way of copying Objects. An object may only be cloned if its class implements the `Clonable` marker interface. All objects inherit a `clone()` method from `java.lang.Object` - but invoking this method on a non-`Clonable` object will throw the `CloneNotSupportedException`.

The default clone operation simply instantiates a new object of the appropriate type, and copies the values in the fields across to this new object. This is similar to the copy constructor outlined above. However sometimes this *shallow cloning* is insufficient. If a reference to a mutable object is copied in this way, then the two object will share this reference. In such cases, deep cloning is required. A *deep clone* requires overriding the inherited clone method with a custom method that creates a new object and copies/clones instance fields as appropriate.

Cloning is not recommended as good practice by many Java developers[4] – use of copy constructors appears to be more widely approved and supported.

Mutating Strings

As outlined above, since `String`s are immutable then repeated concatenation is inefficient. It is better to use a mutable class such as a `StringBuffer`[5] to update a character string representation.

```
StringBuffer [] bs = { new StringBuffer("man"),
                       new StringBuffer("califragilistic") };
for (StringBuffer b : bs) {
  b.insert(0, "super");
}
bs[1].append("expialidocious");
```

[3] See http://www.javapractices.com/topic/TopicAction.do?Id=12 for more details
[4] See http://en.wikipedia.org/wiki/Object_copy#In_Java for some reasons.
[5] See http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html