CPEN221 Tutorial 4 - Mutability & Generics

01 - Lab 3 DNA Datatype w/ Cut-And-Splice: Key Lessons + Discussion

Quiz Prep: Do lab 4.2 quiz on

Object .equals(Object o) method

- All classes in java inherit the *Object* class
- Object class has an *equals()* method

Why override *.equals()*? Why not just create a new method?

```
DNA d1 = new DNA("ATG");
Set<DNA> strands = new HashSet<>();
strands.add(d1);
DNA d2 = new DNA("ATG");
boolean exists = strands.contains(d2); // T or F?
```

- How does the HashSet know whether the objects referred to by d1 and d2 are equal or not?
- Other classes access the *equals()* method, so you must override it.
- *assertEquals()* uses the *.equals(Object o)* method, so overloading the method with *.equals(DNA d)* wont work since it is not part of the Object class specification.

If we override *.equals()*, we must also override *.hashCode()*. Using the same example:

```
DNA d1 = new DNA("ATG");
Set<DNA> strands = new HashSet<>();
strands.add(d1);
DNA d2 = new DNA("ATG");
boolean exists = strands.contains(d2); // T or F?
```

- When two objects are the same using *.equals()*, they must also output the same hashCode integer.

Hashing

Collections use hashing to enable quicker O(1) access retrieval.

- **Hash Function** is a *deterministic* mathematical function that maps a large (potentially infinite) set of data into a *finite* range of integers
- To get O(1) retrieval, put the object into a hash function and then it severely limits the possible outcomes

Correct implementation of .equals() and .hashCode() for DNA

```
@Override
public boolean equals(Object o) {
    if (o == null) {
        return false;
    }
    if this == o {
        return true;
    }
    if (!(o instanceof DNA)) {
        return false;
    }
    DNA do = (DNA) o;
    String sequenceMine = this.sequence;
    String sequenceOther = do.sequence;
    return sequenceMine.equals(sequenceOther) && this.mass == o.mass;
}

// Spec: Two equal DNA objects must have equal hashCode integer
@Override
public int hashCode() {
    return this.sequence.hashCode(); // In my lab, I forgot "this." GG
}
```

02 - Mutability

Mutability is a property of a data type, specifically whether the objects of the data type can change after instantiation.

Immutable: "Fields" of objects of this type cannot change after instantiation.

Mutable: Can be changed after instantiation. Any data type that contains references to mutable objects is – by extent – mutable.

In many cases, we prefer immutable types as they make reasoning about correctness easier and prevent unsafe issues due to aliasing.

Aliasing: Multiple references to same object

The issue with aliasing a mutable object is that changing the object for one reference changes the value for all aliases.

General rule of practice:

- Create deep copies if the original datatype contains references to mutable objects
- Shallow copies can be made if the original datatype only contains immutable objects

Example

```
public class myClass {
    public myClass(String mystring) throws IllegalArgumentException {
        this.mystring = mystring;
    }
    private int mymethod(String param) {
        //...
    }
}
```

This class is *immutable* because it only contains references to immutable objects. (String).

03 - Generics

```
class Box<T> {
    private T value;
    public void set(T value) { this.value = value; }
    public T get() { return value; }
}
```

Generics are a mechanism that allows you to parameterize a class or method without knowing the exact type in advance.

```
Box<String> boxStr = new Box<>();
String str = new String("ABC");
boxStr.set(str);
String strInBox = boxStr.get();

Box<DNA> boxDna = new Box<>();
DNA dna = new DNA("TAG");
boxDna.set(dna);
DNA dnaInBox = dnaStr.get();
```

What would we do without generics?

```
class Box {
    private Object contents;
    public void set(Object contents) { this.contents = contents; }
    public Object get() { return this.contents; }
}
```

With generics:

```
class Box<T> {
    private T value;
    public void set(T value) { this.value = value; }
    public T get() { return value; }
}
```

Benefits:

- Gives type safety at compile time (You do not need to)
- Allows you to create classes that take any object

04 - Lab 4 Information

Generics are useful in this lab, since we take a “pair” of any two types of object.