

INFSCI 2950: Independent Study (Systems)

Prof. Peter Brusilovsky

Summer Term 2016-2017

Final Project Report:

Parsons Programming Puzzles for Java Language



Sai Supreetha Varanasi

MSIS - SAIS

sav52@pitt.edu

School of Computing and Information

University of Pittsburgh

Table of Contents

Introduction:	2
Software:	4
Project Structure:	4
Database:	6
Application:	8
Scripts:	12
Hackerrank API:	14
Conclusion and Future Scope:	16
References:	16

Introduction:

This project is an extension of the prior work on Interactive Programming Puzzles design and development that can be found here: <http://tinyurl.com/InteractiveJ>. The prior work dealt with the various aspects like the Requirement gathering, Mockup preparation, Design and Task Analysis of the mockup and iterative improvement, and, Prototype development.

The earlier prototype had three types of programming puzzles called “Example” which is a quiz like rendering of the program for the user to select the correct code fragments out of 4 options to satisfy the given goal, “Faded Example” which is similar to the former except there are no hints while the former had fragment-level hints and the “Problem” which is Parsons programming puzzles implementation for Java language.

The evaluation of the earlier prototype resulted in the need for an extensible application, specially for the ‘Problem’ or the Parsons puzzles. This project aims to satisfy that goal of making the process of creation of Puzzles as well as handling of the compilation and running process entirely at the application level.

The extensible-ness is made possible in this project by the fact that the entire Parsons puzzle creation is dependent on the it corresponding fully functional .java file with some additional meta data as a part of it. This process is majorly automated and requires running a couple of scripts in a particular sequence.

The application level compiling and running the code after the puzzle has been solved is handled by the Hackerrank API where a request to the api is posted and the response from it contains the compilation information is used to provide feedback.

Fig[1,2,3]: Earlier prototype of ‘Example’, ‘Faded Example’ and ‘Problem’

Interactive Programming Examples in Java

Example
Faded Example
Problem

Instructions

Goal: Drag a tile to the highlighted field to construct a program that creates an array of length 5 and fills up the array with the first five even integers.

Next
→

Check

Correct answer! :)

Show me what this program constructs ?

This program fills up array with first five even integers: [0,2,4,6,8]

Solution:

```

public class ArrayInitialization{
    public void initArray{
        //Step 1: Creating the array
        int[] list = new int[5];
        //Step 2: Filling the array
        for ( int i = 0; i < list.length; i++ ) {
            list[i] = 2 * i;
        }
    }
}
                    
```

Drag a tile from here:

- list[i] = 3; ?
- list[i] = 2 * i + 1; ?
- list[i] = list[i] - 1; ?
- list[i] = list[i] + 1; ?

Interactive Programming Examples in Java

Example
Faded Example
Problem

Instructions

Goal: Drag a tile to the highlighted field to construct a program that creates an array of length 5 and fills up the array with the first five odd integers.

Hint
?

Check

Solution:

```

public class ArrayInitialization{
    public void initArray{
        //Step 1: Creating the array
        int[] list = new int[5];
        //Step 2: Filling the array
        for ( int i = 0; i < list.length; i++ ) {
        }
    }
}
                    
```

Drag a tile from here:

- list[i] = 3;
- list[i] = 2 * i;
- list[i] = 2 * i + 1;
- list[i] = list[i] - 1;
- list[i] = list[i] + 1;

Interactive Programming Examples in Java

Example
Faded Example
Problem

Instructions

Goal: Drag a tile to the highlighted field to construct a program that creates an array of length 5 and fills up the array with the first five odd integers.

Hint
?

Check

Code fragments in your program are wrong, or in wrong order. This can be fixed by moving, removing, or replacing highlighted fragments.

Click on Hint button if you need help.

Solution:

```

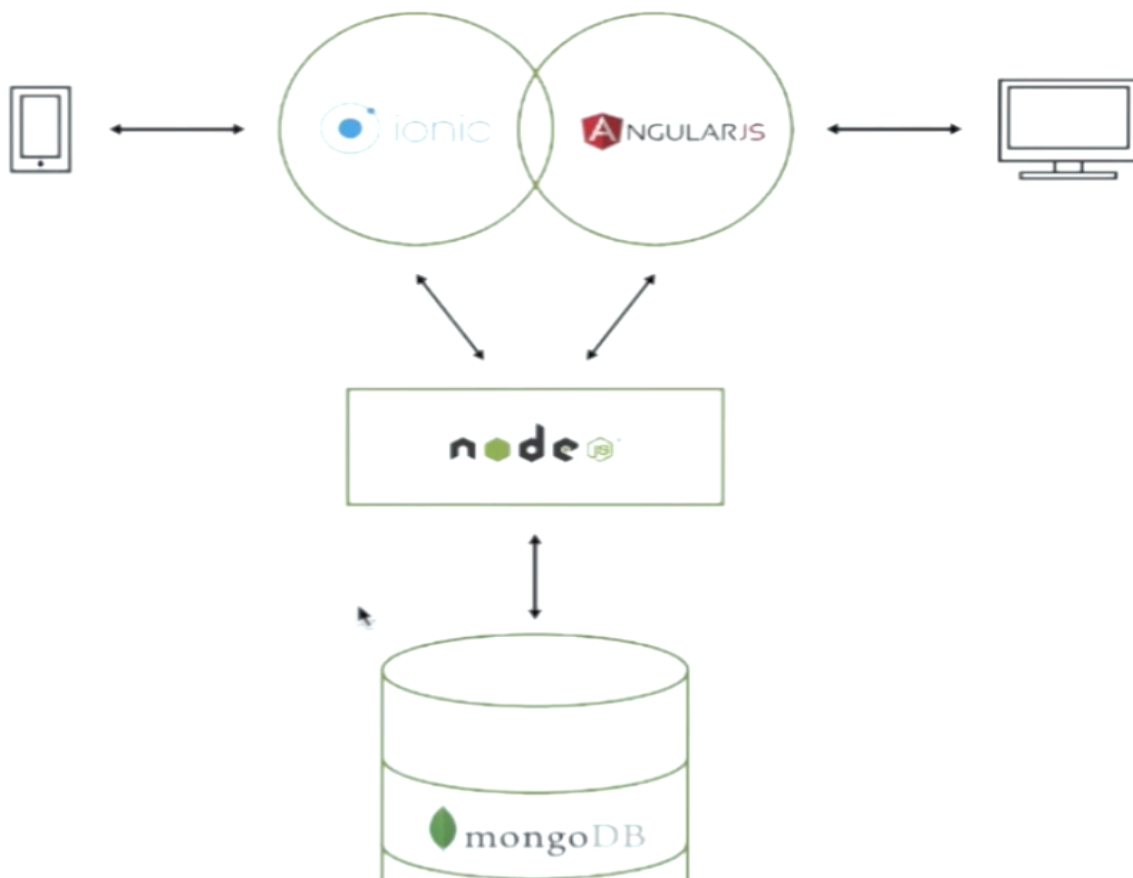
public class ArrayInitialization{
    public void initArray{
        for ( int i = 0; i < list.length; i++ ) {
            int[] list = new int[5];
            list[i] = 2 * i + 1;
            list[i] = 2 * i;
            list[i] = 3;
        }
    }
}
                    
```

Drag a tile from here:

Software:

The software used to develop this project is the full-stack JavaScript framework, the MEAN stack. MEAN stands for MongoDB, ExpressJS, AngularJS and NodeJS respectively. Node is the application server which handles the mongodb database and provisions the business logic. MongoDB is the NoSQL database which is ideal here as this project deals with JSON data. ExpressJS provides the middle layer logic between the front-end and the back-end. AngularJS is the front-end framework that is used to handle the data obtained from the middle layer and plug it into the front-end views. The architecture is as shown in Fig4. The use of these JavaScript framework also due to the fact that it provides fluidity and responsiveness to this application.

Fig4: Full stack architecture of Parsons Programming Puzzles in Java project



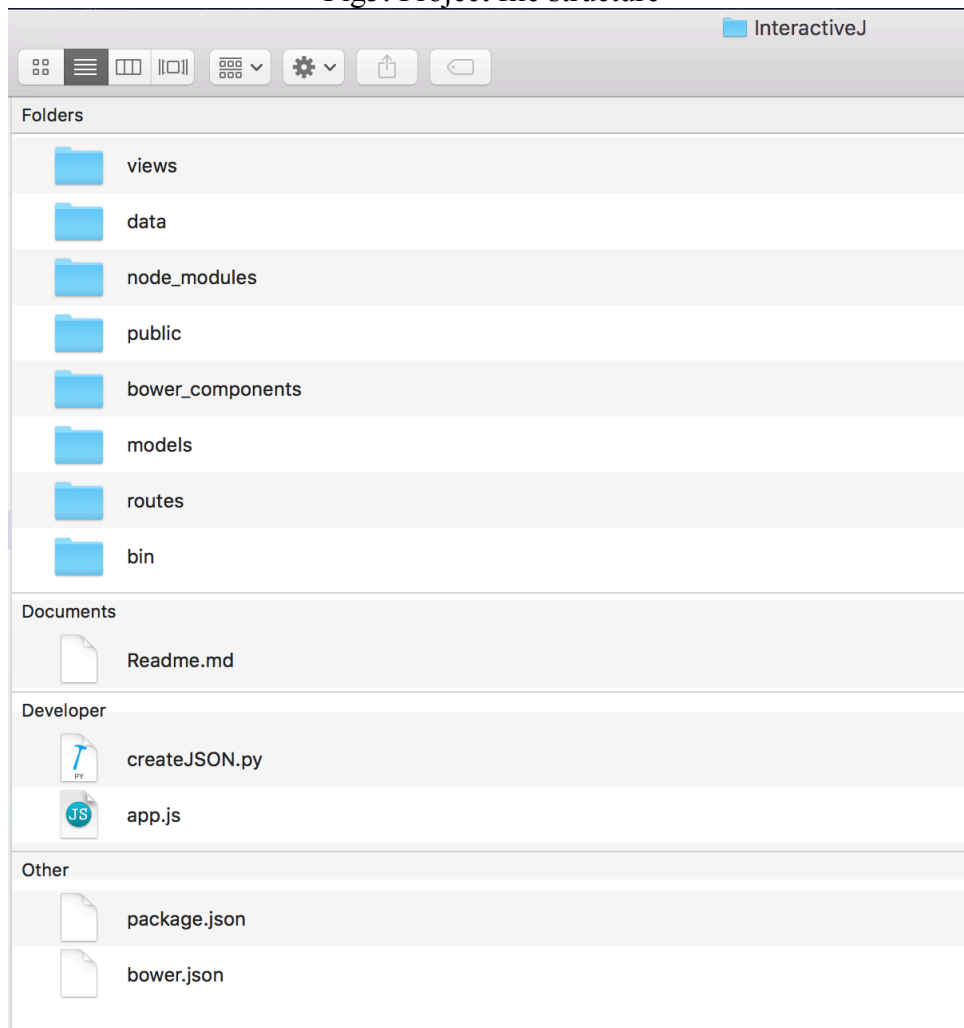
Project Structure:

The project contains various folders for compactness and for general organizing as can be seen from the Fig5. Descriptions and contents of each folder is given below:

- /view/ contain the front-end html files – gateway to the project.
- /data contains the .java files and their respective .json files along with a bash script.

- /node_modules/ folder is created when a new node app is created and its dependencies are installed using package.json file which can also be seen below.
- /public/ folder contains the static files like images, stylesheets and the js scripts.
- /bower_components/ is created when front-end libraries are installed using bower.json file.
- /models/ is the folder where the database schema is present and that is used to make calls to the mongodb database.
- /routes/ folder has the rest api implementation which are rendered as views with the help of angularjs components.
- /bin/www/ contains the compiled version of the project which is used for starting the project on local node server.
- Readme.md is an intro to the file for github repository's sake.
- createJSON.py is a python script that is used to convert the .java files to .json files.
- app.js is the main entry point to this project. It describes the configuration information.

Fig5: Project file structure



Database:

As mentioned above, the database being used is mongodb. The steps to be followed to setup a database for the project and the steps for adding new data to the database are currently same. The foremost thing to do is to get the working .java file in the format as shown in Fig6.

Fig6: Format of the java file with additional metadata

```
1 //goalDesc: Drag the tiles from Problem code fragments field (right) to Solution field (left) to construct a pr
2 //outputDesc: This program should print the looping variable's values in each iteration of the loop.
3 public class Conditionals1 {
4     public static void main(String[] args) {
5         //isSubgoal: Step 1 - Initialization of the looping variable.
6         int i=1; //helpDesc: Can initialize to any number, but here it is 1.
7         do{ //isBlank,isDistractor,helpDesc: Using the do keyword to start looping.
8             System.out.println(i);
9             //isSubgoal: Step 2 - Incrementing the looping variable inside the loop.
10            i++;
11        }while(i<=10); //isBlank,isDistractor,helpDesc: At every iteration check for the condition using while keywo
12    }
13 }
```

Java file and its Metadata:

The file naming follows a convention that:

- Each of the program file is illustrating a Java concept. This is called *activityName* attribute. Examples are: ArrayInitialization, Conditionals, DataStructures, InputOutput, Puzzles etc.,
- Each of the file therefore is named as [activityName][id].java like for example: ArrayInitialization1.java, Conditionals5.java, Puzzles3.java etc.

The additional information in the .java file apart from the code lines are the metadata attributes. These attributes start with a single-line comment (//) sign. Care should be taken so as to **not include any quotes in the entire file as having ‘ ’ or “ ” inside the .java file** is causes a JSON parsing error. Also, there **should not be any unnecessary empty new lines in the .java file**.

The list of attributes are:

- **outputDesc:** The expected output as per the program author. **[required]**
- **goalDesc:** The main goal of the program. The goal is expected to start with “Drag the tiles from Problem field to Solution field **to construct a program** . . .” **[required]**
- **isSubgoal:** The important line can be flagged with this attribute but should be **above** that that important line. These are just like general comments in actual code. **[Optional]**

[Atleast one line of file should be provided as these attributes below]

- **isBlank:** This attribute is designed for ‘Example’ & ‘Faded Example’ puzzles for creation of empty slot in the static blocks. It should be specified **beside the line** just as shown in Fig6. [Atleast one fragment should be flagged]
- **isDistractor:** This attribute is also designed for ‘Example’ & ‘Faded Example’ puzzles for multiple option list creation. It should be specified **beside the line** just as shown in Fig6.
- **helpDesc:** This attribute is also designed for ‘Example’ & ‘Faded Example’ puzzles for hover-for-help implementation. It should be specified **beside the line** just as shown in Fig6.

JSON structure:

Below is the example JSON structure that is based on the metadata given above as well as some extra attributes:

```
{
  "activityName" : Conditionals,
  "goalId" : 1,
  "correctAnsSet" : [1,2,3,4,5,6,7,8,9],
  "goalDesc" : Drag the tiles from Problem code fragments field (right) to
  Solution field (left) to construct a program to execute looping using do-
  while and print the value of looping variable.,
  "outputDesc" : This program should print the looping variable's values in
  each iteration of the loop.,
  "fragments" : [
    /*Auto identification of code lines and creating the fragments list
    item*/
    {
      "fragId" : 1,
      "isSubgoal" : false,
      "isBlank" : false,
      "isDistractor" : false,
      "helpDesc" : ,
      "fragText" : public class Conditionals1 {
        },
        {
      "fragId" : 2,
      "isSubgoal" : false,
      "isBlank" : false,
      "isDistractor" : false,
      "helpDesc" : ,
      "fragText" : public static void main(String[] args) {
        }
      }
    /*And so on .. */
  ]
}
```

The extra attributes are those that are generated automatically at script level and need not be mentioned by the program author in the .java code file.

- **activityName:** Taken from the name of the .java file.
- **goalId:** It is the id or number in the name of the .java file.
- **correctAnsSet:** This auto-generated attribute is for use in the 'Example' and 'Faded Example' puzzles where there is no provision for application level compilation and running.
- **fragments:** The array of the json representation of code lines auto-generated using the python script.
 - **fragId:** This is auto-assigned
 - **fragText:** This is the main line of code
 - The remaining attributes are author given as described in previous section and are auto-assigned using the script.
 -

The script createJSON.py is run on the command line from the root of the project directory as:

```
/InteractiveJ>$ python createJSON.py
```

This automatically parses the /data/ folder, obtains all the .java files and creates a corresponding .json file and puts them inside /data/. Adding a new java file implies regenerating all .json files again. This is done so as to ensure that any changes in other old files are reflected in the database.

Adding data to mongodb database:

Once the authoring of .java files and their subsequent .json file generation is done, they are added to the mongodb database called parsonsdB and the collection called programs.

On terminal window[1]:

```
$ mongod
```

On terminal window[2] go to the database server:

```
$ mongo
```

Whether parsonsdB already present or not enter this command. If it was not present then it is created:

```
> use parsonsdB
```

If the parsonsdB already present with json documents inserted in programs collection then, delete the collection before running the load.sh script:

```
> db.programs.remove({})
```

On terminal window[3] navigate inside the /data/ folder:

```
/InteractiveJ>$ cd data
```

The load.sh file inserts the .json files inside programs collection of parsonsdB database.

```
/InteractiveJ/data>$ sh load.sh
```

On terminal window[2] check if the json documents are inserted:

```
> db.programs.find()
```

Application:

Once the database setting is done, the next step is to start the parsons programming puzzles application.

If it is new run of the application, then there will not be node_modules folder and needs to be generated with all the dependencies mentioned in package.json by doing:

```
/InteractiveJ>$ npm install
```

If the application has node_modules already installed then simply run:

```
/InteractiveJ>$ nodemon
```

The application can be seen running on localhost:3000

The home page as shown in Fig7 is served by the file /views/home.html. All the programs are listed out there in a fluid manner so a user can navigate to any programming puzzle that he/she likes. Currently there are these 14 parsons java programming puzzles in the application.

Clicking on the puzzle on the home page takes the view to the puzzle page as shown in Fig8 which is served by the /views/index.html. This is optimized and responsively changes for mobile or a screen width of any size.

Fig7: The home page of the parsons Java application

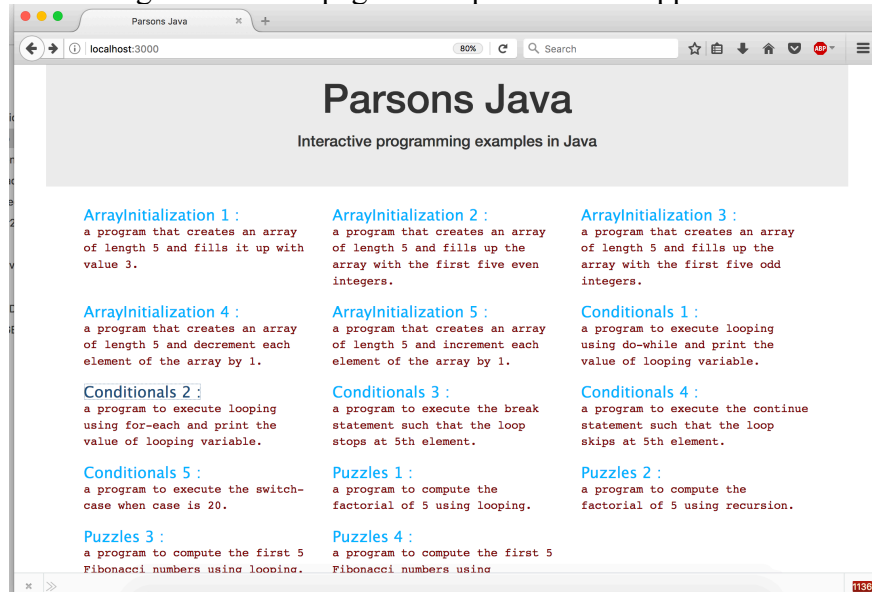
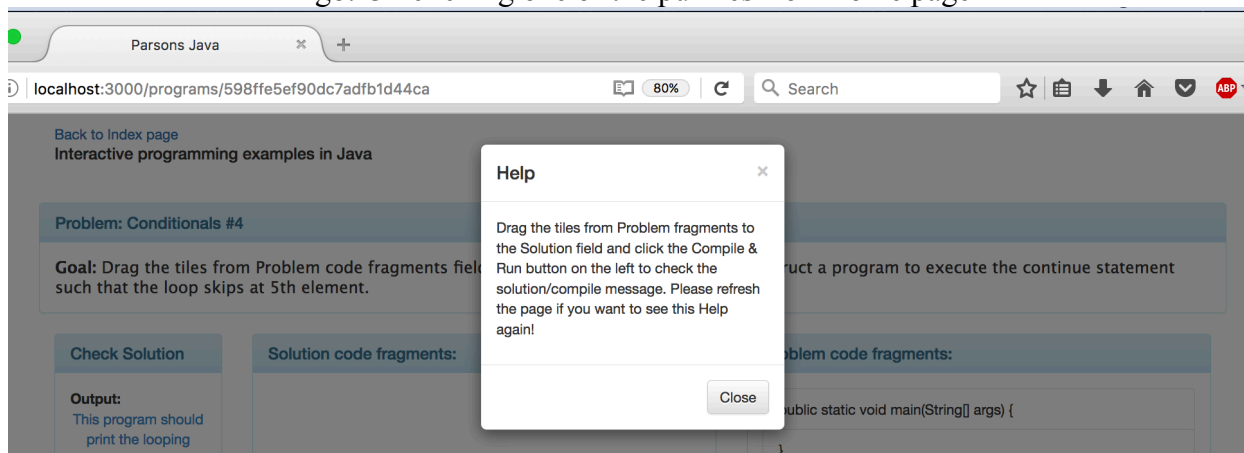


Fig8: On clicking one of the puzzles from home page



It can be seen that an alert-like pop-up is issued on every load of the puzzle to serve as a general-purpose help for solving parsons programming puzzles and for what needs to be done on the interface. This kind of help is chosen because the user is limited to just drag and drop of code fragments in this interface. Levels of help based on the fragments selected and their order is beyond the scope of this project. At present, the application only tries to compile whatever the fragments placed in the Solution field. If they are correct, then the program executes; else, the program returns an error and it is displayed in the Check solution field. The programs may or may not give outputs on the console. That case is handled and the Success message changes accordingly.

Fig9 depicts the index page of the puzzle after the help disappears. This has 4 major elements: Goal, Compile&Run, Solution and Problem. The interface follows the same design as is described in the <http://tinyurl.com/ISD-prototype2>

Fig9: The parsons puzzle interface

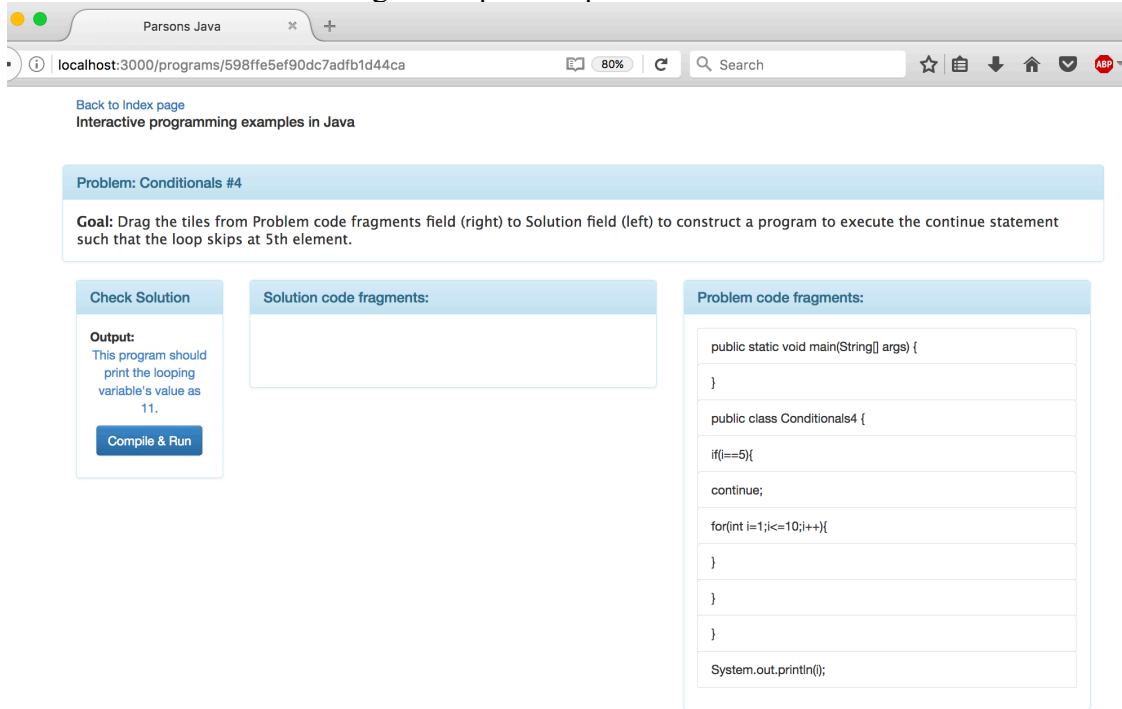


Fig10: Solving the puzzle so the arrangement is wrong

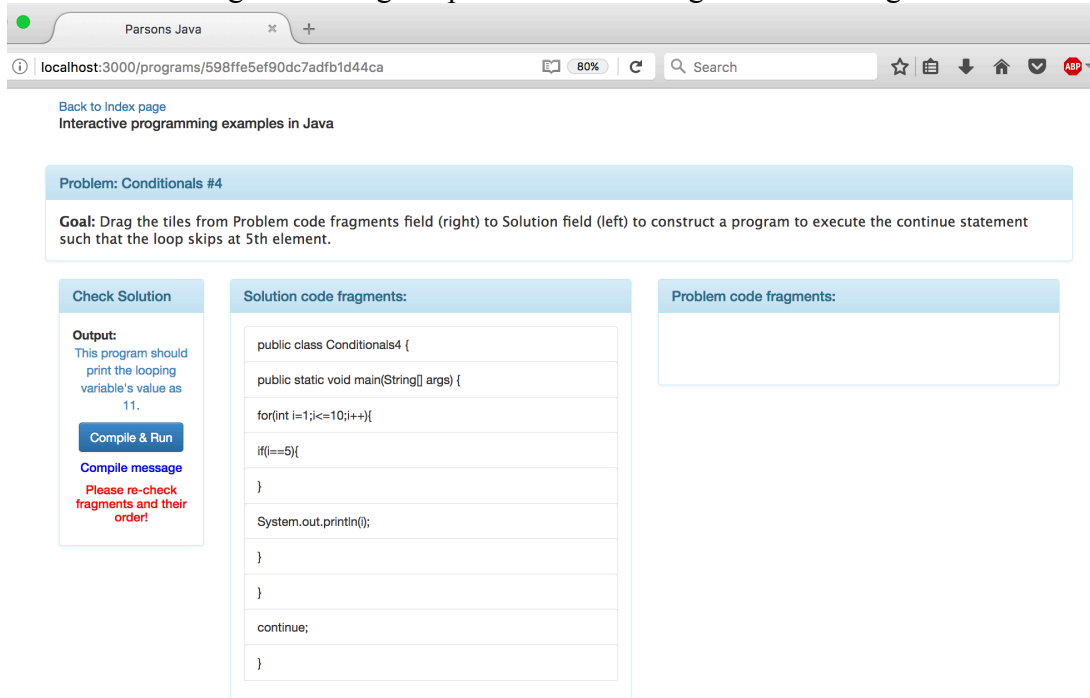


Fig11: Solving the puzzle success

Back to Index page
Interactive programming examples in Java

Problem: Puzzles #1

Goal: Drag the tiles from Problem code fragments field (right) to Solution field (left) to construct a program to compute the factorial of 5 using looping.

Check Solution

Output:
This program should print the value of 5 factorial which is 120.

Compile & Run

Compile message:
Success: ["120\n"]

Solution code fragments:

```
public class Puzzles1{
    public static void main(String args[]){
        int i,fact=1;
        int number=5;
        for(i=1;i<=number;i++){
            fact=fact*i;
        }
        System.out.println(fact);
    }
}
```

Problem code fragments:

Fig12: Console output of Hackerrank Api request and response

```
node · node /usr/local/bin/nodemon ~ -- mongo ~/Desktop/Interactive.J -- mongod ... ~/Desktop/Interactive.J/data -- -bash +
server: 'ip-10-10-164-200',
signal: [ 0 ],
stderr: [ false ],
stdout: [ '1\n2\n6\n24\n120\n' ],
time: [ 0.1 ] } }
POST /compile_run 200 3475.484 ms - 609
=====
Submission:
{ language: '3',
  code: 'public class Puzzles1{public static void main(String args[]){int i,fact=1;int number=5;for(i=1;i<=number;i++){fact=fact*i;}System.out.println(fact);}}',
  testCases: ['"1"'],
  hackerRankApi: 'hackerrank|132697-1690|071ae5d7bb939c23bc27df9e64d1295cc54b5a6e' }
=====
Response:
{ result:
  { callback_url: '',
    censored_compile_message: '',
    codechecker_hash: 'run-gwskSsbZ6EicyhDdSzcM',
    compile_command: '/usr/lib/jvm/java-7-sun/bin/javac -encoding UTF-8 -classpath \':/usr/share/java/*\' Puzzles1.java 1> compile.err 2> &1',
    compilemessage: '',
    error_code: 0,
    hash: '1502609161-1067901096',
    loopback: null,
    memory: [ 36556800 ],
    message: [ 'Success' ],
    response_s3_path: '2017_08_13_07/THgh1a8rsQxInpVE6Ucu0D37oyYbtd2FCz595m4XqWB10JleKA598fff09026530.35752001',
    result: 0,
    run_command: '',
    server: 'ip-10-10-80-36',
    signal: [ 0 ],
    stderr: [ false ],
    stdout: [ '120\n' ],
    time: [ 0.11 ] } } }
POST /compile_run 200 3460.412 ms - 595
```

Fig10 depicts what happens if the fragments are arranged incorrectly. Fig11 depicts what happens if the fragments are arranged correctly. Fig12 depicts the console output of the request string, the response string to and from hackerrank api. This console output is logged for every compilation and run. It provides a more detailed information than is shown on the puzzle interface. This is done so as to not overwhelm the beginner users who are the target users of the platform.

Scripts:

Since this is a full stack application in Javascript, every major functionality is provided by the same. The various scripts used along with their functionality code snippets and description are provided in this section.

[app.js:](#)

This is the starting point of the application and in this file is where the server is initialized, all the dependencies are declared and configurations are set. This handles the routing mechanism as well where it says which script should take care of the view if localhost:3000 is navigated to or if localhost:3000/programs/:id is navigated to. It takes care of calling the hackerrank api with the request structure needed for the api call to take place successfully. It also establishes the connection to the database.

Fig12: app.js configurations

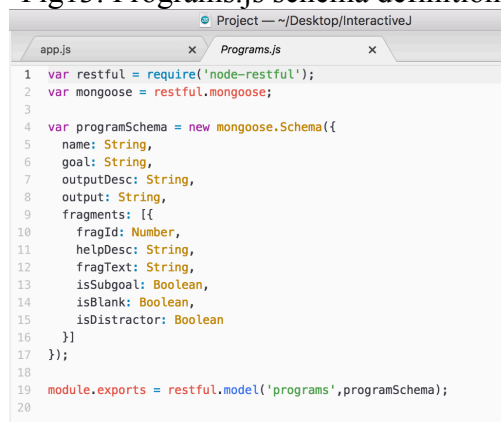
```
30 app.use(logger('dev'));
31 app.use(bodyParser.json());
32 app.use(bodyParser.urlencoded({ extended: false }));
33 app.use(cookieParser());
34 app.use(cors());
35 app.use(express.static(path.join(__dirname, 'public')));
36 app.use(express.static('files'));
37
38
39 app.use('/', index);
40 app.use('/:id', programs);
41
42 var mongoose = require('mongoose'); //for mongodb
43 mongoose.Promise = global.Promise;
44 mongoose.connect('mongodb://localhost/parsonsdb', function(err, db) {
45   if (err) {
46     console.log('Unable to connect to the server. Please start the server. Error:', err);
47   } else {
48     console.log('Connected to Server successfully!');
49   }
50 });
51
52 // error handler
53 app.use(function(err, req, res, next) {
54   // set locals, only providing error in development
55   res.locals.message = err.message;
56   res.locals.error = req.app.get('env') === 'development' ? err : {};
57   // render the error page
58   res.status(err.status || 500);
59   res.render('error');
60 });
61
```

[Programs.js](#)

This is the schema definition script for our database. This takes the help of the mongoose ORM which effortlessly handles the JSON documents procurement. As can be seen from Fig13, the schema is just as given by the JSON structure that was designed and created to suit the application. In a MEAN stack application the data between the views and business logic are also exchanged only in the form of JSON. Anything that gets passed or returned is of that format as well, which makes it difficult if an object relation mapper like mongoose module is not used. The

programSchema called programs being exposed to other Express scripts that handles business logic using the module.exports mechanism.

Fig13: Programs.js schema definition



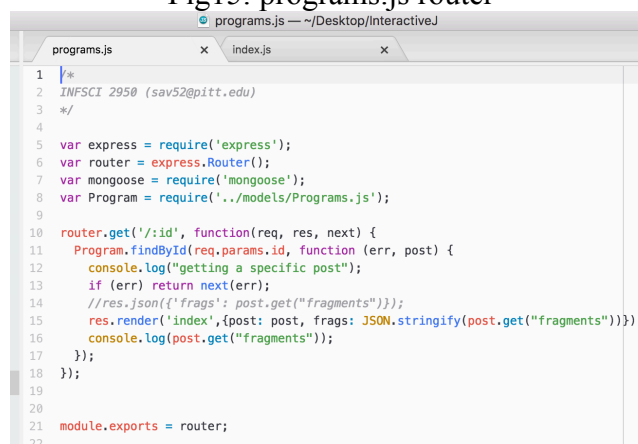
```
1 var restful = require('node-restful');
2 var mongoose = restful.mongoose;
3
4 var programSchema = new mongoose.Schema({
5   name: String,
6   goal: String,
7   outputDesc: String,
8   output: String,
9   fragments: [{
10    fragId: Number,
11    helpDesc: String,
12    fragText: String,
13    isSubgoal: Boolean,
14    isBlank: Boolean,
15    isDistractor: Boolean
16  }]
17 });
18
19 module.exports = restful.model('programs', programSchema);
20
```

Fig14: index.js router



```
1 /*
2 INFSCI 2950 (sav52@pitt.edu)
3 */
4
5 var express = require('express');
6
7 var router = express.Router();
8 var Program = require('../models/Programs.js');
9
10 /* GET home page. */
11 //What happens when we go to just localhost:3000 for example?
12 router.get('/', function(req, res, next) {
13   Program.find({}, function (err, post) {
14     if (err) return next(err);
15     //res.json(post);
16     res.render('home', {programs: post});
17   });
18 });
19
20 module.exports = router;
21
```

Fig15: programs.js router



```
1 /*
2 INFSCI 2950 (sav52@pitt.edu)
3 */
4
5 var express = require('express');
6 var router = express.Router();
7 var mongoose = require('mongoose');
8 var Program = require('../models/Programs.js');
9
10 router.get('/:id', function(req, res, next) {
11   Program.findById(req.params.id, function (err, post) {
12     console.log("getting a specific post");
13     if (err) return next(err);
14     //res.json({'frags': post.get("fragments")});
15     res.render('index', {post: post, frags: JSON.stringify(post.get("fragments"))});
16     console.log(post.get("fragments"));
17   });
18 });
19
20
21 module.exports = router;
22
```

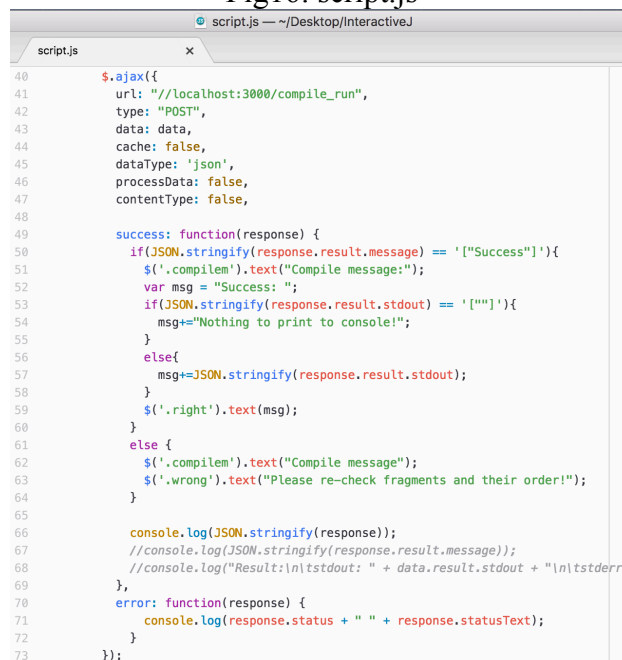
[index.js & programs.js:](#)

The Fig14 and Fig15 gives the router scripts that are used to handle the requests from user when navigated to localhost:3000 or localhost:3000/:id. The index.js route retrieves all the program documents from the database and renders them on the home.html view. The programs.js routes the individual program's specific JSON object and renders it as a parsons puzzle view using index.html. Note that this programs.js which routes the JSON data specific to a puzzle is different from the Programs.js that defines the schema.

[script.js:](#)

The script.js is present inside the /public/javascripts/ folder of the project. The main task of this script is to handle the functionality when the 'Compile & Run' button on the puzzle page is clicked. After user drags and drops the code fragments to solve a parsons puzzle, and when the button is clicked, the code fragments in the solution field are converted to a string. A jQuery ajax call is used to post this string to the node server. Since our node server is provided by app.js, that itself handles this ajax post request by the client.

Fig16: script.js



```
script.js
40 $.ajax({
41   url: "http://localhost:3000/compile_run",
42   type: "POST",
43   data: data,
44   cache: false,
45   dataType: 'json',
46   processData: false,
47   contentType: false,
48
49   success: function(response) {
50     if(JSON.stringify(response.result.message) == '["Success"]'){
51       $('#compile').text("Compile message:");
52       var msg = "Success: ";
53       if(JSON.stringify(response.result.stdout) == '[""]'){
54         msg+="Nothing to print to console!";
55       }
56       else{
57         msg+=JSON.stringify(response.result.stdout);
58       }
59       $('#.right').text(msg);
60     }
61     else {
62       $('#.compile').text("Compile message:");
63       $('#.wrong').text("Please re-check fragments and their order!");
64     }
65
66     console.log(JSON.stringify(response));
67     //console.log(JSON.stringify(response.result.message));
68     //console.log("Result:\n\tstdout: " + data.result.stdout + "\n\tstderr:");
69   },
70   error: function(response) {
71     console.log(response.status + " " + response.statusText);
72   }
73 });
```

Hackerrank API:

Before the ajax call is made, the request data is prepared by appending the Hackerrank api corresponding information. The api url is api.hackerrank.com/checker/submission.json and the code related information needs to be POSTed to this api to successfully compile and run it. As can be seen from Fig18, source code, language, testcases and the apikey are the required parameters to be sent with the POST request.

Hackerrank is a programming competitions platform. The testcases parameter is important in that sense. As the api is being used to compile and run the static code obtained from code fragments (the parsons problems do not accept the user input nor can the code be changed), a dummy testcases parameter is provided. The platform also provides numerous programming language choices, but since the language here is java, only its corresponding language code “3” is sent with the request. The apikey is also hardcoded along with testcases and language code.

Fig17: app.js ajax post request handler

```

61
62 app.post('/compile_run', multer().single(), function(req, res, next) {
63
64 var returnContent;
65
66 var jsonToSend = querystring.stringify({
67   'request_format': 'json',
68   'source': req.body.code,
69   'lang': req.body.language,
70   'wait': false,
71   'callback_url': '',
72   'api_key': req.body.hackerRankApi,
73   'testcases': req.body.testCases
74 });
75
76 console.log("=====");
77 console.log("Submission:");
78 console.log(req.body);
79
80 var HROptions = {
81   hostname: 'api.hackerrank.com',
82   port: 80,
83   path: '/checker/submission.json',
84   method: 'POST',
85   headers: {
86     'Content-Type': 'application/x-www-form-urlencoded',
87     'Content-Length': Buffer.byteLength(jsonToSend)
88   }
89 };
90
91 var HRrequest = http.request(HROptions, function(HRresponse) {
92   HRresponse.setEncoding('utf8');
93   HRresponse.on('data', function (data) {
94     try {

```

Fig17: Hackerrank Api parameters

Secure <https://www.hackerrank.com/api/docs>

cases.

POST <http://api.hackerrank.com/checker/submission.json>

Parameter	Value	Type	Description
source	<input type="text" value="required"/>	text	The source code for a submission
lang	<input type="text" value="required"/>	text	The language code for the submission Ex: 5 (Python)
testcases	<input type="text" value="required"/>	string	A JSON list of strings, each being a test case.
api_key	A dummy api key will be used	string	Your HackerRank API key
wait	<input type="text" value="true"/>	enumerated	<p>true The response is sent only after the submission is compiled and run</p> <p>false The request returns immediately and submission response will be posted through the callback URL.</p>
callback_url	<input type="text" value="optional"/>	string	A callback url, on which the submission response will be posted as a JSON string under 'data' parameter.
format	<input type="text" value="json"/>	enumerated	Output format as JSON or XML

Conclusion and Future Scope:

This interactive programming problems developed in the format of parsons puzzles are targeted towards the beginner users of Java programming language. The static examples of textbooks can be transformed to jumbled puzzles to suit a specific goal which is then solved to get the result. Currently this only has 14 puzzles which can be extended to any number of static programs. *The aim is to make it extensible so as to not write program specific views and make just one view adopt to any kind of java program with any number of lines.*

Since the hackerrank api accepts the testcases, one improvement of this platform is to make these puzzles more dynamic so the user can provide input and solve a particular problem while retaining the puzzle structure instead of going for a quiz like structure as that of 'Example' or 'Faded Example'.

A future scope for an application like this is to extend this to maintain session and gamify the solving process by awarding/deducting points based on the number of attempts taken to correctly solve the puzzle. Further extension would be to maintain the past trials of the users by allowing them to create an account and save that information as part of the profile to determine what concepts are the strong point of the user.

Further more improvement would be to forgo the hackerrank api and use a docker based sandboxed environment deployed on cloud to compile and run the code derived from solution code fragments.

References:

1. Hackerrank API docs [<https://www.hackerrank.com/api/docs>]
2. Mongodb http interfaces [<https://docs.mongodb.com/ecosystem/tools/http-interfaces/>]
3. Mongodb REST API app [<https://www.mongodb.com/blog/post/building-your-first-application-mongodb-creating-rest-api-using-mean-stack-part-1>]
4. Working with IDE One API [<https://tareq.co/2011/07/working-with-ide-one-api/>]
5. Online Code Editor with Node js and Hackerrank API [http://code.runnable.com/V436x_gKK2FxZB2t/online-code-editor-for-node-js-api-and-hacker-rank]
6. Angular drag & drop with HTML5 [<https://marceljuenemann.github.io/angular-drag-and-drop-lists/demo/#/simple>]
7. How we used Docker to compile and run untrusted code [<https://blog.remoteinterview.io/how-we-used-docker-to-compile-and-run-untrusted-code-2fafbffe2ad5>]
8. Ideas and code-snippets from stackoverflow.com