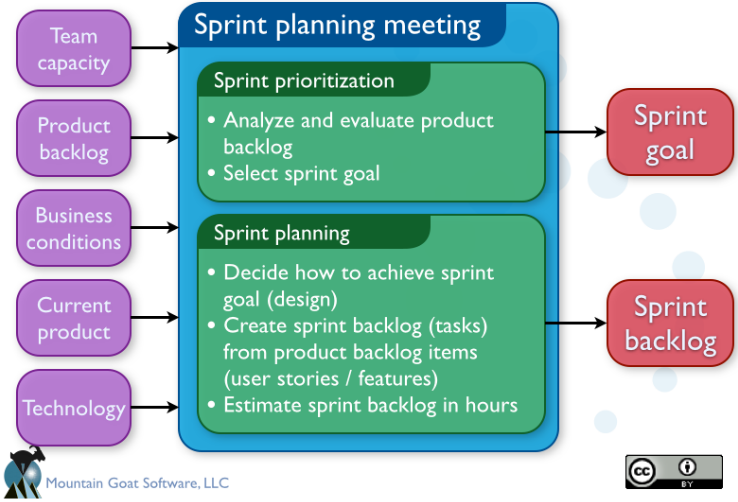
Objectives

- Course Notes
- Review Quiz
- Javascript Arrays
- Synchronous and Asynchronous functions
- Javascript Promises

Notes

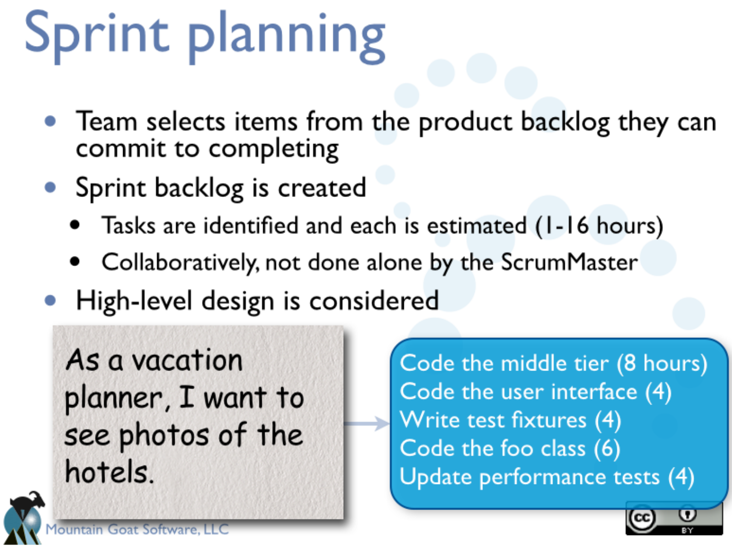
- Representatives from Facebook will be visiting on the October 8-9
  - Give a tech talk in class that Wednesday
- Term Project Statements are due today
- First sprint will begin when we return from fall break (here is a schedule)
  - **Sprint 1** October 7 – October 18
  - **Sprint 2** October 21 – Novemeber 1
  - **Sprint 3** November 4 – November 22 (last day of class)
- We deliver working software at the end of each sprint

Scrum Events



Team capacity
Product backlog
Business conditions
Current product
Technology
Sprint planning meeting
Sprint prioritization
• Analyze and evaluate product backlog
• Select sprint goal
Sprint planning
• Decide how to achieve sprint goal (design)
• Create sprint backlog (tasks) from product backlog items (user stories / features)
• Estimate sprint backlog in hours
Sprint goal
Sprint backlog
Mountain Goat Software, LLC

Scrum Events



Sprint planning

- Team selects items from the product backlog they can commit to completing
- Sprint backlog is created
  - Tasks are identified and each is estimated (1-16 hours)
  - Collaboratively, not done alone by the ScrumMaster
- High-level design is considered

As a vacation planner, I want to see photos of the hotels.

Code the middle tier (8 hours)
Code the user interface (4)
Write test fixtures (4)
Code the foo class (6)
Update performance tests (4)

Team Task

- Have a sprint planning meeting with your team before October 7
- Take one of the user stories from the project description (this is your sprint goal)
- Write up a use case/s that could be a component of the selected user story
- Form a sprint backlog by breaking up the goal into smaller tasks (things that you could conceivably finish in a day)
- Note: Your sprint goal should include at least the following
  - That your application runs on a container
  - Frontend and Backend component
  - Demonstrates that you can successful connect to a database and send and receive data

Composite Pattern Question

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

```
class MenuComposite extends MenuNode {
  private Set<MenuNode> childNodes = new HashSet<MenuNode>();

  public MenuComposite(String compositeName) {
    this.name = compositeName;
  }
  public Set<MenuNode> getChildNodes() {
    return childNodes;
  }
  public void addNode(MenuNode n) {
    this.childNodes.add(n);
  }

  @Override
  public String getHTML() {
    StringBuilder sb = new StringBuilder();
    sb.append("<div>");
    sb.append(this.name + "<br/>");

    for (MenuNode n : childNodes) {
      sb.append(n.getHTML());
    }

    sb.append("</div>");
    return sb.toString();
  }
}
```

- One possible solution to the composite pattern question

Configuring a DB in a Container

```
<Resource name="jdbc/DistDB"
          auth="Container"
          type="javax.sql.DataSource"
          maxActive="100"
          maxIdle="30"
          maxWait="10000"
          username="username"
          password="password"
          driverClassName="com.mysql.jdbc.Driver"
          url="jdbc:mysql://10.10.3.14:3306/testdatabase361"/>
```

- Need to inform our server or servlet container where our database is
- In Tomcat we can specify this information as a JNDI resource in the `context.xml` file
- Use classes in the `java.sql` and `javax.sql` packages

Basic SQL

```
--Create table
CREATE TABLE mytable (
  id int NOT NUll AUTO_INCREMENT,
  name VARCHAR(30),
  PRIMARY KEY (id)
);

--Insert
INSERT INTO mytable (name) VALUES ('mydata');

--Select
SELECT * FROM mytable;
```

- Some basic SQL queries

Connections

```
private static Connection getDatabaseConnection() {
  Connection conn = null;
  try {
    Context initCtx = new InitialContext();
    Context envCtx = (Context) initCtx.lookup("java:comp/env");
    DataSource ds = (DataSource) envCtx.lookup("jdbc/DistDB");
    conn = ds.getConnection();
    } catch (Exception e) {
      e.printStackTrace();
    }
    return conn;
  }
```

- Getting the database connection in Java

Statements

```
try {
  Statement statement = conn.createStatement();
  ResultSet resultSet = statement.executeQuery(query);
  // do something with result

  conn.close();
  } catch (SQLException e) {
    e.printStackTrace();
  }
```

- Making an SQL query from Java
- In this case the query is likely be some kind of selection
- The `Statement` class has different methods to make different kinds of queries
- Even better to work with the `PreparedStatement` class

JDBC Dependency

```xml
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>6.0.4</version>
</dependency>
```

- How to add the Java Database Connector with `pom.xml`

Common Front-end Tasks

- As a JS project grows more complicated you will want to simplify your build process as much as possible
- There are many common tasks in front-end development
- *Grunt* and *Gulp* are popular task automation tools for the JS ecosystem
  - Similar to tools like `make` or `ant`
- Can automate tasks such as
  - Running a linter
  - Minifying code
  - Creating binaries
  - Unit tests
  - Deployment

Front-end Builds

- We have been using Maven for our Java applications
  - Manage dependencies and configuration
  - Automate the build process
- Modern front-end applications can also become quite complex
  - Include Javascript libraries
  - Stylesheets
  - Copy files from one location to another
- What tools are available for managing the Front-end Build?

Webpack and Bower

- If you are using `node` there are many options available to you
- Logos for *bower* and *webpack* are shown above
- These are basically package managers for Javascript/HTML/CSS

Sample Project Structure

```
/projectroot
  |
  |--- application/
  |--- node_modules/
  |
  |--- dist/
  |
  |--- package.json
  |--- webpack.config.js
  |--- .git/
```

- One possible way to organize your project if you want to use front-end tools
- Can be compatible with our Java webapps

webpack.config.js

```
module.exports = {
  entry: "./application/app.js",
  output: {
    filename: "bundle.js"
  }
}
```

- The listing above shows the configuration for webpack
- Write Javascript in `app.js` and build it, along with all dependencies to a file called `bundle.js`

Sample Application

```
-- app.js
let $ = require('jquery');
$('body').append('<b>hello</b>');

-- html file
<!DOCTYPE html><html><head>
  <title>My App</title>
</head>
<body>
  <p>Hello, world!</p>
  <script src="bundle.js"></script>
</body></html>
```

- The listing above shows sample code for both the Javascript application and the html file where we intend to include it
- Notice that the application depends on `jquery`
  - This will work as long as we install `jquery` using `npm` (placed in the `node_modules` directory)
- In addition, webpack has facilities for loading stylesheets, compiling stylesheets using preprocessors etc.

Starting the webpack build

```
{ "name": "pine-js",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "webpack --config webpack.config.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "jquery": "^3.3.1"
  },
  "devDependencies": {
    "webpack": "^4.1.0",
    "webpack-cli": "^2.0.10"
  }}
```

- Part of `package.json`, notice the `scrtipts.start` property, we can launch `webpack` from our project root with the command `npm start`

JS Arrays

- Arrays are a common data type in programming
  - In Javascript they have some peculiarities
  - Here are just special objects and thus have access to methods
- Can create arrays with either the literal notation `[]` or with the built-in `Array` constructor

Accessing Elements

- Arrays have a property `length` that specifies the size of the array
  - First element at 0 and last element at `arr.length - 1`
- No out of bounds message if we try to access elements beyond the length, system simply returns `undefined`
- Assigning to a position beyond the bounds will appropriately expand the array
  - length will be updated
  - intermediate values will be filled in with undefined

Simple Methods for Arrays

- The following code demonstrates four simple methods for adding and removing elements from arrays

```
var people = [];

// Pushes to end of array
people.push("Joe");
people.push("Jim");

// Adds new item to beginning of array
people.unshift("Mary");

// Removes item from end of array
// and returns it
var x = people.pop();

// Removes first item and shifts
// remaining elements
var y = people.shift();
```

Adding and Removing at Specific Locations

- `delete` will remove an element at an index however: will leave undefined and the length property will not be updated
- The `splice` methods allows us to remove elements in a range and takes two arguments
  - Start position
  - Number of items to remove

```
var arr1 = [1,2,3,4,5];
delete arr1[2];

console.log(arr1); // [ 1, 2, , 4, 5 ]

var arr2 = [1,2,3,4,5];

arr2.splice(1,2) // returns [2, 3]
console.log(arr2) // [1, 4, 5]
```

Iteration

- JS provides a `forEach` method on arrays that allows for iteration without need to define an index

```
var arr = ["Mary", "Bob", "Alice"];

// typical iteration
for (let i = 0; i < arr.length; i++) {
   console.log(arr[i]);
}

// forEach approach
arr.forEach(function (person) {
   console.log(person + "!");
});
```

Mapping arrays

- *Mapping* create new array from the items in an already existing array
- The function `map()` is defined on any array
- Applies provided callback to each element

```
var arr = ["Mary", "Bob", "Alice"];

arr.forEach(function (person) {
   console.log(person + "!");
});

var mappedArr = arr.map(function (x) { return x[0];});
// [ ’M’, ’B’, ’A’ ]
```

Testing arrays

- Verify a predicate (given as a callback returning a boolean) over all elements of an array
- Find an element (the first) that matches a description

```
var arr = ["Mary", "Bob", "Alice"];

// false
console.log(arr.every(function (x) {
   return (x.length < 4);})
);

// w === "Bob"
var w = arr.find(function (x) {
   return (x.length < 4);
});
```

Reducing Arrays

- Another common array operation is to reduce the value of an array based on an aggregating function
- In this case we pass in a callback and initial value to the `reduce` function on the array

```
var n = [1,2,3,4,5,6,7];

console.log(n.reduce(function(acc, nn) {
   return acc + nn;
}, 0)
);
```

Synchronous vs Asynchronous

- A lot of JS uses an *asynchronous* model of communication
- An asynchronous approach can be faster in certain cases (e.g. i/o operations in `node.js`)
- In an i/o based application processes will spend most of their time waiting for an operation to return: retrieving data from the disk, requesting resources from the network, connect to a database etc.
- Multithreaded approaches incur some overhead for each process or thread they need to launch

Synchronous Request

- An example showing how a request (for example, an ajax call) would be made in a *synchronous* manner (*Good Parts, D. Crockford*)
- The client will halt until the `reqSync()` function returns

```
request = prepareReq();
response = reqSync(request);
display(response);
```

Asynchronous Request

- Same example from previous slide but written in an asynchronous manner(*Good Parts, D. Crockford*)
- Note that the function `reqAsync()` will return immediately
- When the request is completed the result will be processed by the anonymous function

```
request = prepareReq();
reqAsync(request, function (response) {
    display(response);
    });
```

JS Promises

- *Promise*: Object that serves as a placeholder for a value
    - Usually the result of an asynchronous operation
- The promise syntax helps to manage the execution order of callbacks
- An async function can immediately return a promise object

Promises (Basic Operation)

- The promise represents a pending computation
- Attach callbacks to the promise instead of passing them as arguments

```
var promise = asyncFunc();

promise.then(function (x) {
   // do something with the result
});

promise.catch(function (e) {
   console.log(e);
});
```

Chaining Registration of Callbacks

- Promises are *fluent*: `then` and `catch` methods return promise objects
- Handle callback registration in a more succinct way

Creating Promises

- In compatible versions of JS, there is a global constructor function called `Promise`
- To create a promise we pass in a callback: resolver function

```
var p = new Promise(function resolver(resolve, reject) {
   // computation succeeds
   resolve(20);

   // computation fails
   reject(e);
});
```

States of a Promise object

- A promise can be either: pending, fulfilled or rejected
- Operates like a state machine
- `resolve` and `reject` will change the state of a promise from pending to fulfilled or rejected (respectively)
- A settled promise is immutable

`then` and `catch`

- `then` and `catch` return *new* promise objects
- We can use `then` to make a sequence of steps by chaining many thens together

```
bgn().then(  // callback1
).then( // callback2
).then( etc....

);
```