Assignment 5: Design Patterns
Spring 2015
(solution)

Exercise 1

See `http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html`. Java requires that `Object#equals()` implements an equivalence relation on objects such that:

1. It is reflexive: for any non-null reference value `x`, `x.equals(x)` should return `true`.
2. It is symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
3. It is transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then x.equals(z) should return `true`.
4. It is consistent: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.
5. For any non-null reference value `x`, `x.equals(null)` should return `false`.

The definition of equality from the exercise violates symmetricity. Here is a definition that conforms to all contracts.

```
public class A {

    int aField;

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass()) // why not just check for A.class?
            return false;
        A other = (A) obj;
        if (aField != other.aField)
            return false;
```

```
        return true;
    }

    @Override
    public int hashCode() { ... }
}

public class B extends A {

    int bField;

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!super.equals(obj))
            return false;
        if (getClass() != obj.getClass()) // why not just check for B.class?
            return false;
        B other = (B) obj;
        if (bField != other.bField)
            return false;
        return true;
    }

    @Override
    public int hashCode() { ... }
}
```

Exercise 2

Mostly taken from BalusC' answer at `http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns`

Creational (abstract factory, builder, singleton, static factory method)

1. Singleton
   (a) `java.lang.Runtime`
   (b) `java.lang.Desktop`
2. Builder
   (a) `com.google.common.collect.MapMaker`
3. Static factory method
   (a) `java.util.Calendar`
   (b) `java.text.NumberFormat`
   (c) `java.nio.charset.Charset`
4. Abstract factory

(a) `javax.xml.parsers.DocumentBuilderFactory`
(b) `javax.xml.transform.TransformerFactory`
(c) `javax.xml.xpath.XPathFactory`

Structural (adapter, decorator, flyweight)

1. Flyweight
   (a) `java.lang.Integer`
   (b) `java.lang.Boolean`
2. Adapter
   (a) `java.io.InputStreamReader`
   (b) `java.io.OutputStreamWriter`
   (c) `java.util.Arrays`
3. Decorator
   (a) `java.io.BufferedInputStream`
   (b) `java.io.DataInputStream`
   (c) `java.io.BufferedOutputStream`
   (d) `java.util.zip.ZipOutputStream`
   (e) `java.util.Collections#checkedList()`

Behavioural (chain of responsibility, command, iterator, observer, strategy, template method)

1. Chain of responsibility
   (a) `javax.servlet.FilterChain`
2. Command
   (a) `java.lang.Runnable`
   (b) `java.util.concurrent.Callable`
3. Iterator
   (a) `java.util.Iterator`
4. Strategy
   (a) `java.util.Comparator`
   (b) `javax.servlet.Filter`
5. Template method
   (a) `java.util.AbstractList`, `java.util.AbstractSet`, `java.util.AbstractMap`
   (b) `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader`, `java.io.Writer`
6. Observer
   (a) `java.util.EventListener`
   (b) `java.util.Observer`/`java.util.Observable`

Exercise 3

Returning a specific response depends on knowledge of the actual class that implements that kind of response. If a new implementation class needs to be used later:

1. Either the clients of the hierarchy need to be changed to use the new class.
2. Or else, the old classes have to become proxies for the new ones.

The first point is clearly undesirable, while the second one keeps an API that is no longer relevant, thus bloating the code base.

```
public class Response {
    ...  // constructor and accessors omitted for clarity
    private String status;
    private Map<String, String> headers;
    private String body;
}

public class Responses {
    public static Response response(String status, Map<String, String> headers, String body) {
        return new Response(status, headers, body);
    }

    public static Response file(String status, String path) {
        Path filePath = Paths.get(path);
        HashMap<String, String> headers = new HashMap<String, String>();
        headers.put("content-type", Files.probeContentType(filePath));
        byte[] bytes = Files.readAllBytes(filePath);
        String body = new String(bytes);
        return response(status, headers, body);
    }

    public static Response notFound() {
        return file("404", app.Assets.getInstance().getNotFoundPage());
    }

    public static markdown(String body) {
        HashMap<String, String> headers = new HashMap<String, String>();
        headers.put("content-type", "text/html");
        return response("200", Markdown.parse(body).toHtml());
    }
}
```

This design has its defficiencies, and the challenge addresses some of them.