Interfaces in Java

Interface Overview

- Think of interfaces as higher level templates than classes
    - In the blueprint analogy.
        - Class = blueprint (template of how a building will look like when created)
        - Object = creation of the blueprint (the actual building)
        - Interface = commonalities between blueprints (house, skyscraper, library, etc)
    - Interfaces are groupings of methods
        - You're creating a type where you do not care about the implementation (Class) but want to express desired functionality (Interface as a grouping of methods).

Interface Example

```
1    package edu.nyu.cs9053.interfaces;
2
3    /**
4     * User: blangel
5     * Date: 9/21/14
6     * Time: 8:36 AM
7     */
8    public interface StringSplitter {
9
10        String[] split(String input, String by);
11
12   }
```

Interface Usage Example

```java
1   package edu.nyu.cs9053.interfaces;
2
3   /**
4    * User: blangel
5    * Date: 9/21/14
6    * Time: 8:39 AM
7    */
8   public class Book {
9
10      private final String[] words;
11
12      public Book(String text, StringSplitter stringSplitter) {
13          this.words = stringSplitter.split(text, " ");
14      }
15
16      public int getWordCount() {
17          return this.words.length;
18      }
19
20      public String[] getWords() {
21          return words;
22      }
23  }
```

Interface Implementation

```java
package edu.nyu.cs9053.interfaces;

import java.util.regex.Pattern;

/**
 * User: blangel
 * Date: 9/21/14
 * Time: 10:07 AM
 */
public class RegexStringSplitter implements StringSplitter {

    @Override public String[] split(String input, String by) {
        if ((input == null) || (by == null)) {
            return (input == null ? null : new String[] { input });
        }
        return input.split(Pattern.quote(by));
    }
}
```

Interface Implementation (cont)

```java
1    package edu.nyu.cs9053.interfaces;
2
3    /**
4     * User: blangel
5     * Date: 9/21/14
6     * Time: 10:14 AM
7     */
8    public class DistributedStringSplitter extends NetworkAccessor implements StringSplitter {
9
10       @Override public String[] split(String input, String by) {
11           String hashKey = computeHash(by, input);
12           return process(hashKey);
13       }
14   }
```

Interface Particulars

- All methods are public (so no need to put `public` in front of each method)
- Interfaces cannot have default implementations (unlike abstract classes)
- Interfaces cannot have instance variables
  - But can have `static final` variables (if you don't specify this it'll be defaulted)
- Interfaces can `extend` one another
- Interfaces without methods are called marker or tagging interfaces (in general don't do this)
- Classes can `implement` multiple interfaces

Multiple (interface) Inheritance

- A class can implement 0 to many interfaces. This is a very powerful way to get most of the benefits of multiple inheritance without the "double diamond" problem.
    - To share code, prefer composition (more about this later)

Multiple (interface) Inheritance Example

```java
1   package edu.nyu.cs9053.interfaces;
2
3   /**
4    * User: blangel
5    * Date: 9/21/14
6    * Time: 10:14 AM
7    */
8   public class DistributedStringSplitter extends NetworkAccessor implements Distributed, StringSplitter {
9
10      private final String serverUrl;
11
12      public DistributedStringSplitter(String serverUrl) {
13          this.serverUrl = serverUrl;
14      }
15
16      @Override public String[] split(String input, String by) {
17          String hashKey = computeHash(by, input);
18          return process(hashKey);
19      }
20
21      @Override public String getServer() {
22          return serverUrl;
23      }
24  }
```

Super Interfaces

- Interfaces can extend other interfaces

```
1   package edu.nyu.cs9053.interfaces;
2
3   /**
4    * User: blangel
5    * Date: 9/21/14
6    * Time: 10:36 AM
7    */
8   public interface Reader {
9
10      void read(String text);
11
12  }
```

```
1   package edu.nyu.cs9053.interfaces;
2
3   /**
4    * User: blangel
5    * Date: 9/21/14
6    * Time: 10:38 AM
7    */
8   public interface BookReader extends Reader {
9
10      void read(Book book);
11
12  }
```

Marker Interfaces

- Interfaces without methods
  - Form of meta-programming; prefer @Annotations (next Lecture)

```
1    ...
2    * @see java.io.Externalizable
3    * @since    JDK1.1
4    */
5    public interface Serializable {
6    }
```

```
1    ...
2    public class Book implements Serializable {
3    ...
```

Interface v. Abstract Class

- More often than not prefer Interfaces.
- A good distinction is that interfaces are cross-cutting concerns irrespective of concrete type (Splitter, Searcher, Cloner, etc) and abstract-classes are something nearly concrete but which is shared by other concrete classes (e.g., AbstractEmployee is abstract class of Manager and Programmer)
- Typing to an interface allows for more flexibility in the future and makes tasks like testing much easier.
  - A good pattern is to first make an Interface then make an AbstractClass which implements that Interface and provide the common implementations then provide the ConcreteClasses.

Interface v. Abstract Class (cont)

- Consider this (old Employee structure)

```java
public abstract class Employee {

    private final String name;

    private final double salary;

    Employee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public double getSalary() {
        return salary;
    }
}
```

Interface v. Abstract Class (cont)

```
1    package edu.nyu.cs9053.interfaces;
2
3    /**
4     * User: blangel
5     * Date: 9/21/14
6     * Time: 1:02 PM
7     */
8    public interface Employee {
9
10       String getName();
11
12       Double getSalary();
13
14   }
```

Interface v. Abstract Class (cont)

```java
1   package edu.nyu.cs9053.interfaces;
2
3   /**
4    * User: blangel
5    * Date: 9/21/14
6    * Time: 1:17 PM
7    */
8   abstract class AbstractEmployee implements Employee {
9
10      private final String name;
11
12      private final Double salary;
13
14      AbstractEmployee(String name, Double salary) {
15          this.name = name;
16          this.salary = salary;
17      }
18
19      @Override public String getName() {
20          return name;
21      }
22
23      @Override public Double getSalary() {
24          return salary;
25      }
26
27  }
```

Interface v. Abstract Class (cont)

```java
1   package edu.nyu.cs9053.interfaces;
2
3   /**
4    * User: blangel
5    * Date: 9/21/14
6    * Time: 1:35 PM
7    */
8   public class SalesPerson extends AbstractSalesPerson implements Employee {
9
10      private final String name;
11
12      private final Double base;
13
14      private final Double commission;
15
16      public SalesPerson(String name, Double base, Double commission) {
17          this.name = name;
18          this.base = base;
19          this.commission = commission;
20      }
21
22      @Override public String getName() {
23          return name;
24      }
25
26      @Override public Double getSalary() {
27          return (base + commission);
28      }
29  }
```

Callback Pattern

- Common for UI programming, asynchronous programming and as a way to pass function "pointers" around.

```java
package edu.nyu.cs9053.interfaces;

/**
 * User: blangel
 * Date: 9/21/14
 * Time: 3:48 PM
 */
public interface Callback {

    void loaded(String[] results);

}
```

```java
package edu.nyu.cs9053.interfaces;

/**
 * User: blangel
 * Date: 9/21/14
 * Time: 3:59 PM
 */
public class CallbackImplementation implements Callback {

    @Override public void loaded(String[] results) {
        for (String result : results) {
            System.out.printf("%s%n", result);
        }
    }

}
```

Callback Pattern (cont)

```java
package edu.nyu.cs9053.interfaces;

/**
 * User: blangel
 * Date: 9/21/14
 * Time: 3:58 PM
 */
public class CallbackInvoker {

    public void callAsynchronousTask(AsynchronousMethod async) {
        CallbackImplementation callback = new CallbackImplementation();
        async.invoke(callback);
        // will return here immediately, once the async task has
        // completed the callback.loaded value will be called
    }

}
```

Attack of the Clones!

- Do not use `Object.clone()`
  - Older construct of Java which has fallen out of practice.
  - Better ways to do this - immutable objects with constructors
  - Hard to maintain - if you add a field you have to ensure you update your clone method as well.

Inner Classes

```java
package edu.nyu.cs9053.inner;

/**
 * User: blangel
 * Date: 9/21/14
 * Time: 4:21 PM
 */
public class Outer {

    public class Inner {

    }

}
```

Inner Classes (cont)

- Useful organizational construct.
  - Can mark the inner class private to hide from outside classes

```java
public class Sorter {

    private class Alphabetic implements Iterator<String> {

        private int current;

        private Alphabetic() {
            Arrays.sort(values);
            this.current = 0;
        }

        @Override public boolean hasNext() {
            return (current != values.length);
        }
        @Override public String next() {
            return values[current++];
        }
        @Override public void remove() {
            throw new UnsupportedOperationException();
        }
    }

    private final String[] values;

    public Sorter(String[] values) {
        this.values = values;
    }

    public Iterator<String> alphabeticIterator() {
        return new Alphabetic();
    }

}
```

Inner Classes (cont)

- The inner class has an implicit reference to the outer class.
  - Careful as this can leak memory!

```java
1   public class Sorter {
2
3       private class Alphabetic implements Iterator<String> {
4
5           private final String[] values;
6
7           private int current;
8
9           private Alphabetic() {
10              this.values = Arrays.copyOf(Sorter.this.values, Sorter.this.values.length);
11              Arrays.sort(values);
12              this.current = 0;
13          }
14
15          @Override public boolean hasNext() {
16              return (current != values.length);
17          }
18          @Override public String next() {
19              return values[current++];
20          }
21          @Override public void remove() {
22              throw new UnsupportedOperationException();
23          }
24      }
25
26      private final String[] values;
27
28      public Sorter(String[] values) {
29          this.values = values;
30      }
31
32      public Iterator<String> alphabeticIterator() {
33          return new Alphabetic();
34      }
35
36  }
```

Inner Classes - Implicit Reference

```java
public class Outer {

    public class Inner {

        private final Outer implicitReference;

        public Inner(Outer implicitReference) {
            this.implicitReference = implicitReference;
        }
    }

    public Inner createInner() {
        return new Inner(this);
    }

}
```

Inner Class - Gotcha!

```java
public class Processor {

    public class Result {

        private final String result;

        public Result(String result) {
            this.result = result;
        }

        public String getResult() {
            return result;
        }
    }

    private final String[] hasher;

    public Processor() {
        this.hasher = new String[1024];
        for (int i = 0; i < hasher.length; i++) {
            this.hasher[i] = UUID.randomUUID().toString();
        }
    }

    public Result process(String find) {
        for (String hash : hasher) {
            if (hash.equals(find)) {
                return new Result(hash);
            }
        }
        return new Result("");
    }

}
```

Inner Class - Gotcha! (cont)

```java
public class MemoryProcessor {

    public static void main(String[] args) {
        List<Processor.Result> results = new ArrayList<>(1000);
        int i = 0;
        do {
            printFreeMemory();
            Processor.Result result = getResult();
            results.add(result);
            printFreeMemory();
            System.out.printf("Result %s%n", result.getResult());
        } while (i++ < 35);
    }

    private static Processor.Result getResult() {
        Processor processor = new Processor();
        printFreeMemory();
        return processor.process("something");
    }

    private static void printFreeMemory() {
        long memory = Runtime.getRuntime().freeMemory();
        System.out.printf("%d MB memory%n", Math.round(memory / 1024d / 1024d));
        System.gc();
    }

}
```

Inner (static) Class!

```java
public class Processor {

    public static class Result {

        private final String result;

        public Result(String result) {
            this.result = result;
        }

        public String getResult() {
            return result;
        }
    }
...
```

Local Classes - Avoid!

- Classes contained within a method
  - Extremely rare to use (bad practice)
  - One advantage, they are completely hidden from outside world

```java
1   public class ContainingLocalClass {
2
3       public void someMethod() {
4           class Local {
5               void print(String foo) {
6                   System.out.printf("%s%n", foo);
7               }
8           }
9           Local local = new Local();
10          local.print("foo");
11      }
12
13  }
```

Anonymous Classes

- Classes inlined without a name.
  - Often see this with the Callback pattern

```
1   public class AnonymousCallbackExample {
2
3       public void callAsynchronousTask(AsynchronousMethod async) {
4           async.invoke(new Callback() {
5               @Override public void loaded(String[] results) {
6                   for (String result : results) {
7                       System.out.printf("%s%n", result);
8                   }
9               }
10          });
11      }
12
13  }
```

Variable Scope - Anonymous Class

- Can anonymous classes access variables outside of themselves?
  - If so, do they have an implicit reference to the outer class?

```java
public class Scope {

    public void invoke(AsynchronousMethod method) {
        String id = UUID.randomUUID().toString();
        method.invoke(new Callback() {
            @Override public void loaded(String[] results) {
                System.out.printf("Invocation id = %s%n", id);
                for (String result : results) {
                    System.out.printf("%s%n", result);
                }
            }
        });
    }

}
```

Read Chapter 11

All sections

- I'll provide supplemental information in lecture

Homework 5

https://github.com/NYU-CS9053/Fall-2016/homework/week5