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    */
6   public interface StringSplitter {
7
8       String[] split(String input, String by);
9
10  }
```

Interface Usage Example

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

Interface Implementation

```
1  package edu.nyu.cs9053.interfaces;
2
3  import java.util.regex.Pattern;
4
5  /**
6   * User: blangel
7   */
8  public class RegexStringSplitter implements StringSplitter {
9
10     @Override public String[] split(String input, String by) {
11         if ((input == null) || (by == null)) {
12             return (input == null ? null : new String[] { input });
13         }
14         return input.split(Pattern.quote(by));
15     }
16 }
```

Interface Implementation (cont)

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

Interface Particulars

- All methods are public (so no need to put `public` in front of each method)
- 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
package edu.nyu.cs9053.interfaces;

/**
 * User: blangel
 */
public class DistributedStringSplitter extends NetworkAccessor implements Distributed, StringSplitter {

    private final String serverUrl;

    public DistributedStringSplitter(String serverUrl) {
        this.serverUrl = serverUrl;
    }

    @Override public String[] split(String input, String by) {
        String hashKey = computeHash(by, input);
        return process(hashKey);
    }

    @Override public String getServer() {
        return serverUrl;
    }
}
```

Super Interfaces

- Interfaces can extend other interfaces

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

/**
 * User: blangel
 */
public interface Reader {

    void read(String text);

}
```

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

/**
 * User: blangel
 */
public interface BookReader extends Reader {

    void read(Book book);

}
```

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)

```
1  public abstract class Employee {
2
3      private final String name;
4
5      private final double salary;
6
7      Employee(String name, double salary) {
8          this.name = name;
9          this.salary = salary;
10     }
11
12     public String getName() {
13         return name;
14     }
15
16     public double getSalary() {
17         return salary;
18     }
19 }
```

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
package edu.nyu.cs9053.interfaces;

/**
 * User: blangel
 */
abstract class AbstractEmployee implements Employee {

    private final String name;

    private final Double salary;

    AbstractEmployee(String name, Double salary) {
        this.name = name;
        this.salary = salary;
    }

    @Override public String getName() {
        return name;
    }

    @Override public Double getSalary() {
        return salary;
    }

}
```

Interface v. Abstract Class (cont)

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

/**
 * User: blangel
 */
public class SalesPerson extends AbstractSalesPerson implements Employee {

    private final String name;

    private final Double base;

    private final Double commission;

    public SalesPerson(String name, Double base, Double commission) {
        this.name = name;
        this.base = base;
        this.commission = commission;
    }

    @Override public String getName() {
        return name;
    }

    @Override public Double getSalary() {
        return (base + commission);
    }
}
```

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
 */
public interface Callback {

    void loaded(String[] results);

}
```

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

/**
 * User: blangel
 */
public class CallbackImplementation implements Callback {

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

Callback Pattern (cont)

```
1  package edu.nyu.cs9053.interfaces;
2
3  /**
4   * User: blangel
5   */
6  public class CallbackInvoker {
7
8      public void callAsynchronousTask(AsynchronousMethod async) {
9          CallbackImplementation callback = new CallbackImplementation();
10         async.invoke(callback);
11         // will return here immediately, once the async task has
12         // completed the callback.loaded value will be called
13     }
14
15 }
```

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

```
1   package edu.nyu.cs9053.inner;
2
3   /**
4    * User: blangel
5    */
6   public class Outer {
7
8       public class Inner {
9
10      }
11
12  }
```

Inner Classes (cont)

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

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

Inner Classes (cont)

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

```
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

```
1   public class Outer {
2
3       public class Inner {
4
5           private final Outer implicitReference;
6
7           public Inner(Outer implicitReference) {
8               this.implicitReference = implicitReference;
9           }
10      }
11
12      public Inner createInner() {
13          return new Inner(this);
14      }
15
16  }
```

Inner Class - Gotcha!

```
1  public class Processor {
2
3      public class Result {
4
5          private final String result;
6
7          public Result(String result) {
8              this.result = result;
9          }
10
11         public String getResult() {
12             return result;
13         }
14     }
15
16     private final String[] hasher;
17
18     public Processor() {
19         this.hasher = new String[1024];
20         for (int i = 0; i < hasher.length; i++) {
21             this.hasher[i] = UUID.randomUUID().toString();
22         }
23     }
24
25     public Result process(String find) {
26         for (String hash : hasher) {
27             if (hash.equals(find)) {
28                 return new Result(hash);
29             }
30         }
31         return new Result("");
32     }
33
34 }
```

Inner Class - Gotcha! (cont)

```
1  public class MemoryProcessor {
2
3      public static void main(String[] args) {
4          List<Processor.Result> results = new ArrayList<>(1000);
5          int i = 0;
6          do {
7              printFreeMemory();
8              Processor.Result result = getResult();
9              results.add(result);
10             printFreeMemory();
11             System.out.printf("Result %s%n", result.getResult());
12         } while (i++ < 35);
13     }
14
15     private static Processor.Result getResult() {
16         Processor processor = new Processor();
17         printFreeMemory();
18         return processor.process("something");
19     }
20
21     private static void printFreeMemory() {
22         long memory = Runtime.getRuntime().freeMemory();
23         System.out.printf("%d MB memory%n", Math.round(memory / 1024d / 1024d));
24         System.gc();
25     }
26
27 }
```

Inner (`static`) Class!

```
1  public class Processor {
2
3      public static class Result {
4
5          private final String result;
6
7          public Result(String result) {
8              this.result = result;
9          }
10
11         public String getResult() {
12             return result;
13         }
14     }
15 ...
```

Local Classes - Avoid!

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

```
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?

```
1  public class Scope {
2
3      public void invoke(AsynchronousMethod method) {
4          String id = UUID.randomUUID().toString();
5          method.invoke(new Callback() {
6              @Override public void loaded(String[] results) {
7                  System.out.printf("Invocation id = %s%n", id);
8                  for (String result : results) {
9                      System.out.printf("%s%n", result);
10                 }
11             }
12         });
13     }
14
15 }
```

Read Chapter 7

All sections

- I'll provide supplemental information in lecture

Homework 5

https://github.com/NYU-CS9053/Spring-2019/homework/week5