Libraries/Testing/Patterns in Java

Libraries

- Included in your project via jar files
- Use a build tool or the jar executable (included in JDK) to create a jar file of your project
- To include a library jar add it on the classpath
  - See example
- Many (many) libraries available for Java.
  - Search online -> maven central repository is best source for quality libraries
  - If the problem is encountered by more than one person there's probably a library available (so search before creating your own)
- We'll cover Guava, Jackson, Mockito and Junit today

Guava

- Provides a large amount of supplemental “core” Java type functionality
  - New Collections
  - Caching
  - String utilities
- See https://code.google.com/p/guava-libraries/wiki/GuavaExplained
- Examples!

Guava - Immutable Collections

```
1  public class Immutables {
2
3      public static void main(String[] args) {
4
5          Set<String> days = ImmutableSet.of("Sunday", "Monday", "Tuesday", "Wednesday",
6                  "Thursday", "Friday", "Saturday");
7
8          Map<String, Integer> primaryColors = ImmutableMap.<String, Integer>builder()
9                  .put("Red", 0xFF0000)
10                 .put("Yellow", 0xFFFF00)
11                 .put("Blue", 0x0000FF).build();
12
13         List<String> mutableSeasons = new ArrayList<>(4);
14         mutableSeasons.add("Autumn");
15         mutableSeasons.add("Winter");
16         mutableSeasons.add("Spring");
17         mutableSeasons.add("Summer");
18
19         List<String> seasons = ImmutableList.copyOf(mutableSeasons);
20
21     }
22
23 }
```

Guava - Collections

- See https://code.google.com/p/guava-libraries/wiki/NewCollectionTypesExplained for all types

```
1  public class Collections {
2
3      public static void main(String[] args) {
4
5          Multimap<String, Integer> homeworkGrades = new HashMultimap<>();
6
7          homeworkGrades.putItem("Brian", 88);
8          homeworkGrades.putItem("Brian", 90);
9          homeworkGrades.putItem("Brian", 77);
10         homeworkGrades.putItem("Brian", 95);
11         homeworkGrades.putItem("Amey", 95);
12         homeworkGrades.putItem("Amey", 99);
13         homeworkGrades.putItem("Amey", 92);
14
15         Multiset<String> wordCounts = HashMultiset.create();
16         wordCounts.add("Age of Innocence", 12);
17         wordCounts.add("Moby Dick");
18         wordCounts.add("Age of Innocence");
19         wordCounts.add("Tale of Two Cities");
20
21         BiMap<String, String> names = HashBiMap.create();
22         names.put("Brian", "Langel");
23
24         // names.keySet -> {"Brian"}
25         BiMap<String, String> inverted = names.inverse();
26         // inverted.keySet -> {"Langel"}
27     }
28
29 }
```

Guava - Caching

```
1  public class Caches {
2
3      public static void main(String[] args) {
4
5          CacheLoader<String, String> loader = new CacheLoader<String, String>() {
6              @Override public String load(String key) throws Exception {
7                  return Caches.load(key);
8              }
9          };
10
11         // max - sized
12         LoadingCache<String, String> cache = CacheBuilder.newBuilder().maximumSize(1000L).build(loader);
13
14         // expire after access
15         cache = CacheBuilder.newBuilder().expireAfterAccess(12, TimeUnit.DAYS).build(loader);
16
17         // expire after write
18         cache = CacheBuilder.newBuilder().expireAfterWrite(12, TimeUnit.DAYS).build(loader);
19
20     }
21
22     private static String load(String key) {
23         // TODO - load from DB / over the network
24         return key;
25     }
26
```

Guava - String Utilities

- See here for reasoning/examples:
  https://code.google.com/p/guava-libraries/wiki/StringsExplained#Charsets

```
1  public class Strings {
2
3      public static void main(String[] args) {
4
5          Joiner joiner = Joiner.on(", ").useForNull("-");
6          String result = joiner.join("Brian", "John", null, "Reggie");
7          assert result.equals("Brian, John, -, Reggie");
8
9          joiner = Joiner.on(", ").skipNulls();
10         result = joiner.join("Brian", "John", null, "Reggie");
11         assert result.equals("Brian, John, Reggie");
12
13         Iterable<String> split = Splitter.on(",")
14                 .omitEmptyStrings()
15                 .trimResults().split(result);
16         for (String value : split) {
17             System.out.printf("%s%n", value);
18         }
19     }
20
```

Jackson - Easy JSON

- See here for information / examples: https://github.com/FasterXML/jackson

```
public class Person {

    private final String firstName;

    private final String lastName;

    private Person() {
        this(null, null);
    }

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    @Override public boolean equals(Object o) {
```

```
public class Jacksons {

    public static void main(String[] args) throws IOException {
        ObjectMapper mapper = new ObjectMapper();

        Person person = new Person("Brian", "Langel");
        String value = mapper.writeValueAsString(person);

        System.out.printf("%s%n", value);

        Person deserialized = mapper.readValue(value, Person.class);
        assert "Brian".equals(deserialized.getFirstName());
        assert "Langel".equals(deserialized.getLastName());
    }

}
```

Testing (unit testing in particular)

- Good developers always unit test.
- Testing code by units (i.e. methods) makes reasoning about the correctness of your code much easier.
- Testing ensures you've correctly created working code
  - I.e., "A Person object can be serialized and deserialized by Jackson" -> make a unit test to ensure this.
- Testing also ensures the integrity of your assumptions across time.
  - I.e., "No one should pass null to this method" -> make a unit test to ensure this.
- Testing also allows you to refactor with confidence.
  - I.e., "I want to rewrite this method but am unsure if it'll break other portions of the code base" -> if everything has unit tests you can refactor with confidence.

JUnit - Framework for Unit Testing

- Only going to cover version 4 and above which leverages annotations.

```
1  public class PersonTest {
2
3      @Test
4      public void json() throws IOException {
5          ObjectMapper mapper = new ObjectMapper();
6          Person person = new Person("Brian", "Langel");
7          String serialized = mapper.writeValueAsString(person);
8          assertEquals("{\"firstName\":\"Brian\",\"lastName\":\"Langel\"}", serialized);
9
10         Person deserialized = mapper.readValue(serialized, Person.class);
11         assertEquals("Brian", deserialized.getFirstName());
12         assertEquals("Langel", deserialized.getLastName());
13     }
14
15 }
```

JUnit

- By convention, put test classes in `src/test`
- By convention, name the test class `ClassNameToBeTestedTest`
  - The class name with the `Test` suffix
- By convention, write a unit test for every public method
  - Also a good practice to test private/protected/default methods as well - necessitates using reflection
- By convention, name the unit test method the same as the method being tested.
- To mark a method as a testing method annotate it with `@Test`
- To invoke code prior to every test, make a method and annotate it with `@Before`
- To invoke code after every test, make a method and annotate it with `@After`
- Can ignore tests by annotating the method with `@Ignore`

Better Unit Testing - Mockito

- Often when unit testing you'll want to "mock" the implementation of a class so that you can test all scenarios.
  - For instance, how to test this class?

```
1  public class Jsons {
2
3      private final ObjectMapper mapper;
4
5      public Jsons(ObjectMapper mapper) {
6          this.mapper = mapper;
7      }
8
9      public void serialize(Object value, FileOutputStream stream) {
10         try {
11             mapper.writeValue(stream, value);
12         } catch (IOException ioe) {
13             throw new RuntimeException(ioe);
14         }
15     }
16
17     public <T> T deserialize(Class<T> type, FileInputStream stream) {
18         try {
19             return mapper.readValue(stream, type);
20         } catch (IOException ioe) {
21             throw new RuntimeException(ioe);
22         }
23     }
24
```

Mockito - Mocking Objects

```java
public class JsonsTest {

    @Test
    public void serialize() throws IOException {
        ObjectMapper mapper = mock(ObjectMapper.class);
        Jsons jsons = new Jsons(mapper);

        FileOutputStream stream = mock(FileOutputStream.class);
        Person person = new Person("Brian", "Langel");

        doThrow(new IOException()).when(mapper).writeValue(eq(stream), eq(person));

        try {
            jsons.serialize(person, stream);
            fail("Expecting a RuntimeException as ObjectMapper should throw an IOException");
        } catch (RuntimeException re) {
            // expected
        }

        verify(mapper, times(1)).writeValue(eq(stream), eq(person));
        verifyNoMoreInteractions(mapper);
    }

    @Test
    public void deserialize() throws IOException {
        ObjectMapper mapper = mock(ObjectMapper.class);
        Jsons jsons = new Jsons(mapper);

        FileInputStream stream = mock(FileInputStream.class);

        when(mapper.readValue(eq(stream), eq(Person.class))).thenThrow(new IOException());

        try {
            jsons.deserialize(Person.class, stream);
            fail("Expecting a RuntimeException as ObjectMapper should throw an IOException");
        } catch (RuntimeException re) {
            // expected
        }

        verify(mapper, times(1)).readValue(eq(stream), eq(Person.class));
        verifyNoMoreInteractions(mapper);
    }

```

Mockito (cont)

- See https://code.google.com/p/mockito/ for more examples
- Most important methods are
  - mock -> given a class returns a “mocked” object of that type
  - when -> given a mock will allow you to control the behavior
  - verify -> verifies that the expected method calls happened on the object
- Keep in mind that final classes and final methods **cannot** be mocked
  - This is an artifact of how mockito leverages the Java language to perform it’s “magic”
    - as it uses Reflection, subclassing and proxies

Common Design Patterns in Java

- Factory Pattern
  - Have used this already in homeworks
- Builder Pattern
  - Have seen this in the mockito / guava code from last lecture
- Delegate Pattern
  - Also known as composition
- Decorator Pattern
  - Particular form of Delegate Pattern
- Singleton Pattern
  - Careful! Often done in a non-Thread safe manner

Factory Pattern

- A factory class decouples the user from the implementing classes.
- Used in creation of objects
  - Like a constructor but with additional flexibility
    - Saw this in the Factory objects you've created in homework assignments
  - Immutable objects can be cached more easily with the factory pattern
- Form of encapsulation
  - Can leverage the Factory pattern to allow construction of objects
- Example!

Builder Pattern

- A builder is used to construct complicated objects in steps
- Extremely useful for creating immutable objects were immutability can only happen after a number of steps are performed
  - E.g.; see the builder for ImmutableCollection objects in Guava
- Composed of “chaining” methods which end in a final call to a method returning the type in question; usually named **build**()
- More complicated builder patterns can exist if certain steps should happen before others, this works by composing two or more builder classes together
- Example!

Composite/Delegate Pattern

- Known generally as the composite pattern
- The delegate pattern is when desired functionality is “delegated” to another object which has already implemented the functionality
  - You’ve seen this when implementing the Multimap homework. The desired functionality was the Map interface and you delegated the implementation to a HashMap or TreeMap
- You can “compose” multiple delegates to create a very complicated set of functionality for a single class without that class needing to implement all the logic itself
  - This is typically done with interfaces. A class, Foo, implements one to many interfaces. The implementation of those interfaces is provided by Foo via other objects already implementing the interfaces.
- Example!

Decorator Pattern

- The Decorator pattern is a dynamic way to add behavior to an entity at runtime by using the Delegate pattern
- Decorators all implement a common interface. They then use the delegate pattern to combine/concatenate their work
- We've seen this already with the InputStream/OutputStream
  - BufferedInputStream can decorate a FileInputStream, both of which are InputStream classes.
- Example!

Singleton Pattern

- Used when you want a single instance of class
  - one and only one object should exist within the JVM
- Easy to do unless you need to be thread-safe
- Example!
  - Non-thread safe
  - Thread safe

Questions / Final Review

- Regarding libraries?
- Regarding testing?
- Regarding design patterns?
- Regarding Core Java?
- Regarding the Final Exam?