ID1354
Internet Applications

JavaScript

Leif Lindbäck, Nima Dokoohaki

leifl@kth.se, nimad@kth.se

SCS/ICT/KTH

Overview of JavaScript

- Originally developed by Netscape, as LiveScript
- Became a joint venture of Netscape and Sun in 1995, renamed JavaScript
- Now standardized by ECMA as ECMA-262, ECMAScript.
- The only relathionship between JavaScript and Java is similar syntax.

Overview of JavaScript (Cont'd)

- JavaScript is the language for client-side behavior in web applications.
- Can change HTML documents using the Document Object Model, DOM.
- Can communicate with server using for example AJAX.
- Also becoming more used at the server side. This is not covered in the course.

How to Include JavaScript Code

- Write JavaScript in separate files, with the extension **`.js`**
- Include a JavaScript file with the **`src`** attribute of the **`<script>`** element in the HTML file that uses the JavaScript code:

```
<script src = "my-script.js"></script>
```

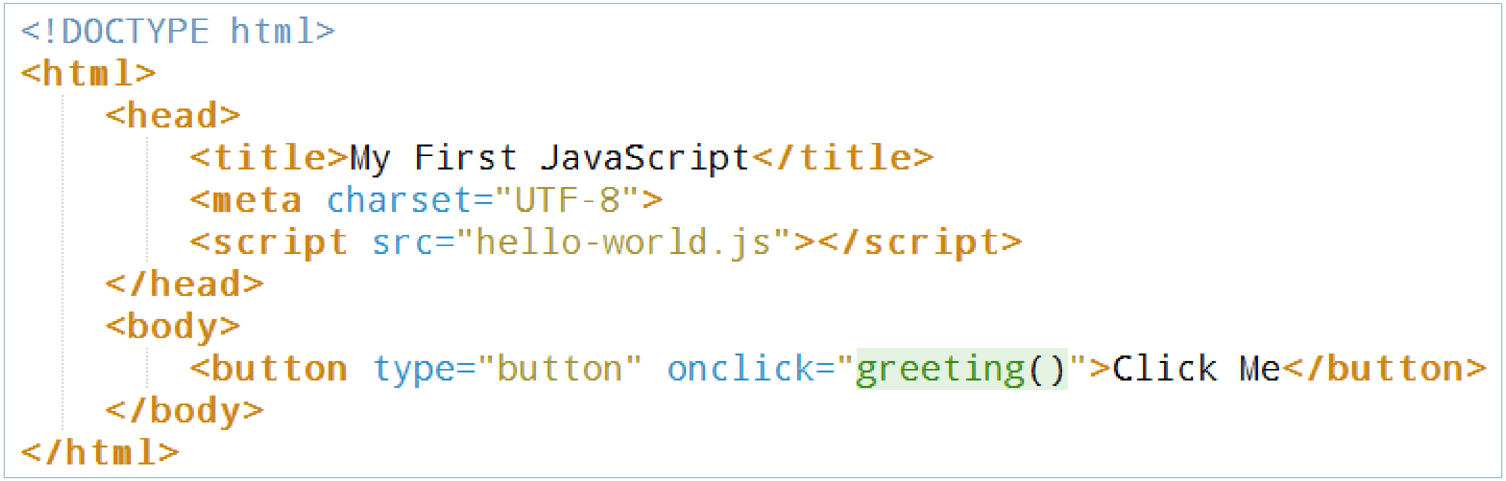
- Possible to write JavaScript code directly in the script tag, but that gives bad cohesion.

When is a Script Executed?

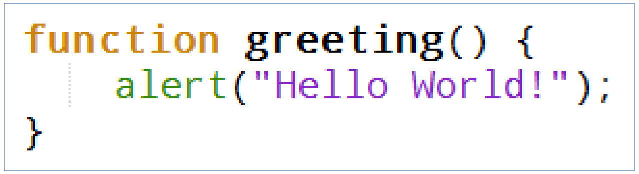
```
<script src=”myScript” async defer></script>
```

- If the **`async`** attribute is specified, and the script is external (there is a **`src`** attribute), the script is executed while the browser continues to parse the HTML document.
- If the **`async`** attribute is not specified, but the **`defer`** attribute is, and the script is external (there is a **`src`** attribute), the script is executed when the browser has finished parsing the HTML document.
- If neither **`async`** nor **`defer`** attribute is specified, the script is fetched and executed immediately, before the browser continues parsing the HTML document.

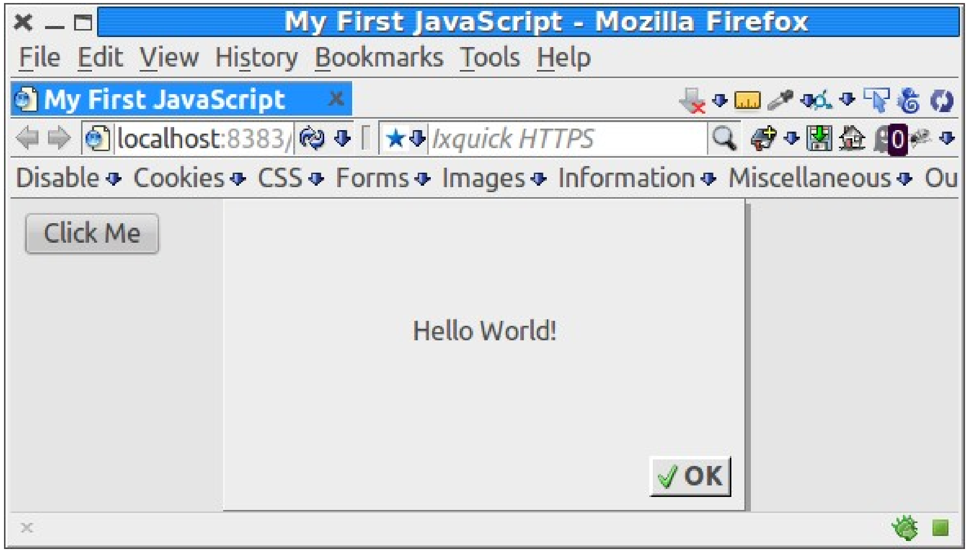
The First Example



```
<!DOCTYPE html>
<html>
    <head>
        <title>My First JavaScript</title>
        <meta charset="UTF-8">
        <script src="hello-world.js"></script>
    </head>
    <body>
        <button type="button" onclick="greeting()">Click Me</button>
    </body>
</html>
```



```
function greeting() {
    alert("Hello World!");
}
```



My First JavaScript - Mozilla Firefox
File Edit View History Bookmarks Tools Help
My First JavaScript
localhost:8383/
Ixquick HTTPS
Disable Cookies CSS Forms Images Information Miscellaneous Ou
Click Me
OK

Hello World!

Syntax

- Identifiers begin with a letter or underscore, followed by any number of letters, underscores, and digits.
- Case sensitive
- Statements are separated with semicolon.
- Reserved words are: abstract, arguments, boolean, break, byte, case, catch, char, class, const, continue, debugger, default, delete, do, double, else, enum, eval, export, extends, false, final, finally, float, for, function, goto, if, implements, import, in, instanceof, int, interface, let, long, native, new, null, package, private, protected, public, return, short, static, super, switch, synchronized, this, throw, throws, transient, true, try, typeof, var, void, volatile, while, with, yield
- Comments: single-line, `//`, and multiple-line, `/* some comment */`

Code Conventions

- Always use the same naming convention for all your code, preferrably similar to Java:
  - Variable and function names written as camelCase.
  - Global variables written in UPPERCASE.
  - Constants (like PI) written in UPPERCASE
- Write declarations at the beginning of the scope.

Strict Mode

- A restricted variant of JavaScript, with different semantics from normal code.
  - Prohibits some syntax, for example assignment to undeclared variable.
  - Eliminates some JavaScript silent errors by changing them to throw errors.
  - Fixes mistakes that make it difficult for JavaScript engines to perform optimizations
- To apply strict mode, put the exact statement "use strict"; (or 'use strict';) before any other statements.

Variables

- JavaScript is dynamically typed, type is never declared and variables change type when needed.
  `year = "in the eighties";` `year` is a `string`.
  `year = 84;` `year` is a `number`.
- Global variables can be declared either implicitly, just write the variable name, or explicitly, variable name preceeded with `var`.

```
var sum = 0;
today = "Monday";
flag = false;
```

- Block scoped variables are preceeded with `let` or `const`.

```
let sum = 0;
const sum = 0;
```

Local Variables Must Be Declared With `let` , `const` or `var`

- Local variables must be explicitly declared with the **`var`**, **`let`** or **`const`** keyworld.

- Here, c is a local variable.

```
function myFunction(a, b) {
  const c = 4;
  return a + b + c;
}
```

- Here, c is a global variable. Avoid this kind of declaration.

```
function myFunction(a, b) {
  c = 4;
  return a + b + c;
}
```

Hoisting

- JavaScript hoists all declarations (except **`let`** and **`const`**), which means they are moved to the top of the current scope (function or script).
- However, initializations are not hoisted.

  ```
  var x = 5;
  var sum = x + y;
  var y = 7;
  ```

  is hoisted to

  ```
  var x;
  var y;
  x = 5;
  var sum = x + y;
  y = 7;
  ```

  which does not make sense since y has no value when it is used.
- Always write declarations at the beginning of the scope, since that is how they are interpreted by JavaScript.

Primitive Values

- All primitive values have one of the five primitive types: Number, String, Boolean, Undefined, Null.
- Number, String, and Boolean have wrapper objects (**`Number`**, **`String`**, and **`Boolean`**), just like Java.
- For **`Number`** and **`String`**, primitive values and objects are coerced back and forth, therefore, primitive values can be treated as objects, for example:

```
const a = 10;
const b = a.toString();
```

Strings

- String literals are delimited by either **'** or **"**.
- Quotes can be used inside strings if they don't match the quotes surrounding the string:

```
"She is called 'Stina'";
'He is called "Pelle"';
```

- Strings can include escape sequences, e.g., **\t** or **\n**. Note that these will not cause tabs or line breaks in a HTML page since they are not HTML tags.

Numbers

- Numbers can be with or without decimals:

```
const PI = 3;
const PI = 3.14;
```

- Numbers are represented in double-precision 64-bit format, meaning the range is **±1.7976931348623157e+308** to **±-5e-324**

Boolean, Null, Undefined

- A Boolean can have the value **`true`** or **`false`**
- The only Undefined value is **`undefined`**. It is the value of a variable that has never been set to any value.
- The only Null value is **`null`**. It is used to unset a variable:

```
name = ”Sara”;  Name has the value ”Sara”.
name = null;    Name has the value null.
```

Assignment Operators

- Assignment operators are the same as in Java, **=**, **+=**, **-=**, etc

Bitwise Operators

- Bitwise operators are and, **&**; or, **|**; not, **~**; xor, **^**; left shift, **<<**; right shift, **>>**
- Bit operators work on 32 bits numbers.
- Any numeric operand in the operation is converted into a 32 bit number and the result is converted back to a JavaScript number.

Arithmetic Operators

- Numeric operators are the same as in Java, **++**, **- -**, **+**, **-**, *****, **/**, **%**
- All operations are in double precision.
- Same precedence and associativity as Java

Concatenation and Conversion

- The string concatenation operator is the same as in Java, **+**
- Concatenation coerces numbers to strings.
- Numeric operators, other than **+**, coerce strings to numbers.
- If either operand of **+** is a string, it becomes a concatenation operator.
- Explicit conversions are as follows:

1. Use the **`String`** and **`Number`** constructors
2. Use **`toString`** method:

```
var a = 10;
a = a.toString();
```

3. Use **`parseInt`** and **`parseFloat`** methods:

```
var a = "10";
a = parseInt(a);
```

Question 1

Number Utilities

- The **`Math`** object provides functions like **`floor`**, **`round`**, **`max`**, **`min`**, trigonometric functions, etc
- The **`Number`** object has useful properties like **`MAX_VALUE`**, **`MIN_VALUE`**, **`POSITIVE_INFINITY`**, **`NEGATIVE_INFINITY`**, **`PI`** and **`NaN`**.
  - **`NaN`** represents an illegal number, for example the result of an overflow.
  - It is not equal to any other number, not even itself. Test for it with the **`isNaN()`** function.

Typeof Operator

- The **typeof** operator returns the type of a variable or expression.
- It returns **”number”**, **”string”**, or **”boolean”** for Number, String, or Boolean, **”undefined”** for Undefined, **”function”** for a function, **”object”** for objects, and **”object”** also for **null**

typeof 10 returns the string **”number”**

The **`Date`** Object

- The **`Date`** Object

  - Create one with the **`Date`** constructor (no params)
  - Local time methods of **`Date`**:

    `toLocaleString` – returns a string of the date

    `getDate` – returns the day of the month

    `getMonth` – returns the month of the year (0 – 11)

    `getDay` – returns the day of the week (0 – 6)

    `getFullYear` – returns the year

    `getTime` – returns the number of milliseconds since Jan 1, 1970

    `getHours` – returns the hour (0 – 23)

    `getMinutes` – returns the minutes (0 – 59)

    `getMilliseconds` – returns the millisecond (0 – 999)

- Example: **`new Date().getDate();`**

The `String` Object

- Some `String` properties and methods:
  - `length` e.g., `var len = str1.length;` (a property, not a function)
  - `charAt(position)` the char at the specified pos, e.g., `str.charAt(3)`
  - `indexOf(string)` the pos of the specified string, e.g., `str.indexOf('B')`
  - `substring(from, to)` the specified substring, e.g., `str.substring(1, 3)`
  - `toLowerCase()` e.g., `str.toLowerCase()`

Output using the **Document** Object

- The **document** object represents the current HTML Document, an **Element** object represents a HTML element.
  - The **document** object is always present in a HTML page.
- The following line returns the HTML element with id **elemid**:

```
document.getElementById("elemid");
```

- The following line sets the HTML code of the element with id **elemid**:

```
document.getElementById("demo").innerHTML =
      ”Some output <br/>”;
```

Output Using the Console

- The **`console`** object has methods for writing to the JavaScript console, for example **`console.log(”a message”);`**
- This is useful when debugging a JavaScript program.
- Javascript errors are printed to the console.
- *Remember to check the console* if the program does not behave as expected.

IO Using the **`alert`**, **`confirm`** and **`prompt`** methods.

1. **`alert("Hej! \n");`**

   - Parameter is plain text, not HTML

   - Opens a dialog box which displays the parameter string and an **`OK`** button.

2. **`confirm("Do you want to continue?");`**

   - Opens a dialog box and displays the parameter and two buttons, **`OK`** and **`Cancel`**.

3. **`prompt("What is your name?", "");`**

   - Opens a dialog box and displays its string parameter, along with a text box and two buttons, **`OK`** and **`Cancel`**

   - The second parameter is for a default response if the user presses **`OK`** without typing a response in the text box.

Control Statements

- **`if`** statements, **`for`** loops and **`while`** loops are similar to Java.
- There are three kinds of conditions: primitive values, relational expressions and compound expressions.

1. Primitive values

   - If it is a string, it is **`true`** unless it is the empty string.

     `if (“hej”)` enters the if block.

     `if (“”)` does not enter the if block.

   - If it is a number, it is **`true`** unless it is zero

Control Statements (Cont'd)

2. Relational Expressions

- The usual six comparision operators: ==, !=, <, >, <=, >=
- Operands are coerced if necessary
  - If one operand is a string and one is a number, the string is coerced to a number.
  - If one operand is a boolean and the other is not, the boolean is coerced to a number (1 or 0)

- The unusual two comparision operators: === and !==
  - Same as == and !=, except that no coercions are done. The expression can only be true if the operands have the same type.

Control Statements (Cont'd)

2. Relational Expressions (Cont'd)

- Comparisons of references to objects compare addresses, not values.

3. Compound Expressions

- The logical operators are: and, **&&**; or, **||**; not, **!**

```
(x < 10 && y > 1)
```

Functions

- Functions are declared and prefixed with the **`function`** keyword, like in PHP.
- Since JavaScript is dynamically typed, neither parameters nor return value has a type:

```
function sum(a, b) {
  return a + b;
}
```

Anonymous Functions

- An anonymous function is defined in an expression, instead of a declaration.
- The reference to the anonymous function is stored in a variable, which can then be used to invoke the function.

```
const myFunc = function(a, b) {return a + b};
myFunc(4, 3); //Returns 7
```

Function Hoisting

- Functions are hoisted the same way as variables, therefore, a function can be called before it is declared:

```
square(5);
function square(y) {
        return y * y;
}
```

Function Parameters

- Parameters are passed by value, like in Java.
- The number of arguments is not checked.

Missing Arguments

- Missing arguments are set to **`undefined`**.
- If undefined variables are not desired, assign default values in the function:

```
function myFunction(x, y) {
  if (y === undefined) {
     y = 0; //default value
  }
  ...
}
```

- Can also be written like this:

```
function myFunction(x, y) {
  y = y || 0;
  ...
}
```

Extra Arguments

- Extra arguments have no name, but can be read from the **arguments** array, which is a built-in object:

```
 x = sumAll(1, 123, 500, 115, 44);

function sumAll() {
    let sum = 0;
    for (let i=0; i<arguments.length;i++){
        sum += arguments[i];
    }
    return sum;
}
```

Question 2

Closures

- Closures are similar to php, but there is a subtle difference.
- In both languages, a variable declared in an outer function is associated with an inner function, and can be used in the inner function after the outer has terminated.
- In JavaScript, this association is by reference, which means the variable of the closure can change value at any time.
- Compare with PHP, where the association is by value, which means the closure uses the variable value when the inner function is created. The value can not be changed after that.

Arrays

- Arrays are normally created with the array literal:
  `const myList = [24, "bread", true];`
- Elements are accessed by referring to index number, **`myList[0]`** has the value **`24`**. The first element is at index **`0`**.
- The **`length`** property is always set to the number of elements in the array.

Arrays (Cont'd)

- Elements can be added:

```
myList[myList.length] = ”Stina”;
```

- Elements can be iterated with a for-of loop:

```
let fruits = ["Banana", "Orange", "Apple"];
for (let fruit of fruits) {
    alert(fruit);
}
```

Some Array Methods

- **`join`** Joins all elements of an array into a string.
- **`sort`** Coerces elements to strings and puts them in alphabetical order.
- **`concat`** Joins two or more arrays, and returns a copy of the joined arrays.
- **`push`** Appends elements to the end.
- **`pop`** Removes the last element.
- **`unshift`** Prepends elements to the beginning.
- **`shift`** Removes the first element.

The Object Model

- The object model is quite different from Java.
- JavaScript is prototype-based. An object has a prototype (another object), which in turn has a prototype, and so on all the way up to the object Object, whose prototype is null.
- An object contains its own properties, and the properties of its prototype.
- No classes, class-based inheritance, interfaces or polymorphism. These features can be mimicked, but they are not built-in as in Java.

Properties

- Like in Java, objects can have properties (variables).
- An object is a collection of properties, a bit like an array with named elements.
- Properties can be accessed the following ways:

```
objectName.property          e.g., person.age
objectName["property"]       e.g., person["age"]
```

Creating an Object With an Object Literal

- Specify a list with a **`name:value`** pair for each property. Such a list is called an object literal.

  ```
  const person = {firstName : "Nisse", age : 3};
  ```

- An object literal is most appropriate when the object is used as a collection of data, without methods.

for-in loop

- Properties can be iterated with the for-in loop:

```
const person = {name:"Stina", age:57};
for (let x in person) {
    alert(x + ": " + person[x]);
}
```

- Difference between **`for-in`** and **`for-of`** loop:
  - **`for-in`** iterates only over enumerable properties of an object.
  - **`for-of`** iterates over all data the iterable object defines to be iterated over.

Creating an Object With a Constructor

- A constructor is an ordinary function. Objects are created with the new keyword, much like in Java or PHP.

```
function Person(firstName, age) {
 this.firstName = firstName;
 this.age = age;
}
const myMother = new Person("Fia", 48);
```

- A constructor is more appropriate when the object is not just a collection of data.

How an Object is Created With a Constructor

-
  ```
  function Person(firstName, age) {
   this.firstName = firstName;
   this.age = age;
  }
  const myMother = new Person("Fia", 48);
  ```

- What actually happens when the constructor is called is:

  1. The **new** operator creates an object.
  2. The created object is passed to the **Person** constructor as the value of this.
  3. The constructor creates the properties **firstName** and **age** in the object.
  4. The object's reference is stored in **myMother**.

By Reference

- A variable that holds an object is a reference to that object.

```
const person = {firstName : "Nisse", age : 50};
const samePerson = person;
samePerson.age = 40; //Updates also person.
```

Add and Delete Properties

- A Property is added by assigning a value to it.

```
const person = {firstName : "Stina", age : 50};
person.lastName = ”Svensson”;
```

- A Property is deleted with the keyword **`delete`**.

```
const person = {firstName : "Stina", age : 50};
delete person.age; //person.age is now
                   //undefined.
```

Methods

- Methods are functions defined as properties.
- Method calls have the same syntax as in Java, **`objectName.methodName();`**

Defining Methods

- Methods can be defined in constructors.

```
function Person(firstname) {
    this.name = firstname;
    this.changeName = function(name) {
        this.name = name;
    }
}

const person = new Person("Olle");
person.changeName("Pelle");
```

- Like properties, methods can also be added with the object literal or added to existing objects.

The `this` keyword

- In previous examples, **`this`** has been used like we would use it in Java.
- That is not a good practice, since **`this`** might point to wrong object when a method is called from an event handler, for example as a consequence of the user clicking a button.

The **`this`** keyword

- A solution is to store **`this`** in a variable in the constructor.

```
function Person(firstname) {
    const self = this;
    self.name = firstname;
    self.changeName = function(name) {
        self.name = name;
    }
}
```

- Note the usage of a closure. **`self`** is used in the anonymous function stored in **`changeName`** after the function **`Person`** has terminated.

Question 3

Object Prototype

- An object has a prototype, and contains also its prototype's properties.
- The prototype is also an object.
- An object created from its own constructor, has the constructor as prototype.
- Objects created with the object literal, or with **`new Object()`**, has the prototype of the object **`Object`**.

Prototype Chain

- Each object has a prototype chain, the top of which is **`Object.prototype`**.
- Objects contains properties from all prototypes in the prototype chain.
- When looking for a property, the whole chain is followed until the prototype is found or the top is reached.
  - This is slow for long chains.

Inheritance

- To inherit an object, set the prototype to the object that shall be inherited:

```
function Person(name) {
    this.name = name;
}

function Employee(name, salary) {
    this.parent = Person;
    this.parent(name);
    this.salary = salary;
}
Employee.prototype = new Person();

const sara = new Employee(”Sara”, 1200);
```

Inheritance (Cont'd)

- The **Employee** constructor from previous slide:

```
function Employee(name, salary) {
    this.parent = Person;
    this.parent(name);
    this.salary = salary;
}
```

- Assigning **Person** to the **parent** property means that property is actually the **Person** function.
- When **this.parent** is called, **Person** executes and adds the **name** property to the object indicated by **this**, namely the newly created **Employee** object.

Inheritance (Cont'd)

- Much can be said about pros and cons of this and other ways to inherit.
- Much can also be said about implementing polymorphism and other object-oriented constructs.
- However, that is outside the scope of this course.

Regular Expressions

- Both HTML and HTTP are string based.
- Web applications often contain a lot of code searching and manipulating strings.
- Regular expressions is a powerful tool for this.
- A regular expression is a sequence of characters that forms a search pattern.

Regexp Syntax

- A regular expression has the form /pattern/modifiers, for example **`/stina/i`**.
  - The **`i`** modifier means the expression is case insensitive.
- Note that the regexp is not a string. In fact, it is a **`RegExp`** object.

Methods Often Used for Regexps.

- The **search** and **replace** methods in the **string** object are good candidates for using regular expressions.

```
const str = "Hi, My name is Stina";
const n = str.search(/stina/i);//n is
15

const str = "Hi, my name is Olle";
const res = str.replace(/olle/i,
                      "a secret");
//res is ”Hi, my name is a secret”
```

Regular Expression Characters

- There are two categories of characters in a regexp pattern:
  - Metacharacters have special meanings in patterns and do not match themselves. The following are metacharacters:
    `\ | ( ) [ ] { } ^ $ * + ? .`
  - Normal characters that do match themselves. All characters except the metacharacters are normal characters.
- A metacharacter is treated as a normal character if it is preceeded by a backslash, **`\`**.

Character Classes

- **`[abc]`** means any of the characters a, b or c.
- **`[a-z]`** means any character in the range a-z.
- A caret at the left end of a class definition means not. **`[^0-9]`** means any charcter not in the range 0-9
- The character order when defining ranges is the Unicode order.

Predefined Character Classes

There are many predefined character classes with abbreviations.

Abbr.	*Equiv. Pattern*	*Matches*
`\d`	`[0-9]`	a digit
`\D`	`[^0-9]`	not a digit
`\w`	`[A-Za-z_0-9]`	a word character
`\W`	`[^A-Za-z_0-9]`	not a word character
`\s`	`[\r\t\n\f]`	a whitespace character
`\S`	`[^\r\t\n\f]`	not a whitespace character

Quantifiers

Quantifier	**Meaning**
`{n}`	exactly n occurences of the preceeding pattern
`{m,}`	at least m occurences
`{m, n}`	at least m but not more than n occurences
`+`	at least one occurrence
`*`	any number of occurences
`?`	zero or one occurrence

Anchors

- The pattern is forced to match only at the beginning with **^**
  `/^Lee/` matches **`"Lee Ann"`** but not **`"Mary Lee Ann"`**
- The pattern is forced to match only at the end with **$**
  `/Lee$/` matches **`"Mary Lee"`**, but not **`"Mary Lee Ann”`**

Handling Errors

- Error handling is done much the same way as in Java, using try-catch blocks.

```
try {
     // Block of code.
} catch(err) {
     // Handle errors from the try block.
}
```

Throwing Exceptions

- The JavaScript interpreter will throw an exception if there is an error in the code.
  - The first **alert** statement below throws an exception since **x** is not defined.

```
try {
  alert(x);
} catch (err) {
  alert(err);
}
```

- Exceptions can also be thrown with the **throw** statement. The exception can be a **String**, a **Number**, a **Boolean** or an **Object**:

```
throw ”Error message”;
```

`finally` Block

- A **`finally`** block is always executed when leaving the try/catch blocks.

```
try {
    // Block of code.
} catch(err) {
    // Handle errors from the try block.
} finally {
    // Always executed.
}
```

Best Practices

- Avoid using global variables.
- Declare local variables with `let` or `const`, to use block scope.
- Use `const` for all variables that are not intended to be modified.
- Always treat numbers, strings, and booleans as primitive values, never as objects.
  - Objects are slower and comparisions may fail when mixing objects and primitives.
- Use `===` and `!==` instead of `==` and `!=`

  `0 == ””` is true
  `0 === ””` is false