Assignment 5: Design Patterns
Spring 2015

Exercise 1

Here is one particular way to take into account ineheritance when overriding Java's `Object#equals()`:

```
public class A {

    int aField;

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (!(obj instanceof A))
            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 (!(obj instanceof B))
            return false;
        B other = (B) obj;
        if (bField != other.bField)
            return false;
        return true;
    }

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

Look at the contracts of `Object#equals()` and identify which of them are broken. Come up with a definition of equality that works with subclassing. The might be more than one solution. Come up with an implementation of your definition of equality.

Exercise 2

Examine the listed Java APIs and identify some of the design patterns present. For each pattern found describe why the API follows it. Each bullet-point group corresponds to one pattern to be identified. Note that most of the patterns have not been covered on lectures. You may need to look them up on the web.

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

1. (a) `java.lang.Runtime`
   (b) `java.lang.Desktop`
2. (a) `com.google.common.collect.MapMaker`
3. (a) `java.util.Calendar`
   (b) `java.text.NumberFormat`
   (c) `java.nio.charset.Charset`
4. (a) `javax.xml.parsers.DocumentBuilderFactory`
   (b) `javax.xml.transform.TransformerFactory`
   (c) `javax.xml.xpath.XPathFactory`

Structural (adapter, decorator, flyweight)

1. (a) `java.lang.Integer`
   (b) `java.lang.Boolean`
2. (a) `java.io.InputStreamReader`
   (b) `java.io.OutputStreamWriter`
   (c) `java.util.Arrays`
3. (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. (a) `javax.servlet.FilterChain`
2. (a) `java.lang.Runnable`
   (b) `java.util.concurrent.Callable`
3. (a) `java.util.Iterator`
4. (a) `java.util.Comparator`
   (b) `javax.servlet.Filter`
5. (a) `java.util.AbstractList`, `java.util.AbstractSet`, `java.util.AbstractMap`
   (b) `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader`, `java.io.Writer`
6. (a) `java.util.EventListener`
   (b) `java.util.Observer`/`java.util.Observable`

Exercise 3

A web application can return one of many kinds of HTTP responses to the user-agent.

```
public interface Response {
    String getStatus();
    Map<String, String> getHeaders();
    String getBody();
}

public class FileResponse implements Response {
    public FileResponse(String path) {
        this.path = Paths.get(path);
    }

    @Override
    public String getStatus() {
        return "200";
    }

    @Override
    public Map<String, String> getHeaders() {
        HashMap<String, String> headers = new HashMap<String, String>();
        headers.put("content-type", Files.probeContentType(path));
        return headers;
    }

    @Override
    public String getBody() {
        byte[] bytes = Files.readAllBytes(path);
        String body = new String(bytes);
    }

    private Path path;
}

public class NotFoundResponse extends FileResponse {
    public NotFoundResponse() {
        super(app.Assets.getInstance().getNotFoundPage());
    }

    @Override
    public String getStatus() {
        return "404";
    }
}

public class MarkdownResponse implements Response {
    public MarkdownResponse(String body) {
        this.body = body;
    }

    @Override
    public String getStatus() {
```

```
        return "200"
    }

    @Override
    public Map<String, String> getHeaders() {
        HashMap<String, String> headers = new HashMap<String, String>();
        headers.put("content-type", "text/html");
        return headers;
    }

    @Override
    public String getBody() {
        return Markdown.parse(body).toHtml();
    }

    private String body;
}
```

And so on. If an application would like to return a "404 Not Found" response it would do something like:

```
return new NotFoundResponse();
```

The subclasses of `Response` do not extend its interface and therefore they are just an implementation detail. In order to improve the maintainability, your task is to

1. Identify the coupling in the usage of this hierarchy.
2. Try to remedy the coupling problems by applying the static factory method pattern.
3. Replace the hierarchy of classes with a single implementation representing all possible responses.

Challenge

Think about how to allow response-dependent optimizations. One such optimisation would be to enable servers to use the zero-copy 'sendFile' call

```
public static native void sendFile(File file, Socket socket);
```

in order to feed any file specified by an application directly into the socket instead of copying the file into user-space memory first.