Principles of Software Construction: Objects, Design and Concurrency

Mutability and Java Potpourri

Jonathan Aldrich

Charlie Garrod

© 2012-13 C Garrod, J Aldrich, and W Scherlis
Administrivia

- Charlie's office hours moved to Sunday this week only:
  - Sunday 12:00 to 2:00 p.m., Wean 5101
Key concepts from Tuesday
Key concepts from Tuesday

- `java.lang.Object` behavioral contracts
  - Challenges of inheritance

- Java Exceptions
Key concepts for today

• Mutability

• Java potpourri
  ▪ The static keyword
  ▪ Inner classes
  ▪ Scope and variable shadowing
  ▪ Java Generics
One possible String implementation

```java
public class String {
    private char value[];
    ...

    public void concat(String s) {
        char newValue[] = new char[value.length + s.value.length];
        for (int i = 0; i < value.length; ++i) {
            newValue[i] = value[i];
        }
        for (int i = 0; i < s.value.length; ++i) {
            newValue[value.length+i] = s.value[i];
        }
        value = newValue;
    }

    public void replace(char old, char new) {
        for (int i = 0; i < value.length; ++i) {
            if (value[i] == old) {
                value[i] = new;
            }
        }
    }
}
```
Another possible String implementation

```java
public class String {
    private final char value[];

    public String concat(String s) {
        char newValue[] = new char[value.length + s.value.length];
        for (int i = 0; i < value.length; ++i) {
            newValue[i] = value[i];
        }
        for (int i = 0; i < s.value.length; ++i) {
            newValue[value.length+i] = s.value[i];
        }
        return new String(newValue);
    }

    public String replace(char old, char new) {
        char newValue[] = new char[value.length];
        for (int i = 0; i < value.length; ++i) {
            newValue[i] = value[i];
            if (value[i] == old) {
                newValue[i] = new;
            }
        }
        return new String(newValue);
    }
}
```
• Data is *mutable* if it can change over time. Otherwise it is *immutable*.
  • Variables declared as final are enforced to be immutable
  • ...but data they reference can still be mutable

```java
final List<Integer> vals = new ArrayList<Integer>();
vals.add(42);  // OK even though it changes the list

// Not OK because it changes vals:
vals = new ArrayList<Integer>();
```
Advantages: Mutability vs. Immutability
Immutable advice

- Make your classes and fields immutable if possible
  - Make all data private and final
  - Guarantee exclusive access to your data
    - Defensively copy your data
    - Don't leak your data
  - Don't provide public mutator methods
  - Make your classes final
Key concepts for today

- Mutability
- Java potpourri
  - The static keyword
  - Inner classes
  - Scope and variable shadowing
  - Java Generics
Static data and methods

• A property is *static* if it is associated with the class (as opposed to being associated with an instance of the class)
  ▪ A.k.a. *class* fields and methods

• Examples
  ▪ A simple Counter example
  ▪ The main method – why is this static?
  ▪ The java.lang.String `valueOf` methods
Classes can be defined inside other classes, or even inside class methods

- e.g., A LinkedList.Node class accessible only from within a LinkedList class:

```java
public class LinkedList {
    private static class Node {
        public int val;
        public Node next;
        public Node(int v, Node n) {
            val = v;
            next = n;
        }
    }
    Node head;
    // ...
}
```
Variable shadowing in Java

• Variable shadowing: when a name within some program inner scope matches a name in some outer scope
  ▪ e.g.,
  
```java
public class Dog {
    String name;
    public Dog (String name) {
        this.name = name;
    }
}
```

• Java has class variables, instance variables, and local variables
  ▪ Local variables may shadow instance or class variables
  ▪ A subclass’s variables may shadow superclass variables
Inner scopes in Java

- Curly braces define a new scope
  - e.g., `public void printAsInt(String s) {
      try {
          Integer x = Integer.valueOf(s);
      } catch (NumberFormatException e) {
          // we ignore the exception
      }
      System.out.println(x); // x is undefined
  } // in this scope
Recall the Java Collection API (excerpt)
Consider the `java.util.Stack`

```java
public class Stack {
    public void push(Object obj) { ... }
    public Object pop() { ... }
}

• Some possible client code?:
    Stack stack = new Stack();
    String s = "Hello!";
    stack.push(s);
    String t = stack.pop();
```
Consider the `java.util.Stack`

```java
public class Stack {
    public void push(Object obj) { … }
    public Object pop() { … }
}
```

- **Some possible client code:**
  ```java
  Stack stack = new Stack();
  String s = "Hello!";
  stack.push(s);
  String t = (String) stack.pop();
  ```

  **To fix the type error**
• The `java.util.Stack` instead
  - A stack of some type `T`:
    ```java
    public class Stack<T> {
        public void push(T obj) { ... }
        public T pop() { ... }
    }
    ```

• Improves typechecking, simplifies (?) client code:
  ```java
  Stack<String> stack = new Stack<String>();
  String s = "Hello!";
  stack.push(s);
  String t = stack.pop();
  ```
Next week: design and testing