**DB 연결 웹 앱**

**\* 자바스크립트 변수-연산자-타입**

변수는 var, let, const 키워드로 선언할 수 있다. 어떤 키워드를 사용하는가에 의해서 변수의 유효범위(scope)가 달라진다.

|  |  |
| --- | --- |
| **var** | var a= 2;  var a= “I love you”  var a= true;  var a= []; // 배열  var a= {}; // 오브젝트  var a= undefined; |
| **let** | ES6 버전 |
| **const** | ES6 버전. 자바의 상수라고 생각하면 된다 |

자바 스크립트에서의 연산자는 일반적인 사칙연산, MOD 등등 자바와 비슷하다. 자바 스크립트에서 OR 연산자는 조금 독특하다.

const name= “crong”;

/\* name에 값이 할당되어 있으면 name을 쓰고 아니면 오른쪽 값을 사용한다. \*/

const myname= name || “defaultName”

**\* 비교 연산자 주의할 점**

자바 스크립트에서는 일반적으로 == 보다는 ===을 많이 사용한다. 여러가지 문제점을 야기할 수 있기 때문이다. 아래 예를 보면,

0 == false, 0 == “”, 0 == “0”, null == undefined 의 결과가 모두 true가 나온다. 자바 스크립트에선 타입까지 정확하게 비교하기 위해서는 === 를 사용한다. == 를 사용하면 자바 스크립트가 암묵적으로 타입을 자동 변경하여 비교하게 된다.

**\* 자바 스크립트의 타입**

자바 스크립트의 타입은 선언 시점이 아닌 실행 시점에 결정된다. 타입을 체크하는 또렷한 방법은 없고 typeof, isArray 함수를 호출하여 타입을 체크할 수 있다.

|  |
| --- |
| **undefined** |
| **null** |
| **boolean** |
| **number** |
| **string** |
| **object** |
| **function** |

**\* 자바 스크립트 비교-반복-문자열**

|  |  |
| --- | --- |
| **if, else if, else** | if 대신 삼항 연산자( a ? “true”:”false”)도 많이 사용 |
| **switch** | 자바와 마찬가지로 case, default로 구성. break문 사용 |
| **for** | 자바와 동일  for(var i=0, len=arr.length; i<len; i++) { } |
| **foreach** | 배열은 foreach를 지원한다. |
| **for-of** | 새로 나온 반복문으로 버전 이슈가 있을 수 있으므로 사용에 주의 필요 |

**\* 문자열 처리**

자바스크립트에서 문자열과 문자는 동일한 개념으로 사용된다. 즉, typeof “abc”; 와 typeof ‘a’;의 결과 모두 string이 나온다.

자바스크립트에서 문자열은 자바와 마찬가지로 문자열 객체이고 때문에 split, replace, trim과 같은 메소드가 존재한다.

**\* 자바스크립트 함수**

자바스크립트에서의 함수는 여러개의 인자를 받아서 그 결과를 출력한다. 아래는 함수를 선언한 예제 소스이다.

|  |
| --- |
| function printName(firstname) {  reutnr “name is “ + firstname;  }  console.log(printName()); // 결과는 name is undefined 라고 나옴..  console.log(printName(“Jisu”, “Crong”)); // name is Jisu 라고 나옴.. |

자바스크립트에서는 함수 명세와 다른 인자를 전달하며 함수를 호출 한다고 해서 함수가 없다고 에러가 나는게 아니라, 적절한 처리를 하여 (인자가 부족한 경우 파라미터에 undefined를 넣어서 등등) 호출하게 된다.

다음으로 함수 표현식이라는 개념이 있다. 변수 값에 함수 선언문을 담아 놓은 것을 의미한다.

|  |
| --- |
| function printName(firstname) {  console.log(“name is “ + firstname);  var inner= function() {  return “inner value”;  }  return inner(); // inner 호출이 inner의 선언보다 먼저 나오면 에러 발생  } |

하지만 아래와 같이 함수 표현식을 함수 선언문으로 변경하면 그 순서가 바뀌어도 에러가 발생 안하고 동작할 수 있다.

|  |
| --- |
| function printName(firstname) {  console.log(inner()); // 선언보다 먼저 호출 되지만 에러 발생 안함.  function inner() {  return “inner value”;  }  } |

그렇다면 왜 함수 선언문에서는 에러가 발생 안하고 함수 표현식에서는 에러가 발생할까? 그 이유는 자바스크립트가 동작전 파싱할 때 수행하는 호이스팅 이라는 개념 때문이다. 호이스팅이란 자바스크립트 함수 실행 전 파싱하며 간단히 정보를 모아 함수 안의 변수들을 함수 바디의 맨 위로 올리는 것을 의미한다. 즉 왼쪽 함수가 오른쪽과 같이 변경되고 변수 a에는 아직 함수가 정의 안되어 undefined 에러가 발생하게 되는 것이다.

|  |  |
| --- | --- |
| function printName(firstName) {  console.log(inner());  console.log(a);  function inner() {  return “Jisu”;  }  var a= function() {  return “Crong”;  }  } | function printName(firstName) {  function inner() {  return “Jisu”;  }  var a;  console.log(inner());  console.log(a);  a= function() {  return “Crong”;  }  } |

자바스크립트에선 return을 안하면 함수 호출 시 그 결과로 undefined가 리턴된다.

**\* 함수 arguments 속성**

함수가 실행되면 그 안에서 arguments라는 특별한 지역변수가 생성된다. 자바스크립트 함수는 선언한 파라미터보다 더 많은 인자를 보낼 수 있는데 이 때 넘어온 인자를 arguments로 배열의 형태로 하나씩 접근할 수가 있다. 단 arguments는 배열 타입은 아니기 때문에 배열의 메서드를 사용할 수는 없다.

|  |
| --- |
| function printArgs() {  for(var i=0; i<arguments.length; i++) {  console.log(arguments[i]);  }  }  printArgs(1,2,3,4); |

**\* window 객체(setTimeout)**

window에는 다양한 기본 제공 전역객체가 있다. alert, setTimeout과 같은 객체가 있다. 사용 예를 보면 window.setTimeout() 이라고 호출할 수도 있고 전역객체이기 때문에 setTimeout()과 같이 객체를 생략하여 호출할 수도 있다.

자바스크립트는 함수를 인자로 받을 수도 있고, 반환할 수도 있다는 특징이 있다. 이렇게 인자로 전달되는 함수를 콜백함수라고 한다.

|  |
| --- |
| function run() {  console.log(“run start”);  setTimeout(function(){  console.log(“run ... ing”);  }, 1000); // 1000ms 후에 인자로 전달된 함수가 실행된다.  console.log(“run end”);  }  run(); |

위의 코드를 실행하면 run start > run end > run ... ing 순서로 실행이 된다. 그 이유는 setTimeout 함수는 비동기로 실행이 되기 때문이다. 그렇다면 시간을 1000이 아닌 0ms를 준다면?? 그래도 그 결과는 바뀌지 않는다. 그 이유는 ‘비동기 함수는 이벤트 큐에 쌓이게 되고 다른 동기 함수는 호출 콜 스택에 쌓여있는 호출 스택이 완료되어 다 비어진 후에 이벤트 큐에 있는 비동기 함수가 이벤트 루프에 의해 콜 스택으로 이동되어 실행’ 는 특징이 있기 때문이다.

참고] https://engineering.huiseoul.com/%EC%9E%90%EB%B0%94%EC%8A%A4%ED%81%AC%EB%A6%BD%ED%8A%B8%EB%8A%94-%EC%96%B4%EB%96%BB%EA%B2%8C-%EC%9E%91%EB%8F%99%ED%95%98%EB%8A%94%EA%B0%80-%EC%97%94%EC%A7%84-%EB%9F%B0%ED%83%80%EC%9E%84-%EC%BD%9C%EC%8A%A4%ED%83%9D-%EA%B0%9C%EA%B4%80-ea47917c8442

**\* DOM과 querySelector**

DOM이란 브라우저가 HTML 코드를 파싱하여 트리 형태의 오브젝트 모델을 만드는데 이 모델을 DOM 이라고 한다. 데이터가 계속 동적으로 변경될 수 있기 때문에 모델을 변경하면서 화면을 그리기 위해 필요하다. 그리고 자바스크립트가 동적인 변화를 줄 때 DOM 계층을 쉽게 방문할 수 있도록 DOM API를 제공 해준다.

|  |  |
| --- | --- |
| **getElementById(“”)** | document.getElementById(“”)를 하면 해당 ID를 가진 DOM 모델을 가져와 style 등등의 속성 값을 읽거나 반영할 수 있다. |
| **getQuerySelector** | ID가 아닌 div를 찾거나 할 때 사용. 아이디도 #ID를 통해 모델을 찾을 수 있다.  응용하면 document.getQuerySelector(“body”, “div”)를 하면 body 태그 하위의 모든 div를 찾는다. |

**\* Browser Event, Event object, Event handler**

브라우저에 마우스 올릴 때, 브라우저 사이즈를 조절할 때, 입력할 때 등등의 이벤트가 발생할 때 브라우저는 이 이벤트를 자바스크립트에게 알려준다. 이 이벤트를 받아 처리를 해주려면 Event listener(또는 Event handler 라고 부름)를 등록해야 한다.

|  |
| --- |
| val ele= document.getElementById(“outside”);  ele.addEventListener(“click”, function() {  // do something...  }, false); |

또한 이벤트의 정보를 호출하는 listener에게 전달 해주므로 매개변수를 아래와 같이 선언해 사용할 수도 있다.

|  |
| --- |
| val ele= document.getElementById(“outside”);  ele.addEventListener(“click”, function(evt) {  var target= evt.target;  console.log(target.className);  }, false); |

**\* Ajax (XMLHTTPRequest) 통신의 이해**

Ajax는 간단히 말하면 서버로부터 비동기로 데이터를 가져오는 것을 의미한다. 예를 들어 여러개의 탭을 가진 웹 페이지에서 탭을 누를 때마다 그 탭에 해당하는 페이지를 서버에서 가져오는 방법은 개발도 힘들고 클라이언트가 매번 페이지를 로딩해야 하기 때문에 성능 이슈도 존재한다. 하지만 Ajax를 통해 데이터가 변경되는 부분만 서버로부터 비동기로 요청하여 가져온 후 페이지에 반영하는 방법을 사용하여 문제를 해결할 수 있다.

클라이언트가 서버에게 데이터 요청을 위해서 메시지 포맷이 있어야 하는데 JSON 이라는 표준 데이터 포맷을 사용한다. Ajax 실행은 아래와 같이 하면 된다.

|  |
| --- |
| function ajax(data) {  var oReq= new XMLHttpRequest();  oReq.addEventListener(“load”, function() {  console.log(this.responseText);  });  oReq.open(“GET”, “http://localhost:8080/endpoint?data=5003”);  oReq.send();  } |

위의 코드를 보면 ajax 함수는 open, send 후 함수가 끝나고 응답이 와서 load 이벤트 발생 시 responseText를 출력하는 함수가 실행되게 된다.

**\* JSP(Java Server Page)란?**

JSP 파일은 정적 리소스(\*.html, \*.css)가 배포되는 곳에 같이 배포된다. JSP는 JSP 자체가 동작하는 것이 아니라 WAS에 의해 서블릿으로 변경된 다음에 동작한다.

JSP 소스에 <%@ %> 로 감싸여져 있는 부분은 JSP를 서블릿으로 바꿀 때 어떻게 바꾸어야 하는지를 설정하는 부분으로 페이지 지시문이라고 한다.

JSP 페이지 지시문은 아래와 같은 형식으로 작성한다.

|  |
| --- |
| <%@ page language=”java” // 언어설정. 하지만 실제로 JSP는 자바로만 작성한다.  contentType=”text/html; charset=UTF-8” // Http 응답의 contentType 설정.  pageEncoding=”UTF-8”%> // JSP 파일이 UTF-8로 작성되었음을 의미. |

다음으로 <% %> 로 감싸여져 있는 부분은 실제 동적인 로직을 넣을 자바 코드를 표시하는 태그로 Scriptlet 이라고 부른다.

|  |
| --- |
| <%  int total= 0;  for(int i=1; i<=10; i++) {  total= total + i;  }  %> |

위와 같이 작성한다고 응답 데이터로 전송이 되는 것은 아니다. 어떤 변수 값을 응답 데이터에 넣고 싶을 때에는 <%= %> 태그를 사용하고 이러한 태그를 표현식 이라고 한다. 전체적인 예제 소스를 보면 아래와 같다.

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=UTF-8"  pageEncoding="UTF-8"%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/....">  <html>  <head>  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  <title>sum10</title>  </head>  <body>  <%  int total = 0;  for(int i = 1; i <= 10; i++){  total = total + i;  }  %>  1부터 10까지의 합 : <%=total %>  </body>  </html> |

**\* JSP 라이프싸이클**

위에서 예제로 배운 JSP가 서블릿으로 변경되면 아래와 같이 변경되게 된다.

|  |
| --- |
| public void \_jspInit() {  }  public void \_jspDestroy() {  }  public void \_jspService(final javax.servlet.http.HttpServletRequest request  , final javax.servlet.http.HttpServletResponse response)  throws java.io.IOException, javax.servlet.ServletException {  .....  try {  .....  out.write("\n");  out.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\" \"http:..\">\n");  out.write("<html>\n");  out.write("<head>\n");  out.write("<meta http-equiv=\"Content-Type\" content=\"text/html; charset=UTF-8\">\n");  out.write("<title>sum10</title>\n");  out.write("</head>\n");  out.write("<body>\n");  out.write("\n");  int total = 0;  for(int i = 1; i <= 10; i++){  total = total + i;  }  out.write("\n");  out.write("\n");  out.write("1부터 10까지의 합 : ");  out.print(total );  out.write("\n");  out.write("\n");  out.write("</body>\n");  out.write("</html>");  } catch (java.lang.Throwable t) {  .....  }  } finally {  \_jspxFactory.releasePageContext(\_jspx\_page\_context);  }  } |

**\* JSP의 실행 순서**

1. 브라우저가 웹서버에 JSP에 대한 요청 정보를 전달한다.

2. 브라우저가 요청한 JSP가 최초로 요청했을 경우에만 아래와 같이 동작한다.

1) JSP로 작성된 코드를 서블릿 자바 코드로 변환.

2) 서블릿 자바 코드를 컴파일 해서 class 파일 생성.

3) 서블릿 클래스를 로딩하고 인스턴스를 생성한다.

3. 서블릿이 실행되어 요청을 처리하고 응답 정보를 생성한다.

기본적으로 <% %> 태그 안의 코드는 service 메소드 안에 코드들이 생성된다. 하지만 그 외 init, destroy 과 같은 다른 메소드 안에 코드를 작성하고 싶은 경우가 있을 수 있다. 그 경우엔 아래와 같이 <%! %> 태그를 사용하고 이러한 태그를 선언문 이라고 한다.

|  |
| --- |
| <%!  // JSP로 만들어진 서블릿은 기본 메소드에 jsp라는 prefix가 붙게 된다.  public void jspInit() {  ...  }  %> |

**\* JSP 내장객체**

JSP를 통해 만들어지는 서블릿의 jspService 메소드 윗 부분을 보면 개발자가 정의 안 한 미리 선언된 객체들이 있다. 이러한 객체를 내장객체 라고 하는데 JSP 에서도 이러한 객체들을 사용할 수가 있다. 내장객체의 종류는 아래와 같다.

