CS106X
Autumn 2012

Handout 41S
December 5th – 7th, 2012

Section Solution

Discussion Problem 1 Solution: Muppet Inheritance

The **`Kermit *`** can address either a **`Waldorf`** object or a **`Gonzo`** object. Each provides implementations of all of the methods—abstract or not—at the **`Kermit`** level. The **`Kermit`** class clearly has two abstract methods, and the **`Statler`** class doesn't provide an implementation for the **`animal`** method, so it's also an abstract class.

Output for **`Waldorf *`**

```
Kermit::fozzie
Kermit::rowlf
Statler::misspiggy
Statler::rowlf
Waldorf::animal
Waldorf::rowlf
```

Output for **`Gonzo *`**

```
Kermit::fozzie
Kermit::rowlf
Gonzo::misspiggy
Kermit::beaker
Gonzo::animal
Gonzo::rowlf
```

Lab Problem 1 Solution: JavaScript Object Notation Take II [code by Aubrey Gress]

```
class JSONElement {
public:
   virtual ~JSONElement() {};
   virtual string toString() = 0;
private:
};

class JSONString : public JSONElement {
public:
   JSONString(const string& str) { value = str; }
   ~JSONString() {}
   string toString() { return value; };
private:
   string value;
};

class JSONInt : public JSONElement {
public:
   JSONInt(int i) { value = i; }
   ~JSONInt () {}
   string toString() { return integerToString(value); };
private:
   int value;
};

class JSONBoolean : public JSONElement {
public:
   JSONBoolean(bool b) {value = b;}
   ~JSONBoolean () {}
   string toString() { return value ? "true" : "false"; };
private:
   bool value;
};
```

```
class JSONArray: public JSONElement {
public:
   JSONArray(const Vector<JSONElement*>& arr) { value = arr; }
   ~JSONArray () {
      for (int i = 0; i < value.size(); i++)
         delete value[i];
   }
   string toString() {
      string str = "[";
      for (int i = 0; i < value.size(); i++) {
         str += value[i]->toString();
         if (i != value.size() - 1) {
            str += ", ";
         }
      }
      str += "]";
      return str;
   };
private:
   Vector<JSONElement *> value;
};

class JSONDictionary: public JSONElement {
public:
   JSONDictionary(const Map<string, JSONElement *>& dictionary) {
      value = dictionary;
   }
   ~JSONDictionary () {
      foreach(string key in value)
        delete value[key];
   }

   string toString() {
      string str = "{";
      int counter = 0;
      foreach (const string& key in value) {
         string keyToPrint = key;
         str += keyToPrint;
         str += " : ";
         str += value[key]->toString();
         if (counter != value.size() - 1) {
            str += ", ";
         }
         counter++;
      }
      str += "}";
      return str;
   };

private:
   Map<string, JSONElement *> value;
};
```

```
JSONElement *parseJSON(TokenScanner& scanner);

Vector<JSONElement *> parseJSONArray(TokenScanner& scanner) {
   Vector<JSONElement *> array;
   bool firstElementConsumed = false;
   while (true) {
      string lookahead = scanner.nextToken();
      if (lookahead == "]") return array;
      if (firstElementConsumed && lookahead != ",") {
         error("Oops!  Commas need to separate elements in a JSON array.");
      } else if (!firstElementConsumed) {
         scanner.saveToken(lookahead);
      }

      JSONElement *element = parseJSON(scanner);
      firstElementConsumed = true;
      array.add(element);
   }
}

Map<string, JSONElement *> parseJSONDictionary(TokenScanner& scanner) {
   Map<string, JSONElement *> dictionary;
   bool firstEntryConsumed = false;
   while (true) {
      string lookahead = scanner.nextToken();
      if (lookahead == "}") return dictionary;
      if (firstEntryConsumed && lookahead != ",") {
         error("Oops!  Commas need to separate entries in a JSON dictionary.");
      } else if (!firstEntryConsumed) {
         scanner.saveToken(lookahead);
      }

      string key = scanner.nextToken();
      if (scanner.nextToken() != ":") {
         error("Expected a colon to separate the key and value pair.");
      }

      JSONElement *value = parseJSON(scanner);
      firstEntryConsumed = true;
      dictionary[key] = value;
   }
}

JSONElement *parseJSON(TokenScanner& scanner) {
   string lookahead = scanner.nextToken();
   if (lookahead.empty()) return NULL;
   if (isdigit(lookahead[0])) {
      return new JSONInt(stringToInteger(lookahead));
   } else if (lookahead == "true" || lookahead == "false" ) {
      return new JSONBoolean(lookahead == "true");
   } else if (lookahead[0] == '"') {
      return new JSONString(lookahead);
   } else if (lookahead[0] == '[') {
      return new JSONArray(parseJSONArray(scanner));
   } else if (lookahead[0] == '{') {
      return new JSONDictionary(parseJSONDictionary(scanner));
   } else {
      error("JSON element type passed to parseJSON not yet supported.");
   }
   return NULL;
}
```