

Section 8 (Week 9) – SOLUTION

Problem and solution authors include Jerry Cain and Aubrey Gress.

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

Output for `Gonzo` *

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

Problem 2 Solution: Using Polymorphism with JavaScript Object Notation

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

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

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

class JSONBoolean : public JMLElement {
public:
    JSONBoolean(bool b) {value = b;}
    ~JSONBoolean () {}
};
```

```

    string toString() { return value ? "true" : "false"; };
private:
    bool value;
};

class JSONArray: public JMLElement {
public:
    JSONArray(const Vector<JMLElement*>& 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<JMLElement *> value;
};

class JSONDictionary: public JMLElement {
public:
    JSONDictionary(const Map<string, JMLElement *>& 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;  
}
```