JAVA DAVE ENVIRONMENT

Code example, Client-server architecture, Introduction to JavaScript

Java workshop training, August, 2017.

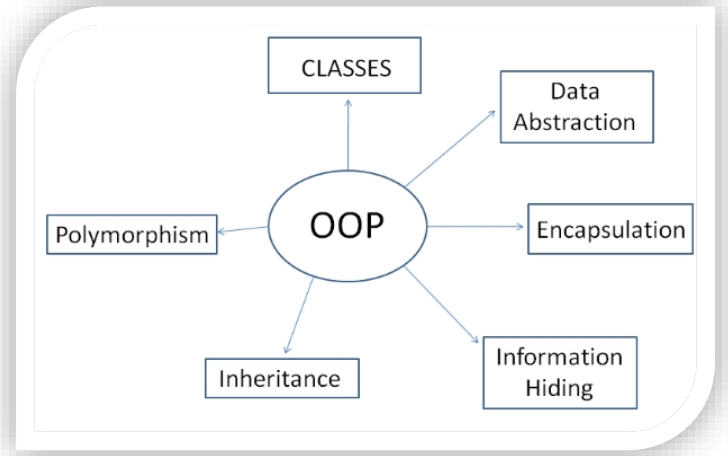
Dr. Kishore Biswas (Forrest/柯修)

PhD. Artificial Intelligence—Natural Language Processing.

CanWay IT Training ®.

CONTENTS

1) Functional Programming code example & practice
2) Multithreading Programming code example & practice
3) Exception handling code example & practice
4) Client-server architecture explained.
5) Introduction to JavaScript
6) Code exercise.



1) Functional Programming Code Example & Practice

FUNCTIONAL PROGRAMMING (FP)...

- Functional programs contain no assignment statements, so variables, once given a value, never change. More generally, functional programs contain no side effects at all. A function call can have no effect other than to compute its result.

- This eliminates a major source of bugs, and also makes the order of execution irrelevant—since no side effect can change an expression's value, it can be evaluated at any time. This relieves the programmer of the burden of prescribing the flow of control. Since expressions can be evaluated at any time, one can freely replace variables by their values and vice versa—that is, programs are "referentially transparent." This freedom helps make functional programs more tractable mathematically than their conventional counterparts

Functional Programming (FP)...

- One major difference between imperative programming and FP is that in FP there are no side effects. This means, among other things,

- No mutation of variables
- No printing to the console or to any device
- No writing to files, databases, networks, or whatever
- No exception throwing

Adding Two Integers With FP

```
public static int add(int a, int b) {
  while (b > 0) {
    a++;
    b--;
  }
  return a;
}
```

```
public static void add(int a, int b) {
  while (b > 0) {
    a++;
    b--;
  }
  System.out.println(a);
}
```

Even a simpler approach

```
static int add(int x, int y) {
  while(y-- > 0) {
    x = ++x;
  }
  return x;
}
```

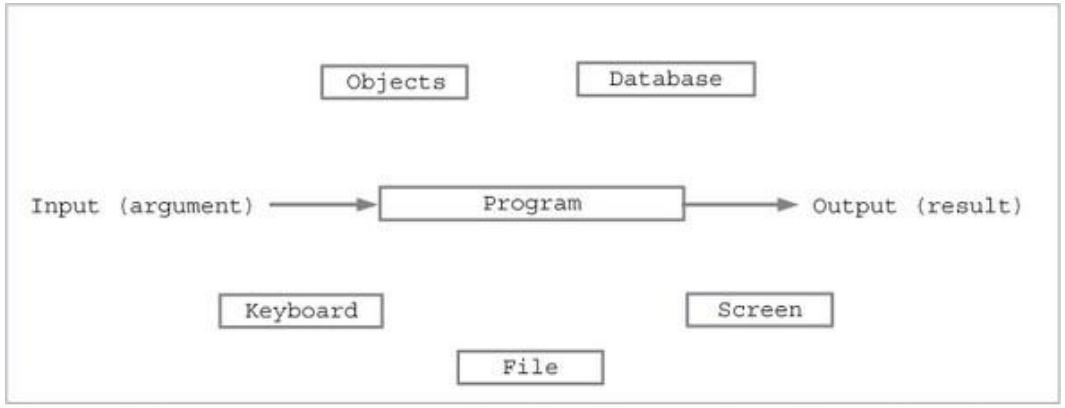
RECURSIVE APPROACH

```
static int addRec(int x, int y) {
  return y == 0
      ? x
      : addRec(++x, --y);
}
```

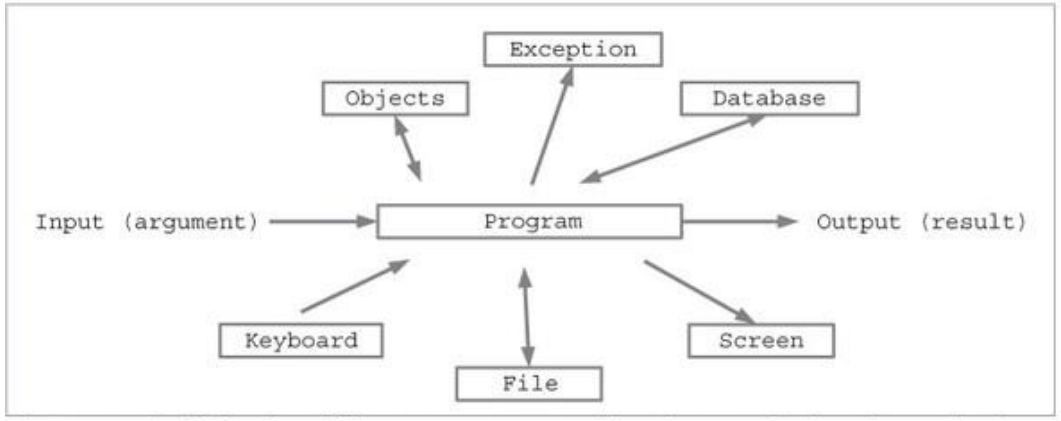
IMPLEMENTING RECURSION IN JAVA

- To understand what's happening, you must look at how Java handles method calls. When a method is called, Java suspends what it's currently doing and pushes the environment on the stack to make a place for executing the called method. When this method returns, Java pops the stack to restore the environment and resume program execution. If you call one method after another, the stack always holds at most one of these method call environments.
- But methods aren't composed only by calling them one after the other. Methods call methods. If method1 calls method2 as part of its implementation, Java again suspends the method1 execution, pushes the current environment on the stack, and starts executing method2. When method2 returns, Java pops the last pushed environment from the stack and resumes execution (of method1 in this case). When method1 completes, Java again pops the last environment from the stack and resumes what it was doing before calling this method.
- Method calls may be deeply nested, and this nesting depth does have a limit, which is the size of the stack. In current situations, the limit is somewhere around a few thousand levels, and it's possible to increase this limit by configuring the stack size. But because the same stack size is used for all threads, increasing the stack size generally wastes space. The default stack size varies from 320 KB to 1024 KB, depending on the version of Java and the system used. For a 64-bit Java 8 program with minimal stack usage, the maximum number of nested method calls is about 7,000. Generally, you won't need more, except in specific cases. One such case is recursive method calls.

How Referential Transparency Makes Programs Safer...



A referentially transparent program doesn't interfere with the outside world apart from taking an argument as input and outputting a result. Its result only depends on its argument.



A program that isn't referentially transparent may read data from or write it to elements in the outside world, log to file, mutate external objects, read from keyboard, print to screen, and so on. Its result is unpredictable.

Having no side effects (and thus not mutating anything in the external world) isn't enough for a program to be functional. Functional programs must also not be affected by the external world. In other words, the output of a functional program must depend only on its argument. This means functional code may not read data from the console, a file, a remote URL, a database, or even from the system. Code that doesn't mutate or depend on the external world is said to be referentially transparent

How Referential Transparency Makes Programs Safer

Referentially transparent code has several properties that might be of some interest to programmers:

- It's self-contained. It doesn't depend on any external device to work. You can use it in any context—all you have to do is provide a valid argument.
- It's deterministic, which means it will always return the same value for the same argument. With referentially transparent code, you won't be surprised. It might return a wrong result, but at least, for the same argument, this result will never change.
- It will never throw any kind of Exception. It might throw errors, such as OOME (out-of-memory error) or SOE (stack-overflow error), but these errors mean that the code has a bug, which is not a situation you, as a programmer, or the users of your API, are supposed to handle (besides crashing the application and eventually fixing the bug).
- It won't create conditions causing other code to unexpectedly fail. For example, it won't mutate arguments or some other external data, causing the caller to find itself with stale data or concurrent access exceptions.
- It won't hang because some external device (whether database, file system, or network) is unavailable, too slow, or simply broken.

APPLYING FUNCTIONAL PRINCIPLES TO A SIMPLE EXAMPLE

As an example of converting an imperative program into a functional one, we'll consider a very simple program representing the purchase of a donut with a credit card.

```
public class DonutShop {

  public static Donut buyDonut(CreditCard creditCard) {
    Donut donut = new Donut();
    creditCard.charge(Donut.price);      ← 1 Charges the credit card as a side effect
    return donut;                        ← 2 Returns the donut
  }
}
```

In this code, the charging of the credit card is a side effect . Charging a credit card probably consists of calling the bank, verifying that the credit card is valid and authorized, and registering the transaction. The function returns the donut .

The problem with this kind of code is that it's difficult to test. Running the program for testing would involve contacting the bank and registering the transaction using some sort of mock account. Or you'd need to create a mock credit card to register the effect of calling the charge method and to verify the state of the mock after the test.

Applying Functional Principles to a Simple Example

- If you want to be able to test your program without contacting the bank or using a mock, you should remove the side effect. Because you still want to charge the credit card, the only solution is to add a representation of this operation to the return value. Your buyDonut method will have to return both the donut and this representation of the payment.
- To represent the payment, you can use a payment class.

```
public class Payment {
  public final CreditCard creditCard;
  public final int amount;
  public Payment(CreditCard creditCard, int amount) {
    this.creditCard = creditCard;
    this.amount = amount;
  }
}
```

Applying Functional Principles to a Simple Example

- This class contains the necessary data to represent the payment, which consists of a credit card and the amount to charge. Because the buyDonut method must return both a Donut and a Payment, you could create a specific class for this, such as Purchase:

- public class Purchase {

- public Donut donut;
- public Payment payment;

- public Purchase(Donut donut, Payment payment) {
- this.donut = donut;
- this.payment = payment;
- }
- }

- You'll often need such a class to hold two (or more) values, because functional programming replaces side effects with returning a representation of these effects. Rather than creating a specific Purchase class, you'll use a generic one that you'll call Tuple. This class will be parameterized by the two types it will contain (Donut and Payment). The following listing shows its implementation, as well as the way it's used in the DonutShop class.

```
public class Tuple<T, U> {

  public final T _1;
  public final U _2;

  public Tuple(T t, U u) {
    this._1 = t;
    this._2 = u;
  }
}
public class DonutShop {

  public static Tuple<Donut, Payment> buyDonut(CreditCard creditCard) {
    Donut donut = new Donut();
    Payment payment = new Payment(creditCard, Donut.price);
    return new Tuple<>(donut, payment);
  }
}
```

Applying Functional Principles to a Simple Example

- Note that you're no longer concerned (at this stage) with how the credit card will actually be charged. This adds some freedom to the way you build your application. You could still process the payment immediately, or you could store it for later processing. You could even combine stored payments for the same card and process them in a single operation. This would allow you to save money by minimizing the bank fees for the credit card service.

```
package com.fpinjava.introduction.listing01_04;


public class Payment {
  public final CreditCard creditCard;
  public final int amount;


  public Payment(CreditCard creditCard, int amount) {
    this.creditCard = creditCard;
    this.amount = amount;
  }
  public Payment combine(Payment payment) {
    if (creditCard.equals(payment.creditCard)) {
      return new Payment(creditCard, amount + payment.amount);
    } else {
      throw new IllegalStateException("Cards don't match.");
    }
  }
}
```

2) Multithreading Programming Code Example & Practice

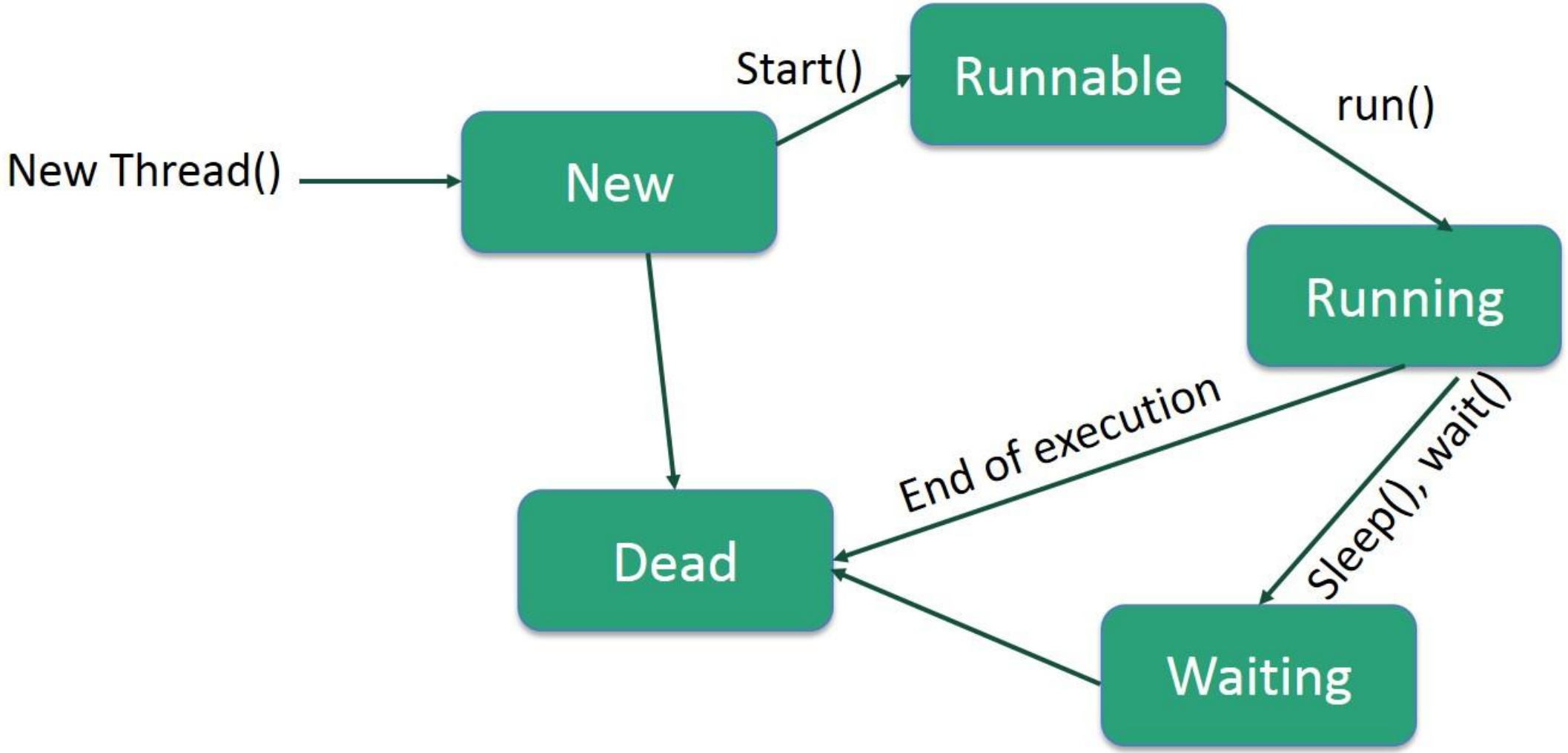
MULTITHREADING IN JAVA

- Java is a multi-threaded programming language which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

- By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

- Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

LIFE CYCLE OF A THREAD



THREAD LIFE CYCLE

- New − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

- Runnable − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

- Waiting − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

- Timed Waiting − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

- Terminated (Dead) − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

THREAD PRIORITIES

- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

- Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

CREATE A THREAD BY IMPLEMENTING A RUNNABLE INTERFACE

If your class is intended to be executed as a thread then you can achieve this by implementing a Runnable interface. You will need to follow three basic steps −

Step 1

- As a first step, you need to implement a run() method provided by a Runnable interface. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of the run() method −
- public void run( )

Step 2

- As a second step, you will instantiate a Thread object using the following constructor −
- Thread(Runnable threadObj, String threadName);
- Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

Step 3

- Once a Thread object is created, you can start it by calling start() method, which executes a call to run( ) method. Following is a simple syntax of start() method −
- void start();

Create a thread by extending a Thread class

The second way to create a thread is to create a new class that extends Thread class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

- You will need to override run( ) method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of run() method −
- public void run( )

Step 2

- Once Thread object is created, you can start it by calling start() method, which executes a call to run( ) method. Following is a simple syntax of start() method −

- void start( );

THREAD METHODS...

Sr.No.	Method & Description
1	**public void start()** Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	**public void run()** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	**public final void setName(String name)** Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	**public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10.
4	**public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread.
6	**public final void join(long millisec)** The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates or the specified number of milliseconds passes.
7	**public void interrupt()** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	**public final boolean isAlive()** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

STATIC THREAD METHODS...

1	**public static void yield()** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled.
2	**public static void sleep(long millisec)** Causes the currently running thread to block for at least the specified number of milliseconds.
3	**public static boolean holdsLock(Object x)** Returns true if the current thread holds the lock on the given Object.
4	**public static Thread currentThread()** Returns a reference to the currently running thread, which is the thread that invokes this method.
5	**public static void dumpStack()** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

JAVA – INTER THREAD COMMUNICATION

- If you are aware of interposes communication then it will be easy for you to understand interthread communication. Interthread communication is important when you develop an application where two or more threads exchange some information.
- There are three simple methods and a little trick which makes thread communication possible.

1	**public void wait()** Causes the current thread to wait until another thread invokes the notify().
2	**public void notify()** Wakes up a single thread that is waiting on this object's monitor.
3	**public void notifyAll()** Wakes up all the threads that called wait( ) on the same object.

4) Exception Handling Code Example & Practice

EXCEPTION METHODS...

1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage().

EXCEPTION METHODS

4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

USER-DEFINED EXCEPTIONS

- You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes −

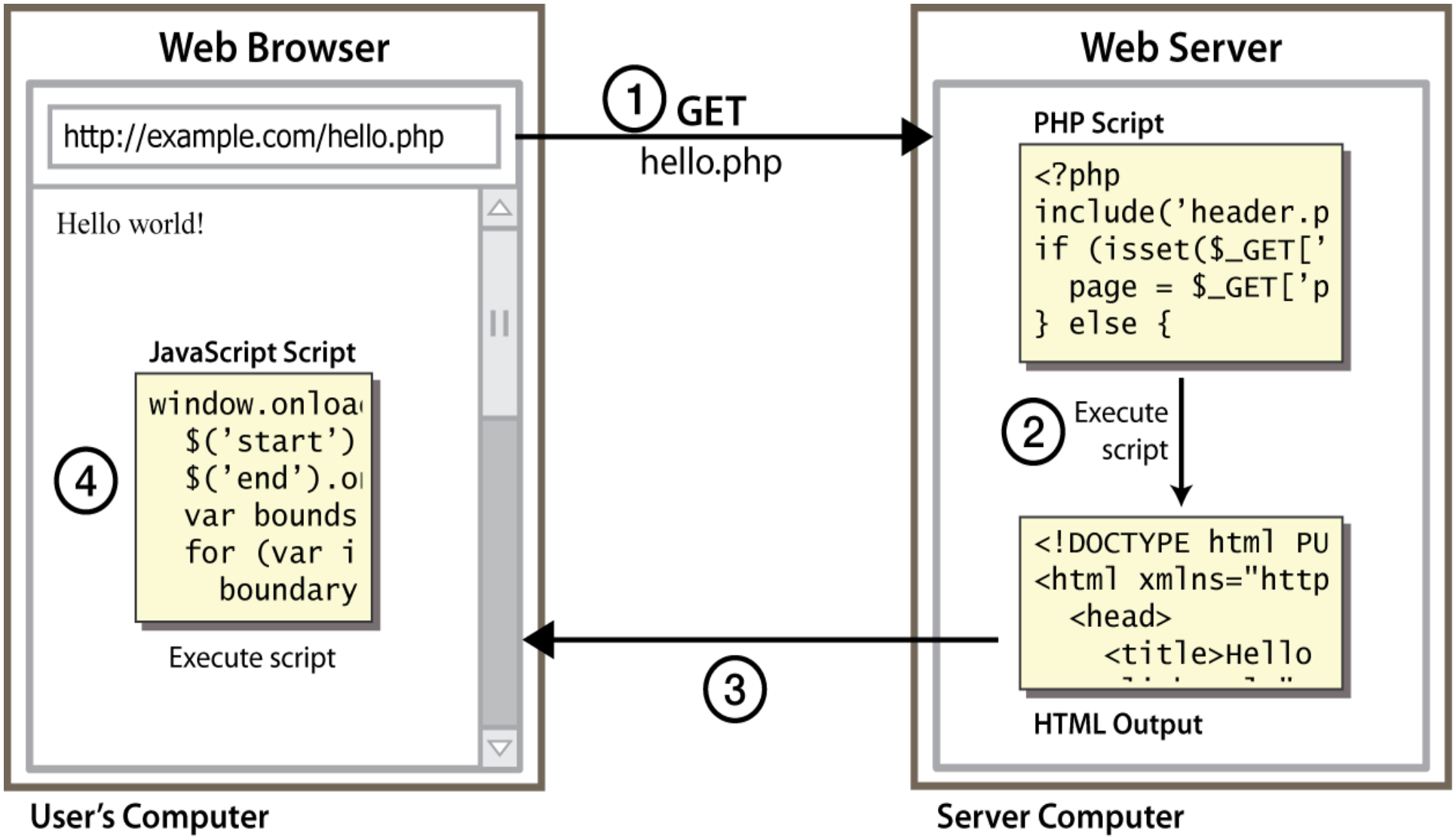
- All exceptions must be a child of Throwable.

- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.

- If you want to write a runtime exception, you need to extend the RuntimeException class.

5) JAVA SCRIPT

CLIENT SIDE SCRIPTING



Why use client-side programming?

PHP already allows us to create dynamic web pages. Why also use client-side scripting?

- client-side scripting (JavaScript) benefits:
  - **usability**: can modify a page without having to post back to the server (faster UI)
  - **efficiency**: can make small, quick changes to page without waiting for server
  - **event-driven**: can respond to user actions like clicks and key presses

Why use client-side programming?

- server-side programming (PHP) benefits:
  - **security**: has access to server's private data; client can't see source code
  - **compatibility**: not subject to browser compatibility issues
  - **power**: can write files, open connections to servers, connect to databases, ...

What is JavaScript?

- a lightweight programming language ("scripting language")
  - used to make web pages interactive
  - insert dynamic text into HTML (ex: user name)
  - **react to events** (ex: page load user click)
  - get information about a user's computer (ex: browser type)
  - perform calculations on user's computer (ex: form validation)

What is JavaScript?

- a web standard (but not supported identically by all browsers)
- NOT related to Java other than by name and some syntactic similarities

JavaScript Vs Java

- interpreted, not compiled
- more relaxed syntax and rules
  - fewer and "looser" data types
  - variables don't need to be declared
  - errors often silent (few exceptions)
- key construct is the function rather than the class
  - "first-class" functions are used in many situations
- contained within a web page and integrates with its HTML/CSS content

JAVASCRIPT VS JAVA

+

=

JavaScript vs. PHP

- similarities:
  - both are interpreted, not compiled
  - both are relaxed about syntax, rules, and types
  - both are case-sensitive
  - both have built-in regular expressions for powerful text processing

JavaScript vs. PHP

- differences:
  - JS is more object-oriented: noun.verb(), less procedural: verb(noun)
  - JS focuses on user interfaces and interacting with a document; PHP is geared toward HTML output and file/form processing
  - JS code runs on the client's browser; PHP code runs on the web server

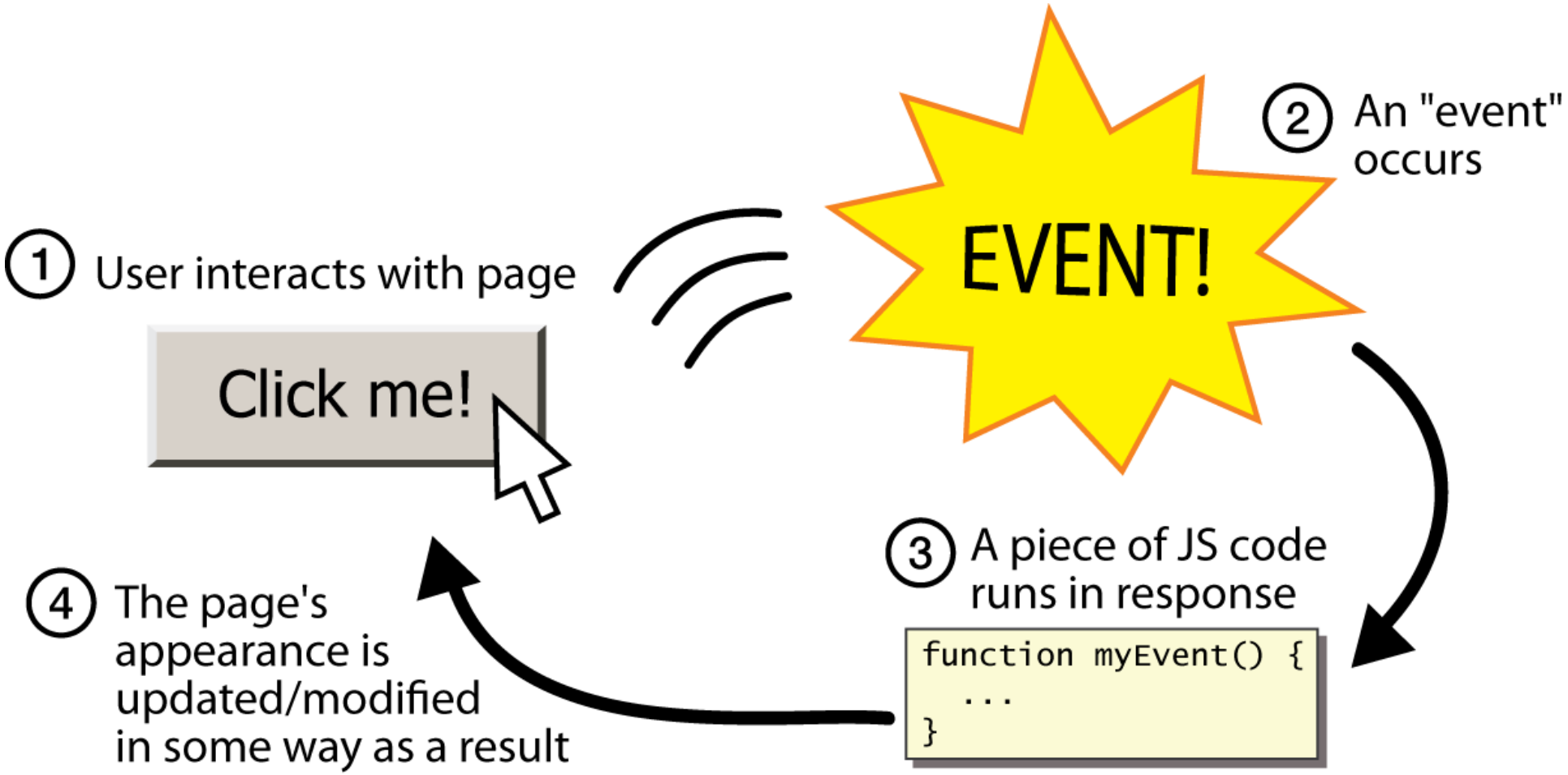
JS <3

Linking to a JavaScript file: `script`

- script tag should be placed in HTML page's head
- script code is stored in a separate .js file
- JS code can be placed directly in the HTML file's body or head (like CSS)
  - but this is bad style (should separate content, presentation, and behavior

```
<script src="filename" type="text/javascript"></script>
```
HTML

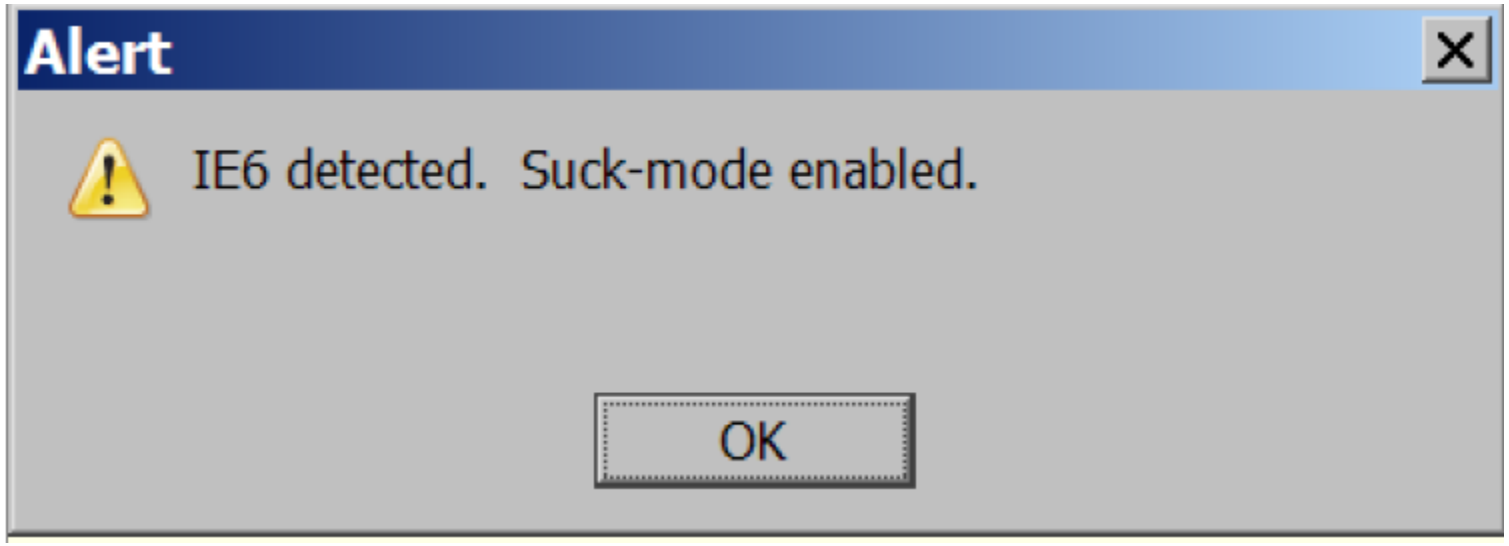
EVENT-DRIVEN PROGRAMMING



A JavaScript statement: `alert`

- a JS command that pops up a dialog box with a message

```
alert("IE6 detected. Suck-mode enabled.");
```
JS



Event-driven programming

- you are used to programs start with a main method (or implicit main like in PHP)
- JavaScript programs instead wait for user actions called *events* and respond to them
- event-driven programming: writing programs driven by user events
- Let's write a page with a clickable button that pops up a "Hello, World" window...

BUTTONS

- button's text appears inside tag; can also contain images
- To make a responsive button or other UI control:
  1. choose the control (e.g. button) and event (e.g. mouse 1. click) of interest
  2. write a JavaScript function to run when the event occurs
  3. attach the function to the event on the control

```
<button>Click me!</button>                                    HTML
```

JAVASCRIPT FUNCTIONS

```
function name() {
statement ;
statement ;
...
statement ;
}
```
JS

```
function myFunction() {
       alert("Hello!");
       alert("How are you?");
}
```
JS

- the above could be the contents of example.js linked to our HTML page
- statements placed into functions can be evaluated in response to user events

Event Handlers

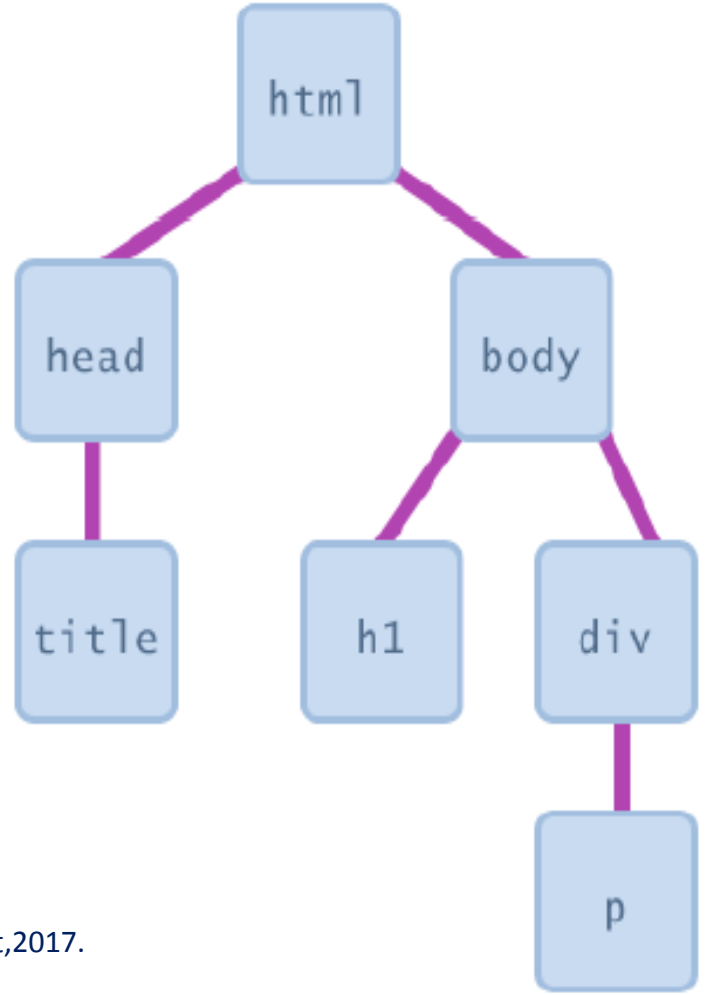
- JavaScript functions can be set as event handlers
  - when you interact with the element, the function will execute
- onclick is just one of many event HTML attributes we'll use
- but popping up an alert window is disruptive and annoying
  - A better user experience would be to have the message appear on the page...

```HTML
<element attributes onclick="function();">...
```

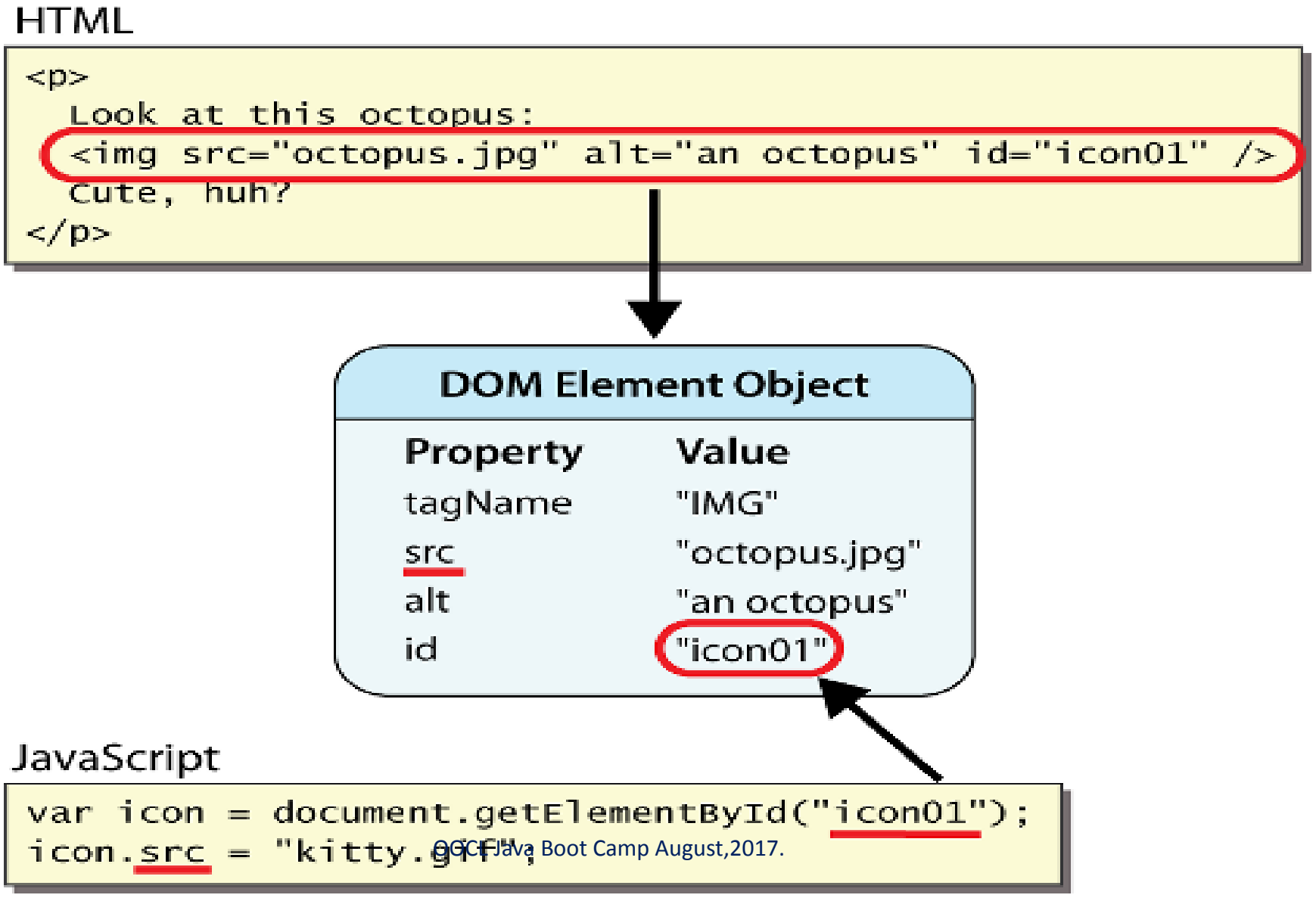
```HTML
<button onclick="myFunction();">Click me!</button>
```

DOCUMENT OBJECT MODEL (DOM)

- most JS code manipulates elements on an HTML page
- we can examine elements' state
  - e.g. see whether a box is checked
- we can change state
  - e.g. insert some new text into a div
- we can change styles
  - e.g. make a paragraph red



DOM ELEMENT OBJECTS



ACCESSING ELEMENTS:

document.getElementById

```
var name = document.getElementById("id");
```
JS

```
<button onclick="changeText();">Click me!</button>
<span id="output">replace me</span>
<input id="textbox" type="text" />
```
HTML

```
function changeText() {
        var span = document.getElementById("output");
        var textBox = document.getElementById("textbox");

         textbox.style.color = "red";

}
```
JS

Accessing Elements:
document.getElementById

- document.getElementById returns the DOM object for an element with a given id
- can change the text inside most elements by setting the innerHTML property
- can change the text in form controls by setting the value property

Changing Element Style: `element.style`

Attribute	Property or style object
color	color
padding	padding
background-color	backgroundColor
border-top-width	borderTopWidth
Font size	fontSize
Font famiy	fontFamily

PREETIFY

```
function changeText() {
	//grab or initialize text here

	// font styles added by JS:
	text.style.fontSize = "13pt";
	text.style.fontFamily = "Comic Sans MS";
	text.style.color = "red"; // or pink?
}
```
JS

DYNAMIC HTML (DHTML)

- Dynamic HTML can be viewed as the combination of HTML 4 ,CSS , and JavaScript
  - HTML 4 represents the static structure
  - CSS represents the appearance details
  - **JavaScript works on the dynamic behaviors of the content**!
    - Document Object Model (DOM) provides a programming interface between HTML/CSS and JavaScript
- DHTML isn't really about HTML
  - An abstract concept of breaking up a page into manipulable elements, and exposing them to script

WHY DHTML?

- Web evolves
  - static displays of data → interactive applications
- Allows a Web page to change after loaded into the browser
  - No need to communicate with server for an update
  - More efficient than Common Gateway Interface (CGI) solution in certain situations
    - Form validation

JavaScript Language

- The JavaScript Programming Language
  - Scripting Languages
    - Executed by an interpreter contained within the web browser (scripting host)
    - Interpreter uses a scripting engine
      - Converts code to executable format each time it runs
      - Converted when browser loads web document
  - Issues
    - Scripting vs. Programming
    - Interpreted vs. Compiled languages
    - Reference: http://www.classes.cs.uchicago.edu/classes/archive/2004/winter/10100-1/02/javascript/javascript01.html

JavaScript Language

- The JavaScript Programming Language
  - JavaScript
    - Originally called LiveScript when introduced in Netscape Navigator
    - In Navigator 2.0, name changed to JavaScript
    - Current version 1.5
  - JScript
    - MS version of JavaScript
      - Current version 5.5

JavaScript Language

- The JavaScript Programming Language
  - ECMAScript
    - International, standardized version (Edition 3)
    - Both versions (JavaScript and JScript) conform to the standard
      - Although both have proprietary extensions
    - Focus of this text

JavaScript Language

- The JavaScript Programming Language
  - JavaScript
    - Two formats:
      - Client-side
        - Program runs on client (browser)
      - Server-side
        - Program runs on server
        - Proprietary to web server platform

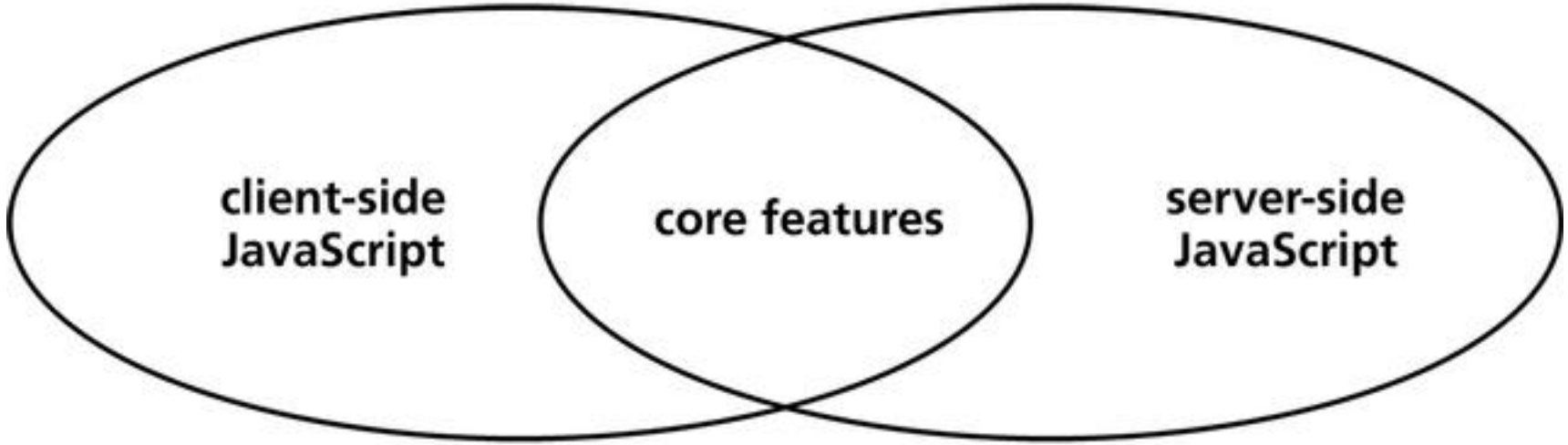


Figure 1-11: Relationship of client-side and server-side JavaScript

JavaScript Language

- JavaScript's role on the Web
  - JavaScript Programming Language
    - Developed by Netscape for use in Navigator Web Browsers
    - Purpose → make web pages (documents) more dynamic and interactive
      - Change contents of document, provide forms and controls, animation, control web browser window, etc.

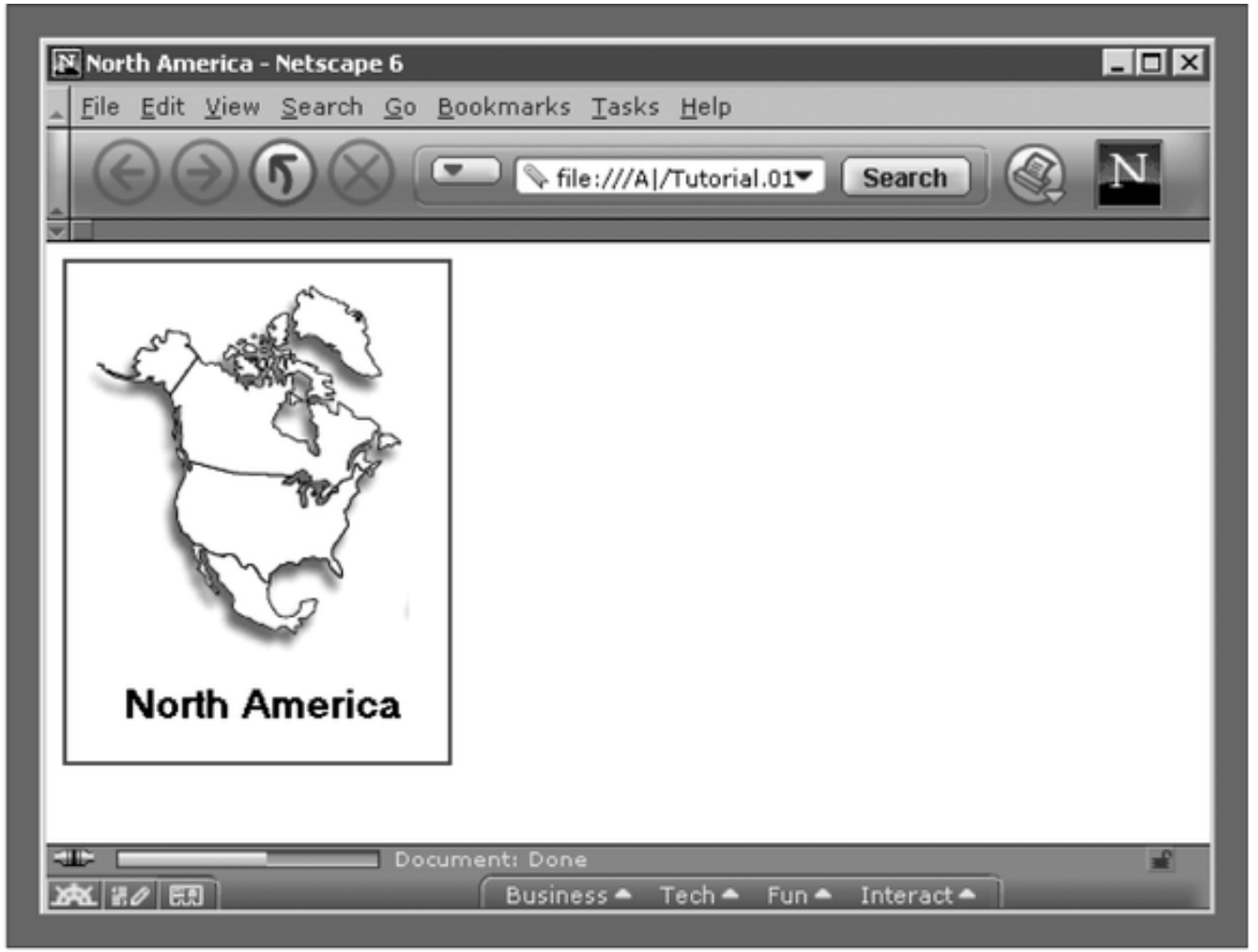


Figure 1-2: Image map

http://www.jsworkshop.com/js3e/list13-1.html

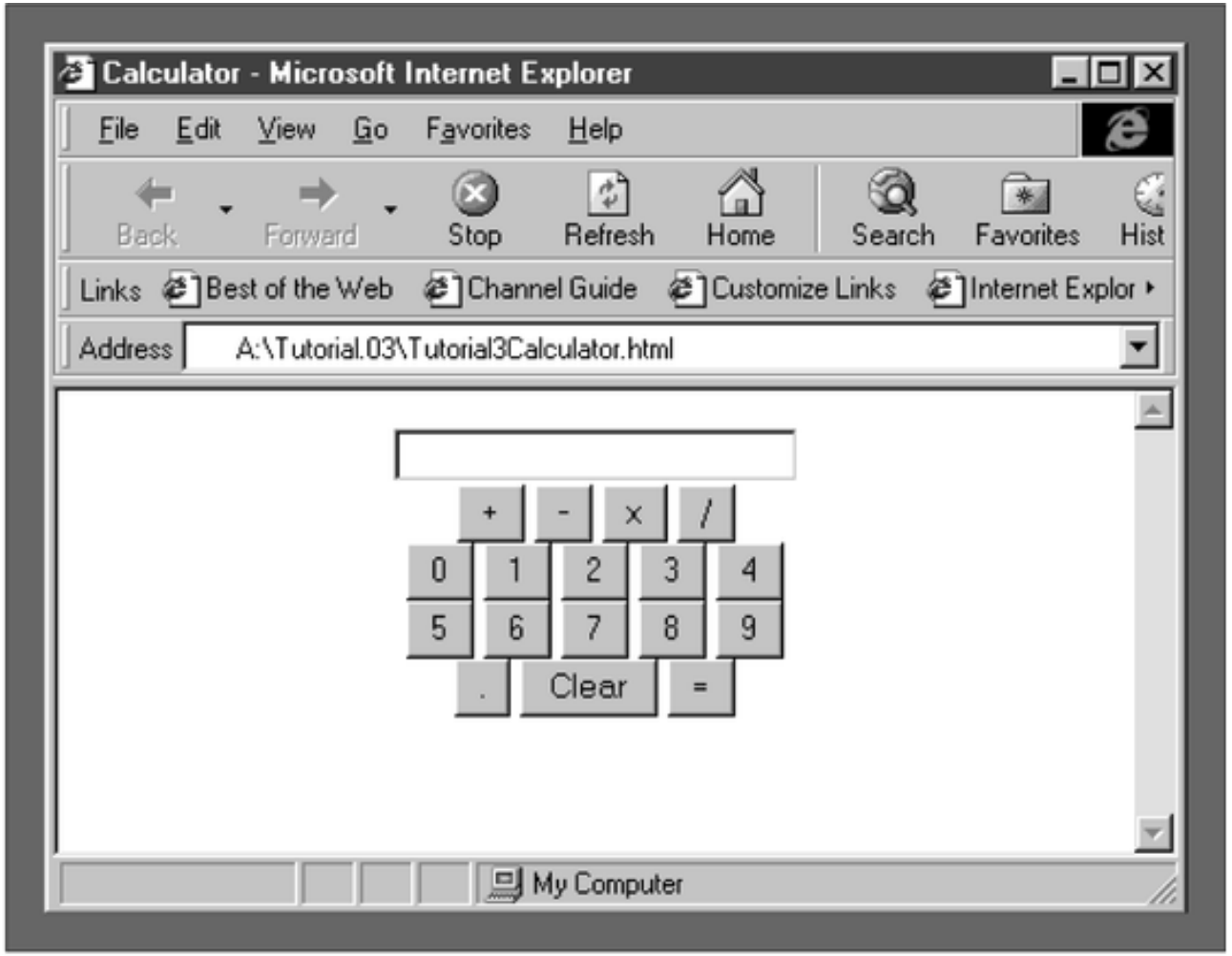


Figure 1-3: Online calculator

http://www.motionnet.com/calculator/

Figure 1-4: Product Registration form

http://javascript.about.com/library/scripts/blformvalidate.htm

A First JavaScript Program

- Section B – A First JavaScript Program
  - About the <script> tag
  - How to create a JavaScript source file
  - How to add comments to a JavaScript Program
  - How to hide JavaScript from incompatible browsers
  - About placing JavaScript in <head> or <body> sections of HTML documents

A FIRST JAVASCRIPT PROGRAM

- The <script> Tag
  - JavaScript programs are run from within an HTML document
  - <script> and </script>
    - Used to notify the browser that JavaScript statements are contained within

A FIRST JAVASCRIPT PROGRAM

- The <script> Tag
  - language attribute
    - Denotes the scripting language being used
      - Default → JavaScript
      - Other languages (e.g., VBScript) may be used
      - Can also specify script language version
        - No space between name and version
        - Checked by browser, scripts ignored if browser doesn't support version
      - For ECMAScript compatible browsers, omit version

Netscape Version	JavaScript Compatibility	Code
Navigator earlier than 2.0	not supported	—
Navigator 2.0	JavaScript 1.0	<SCRIPT LANGUAGE="JavaScript">…</SCRIPT>
Navigator 3.0	JavaScript 1.1 and lower	<SCRIPT LANGUAGE="JavaScript1.1">…</SCRIPT>
Navigator 4.0 – 4.05	JavaScript 1.2 and lower	<SCRIPT LANGUAGE="JavaScript1.2">…</SCRIPT>
Navigator 4.06 – 4.5	JavaScript 1.3 and lower	<SCRIPT LANGUAGE="JavaScript1.3">…</SCRIPT>

Figure 1-12: JavaScript versions supported in Navigator

http://devedge.netscape.com/library/manuals/2000/javascript/1.5/reference/preface.html#1003515

A FIRST JAVASCRIPT PROGRAM

- The <script> Tag
  - JavaScript
    - Object-based language *(not Object-Oriented language)*
      - Object
        - Programming code and data that can be treated as an individual unit or component
      - Statements
        - Individual lines in a programming language
      - Methods
        - Groups of statements related to a particular object

A FIRST JAVASCRIPT PROGRAM

- The <script> Tag
  - Document object
    - Represents the content of a browser's window
  - write() & writeln() methods of Document object
    - Creates new text on a web page
    - Called by appending method to object with a period
    - Methods accept arguments
      - Information passed to a method

A FIRST JAVASCRIPT PROGRAM

- The <script> Tag
  - Preformatted text
    - <pre> and </pre> tags
      - Tag set known as a container element because it contains text and other HTML tags
    - Translates literal text to presentation area
    - Required to get carriage return in writeln() method to be sent to presentation area

```
<PRE>

<SCRIPT LANGUAGE="JavaScript">

document.writeln("Hello World");

document.writeln(
"This line is printed below the 'Hello World' line.");

</SCRIPT>

</PRE>
```

Figure 1-13: Hello World script using the writeln() method of the Document object

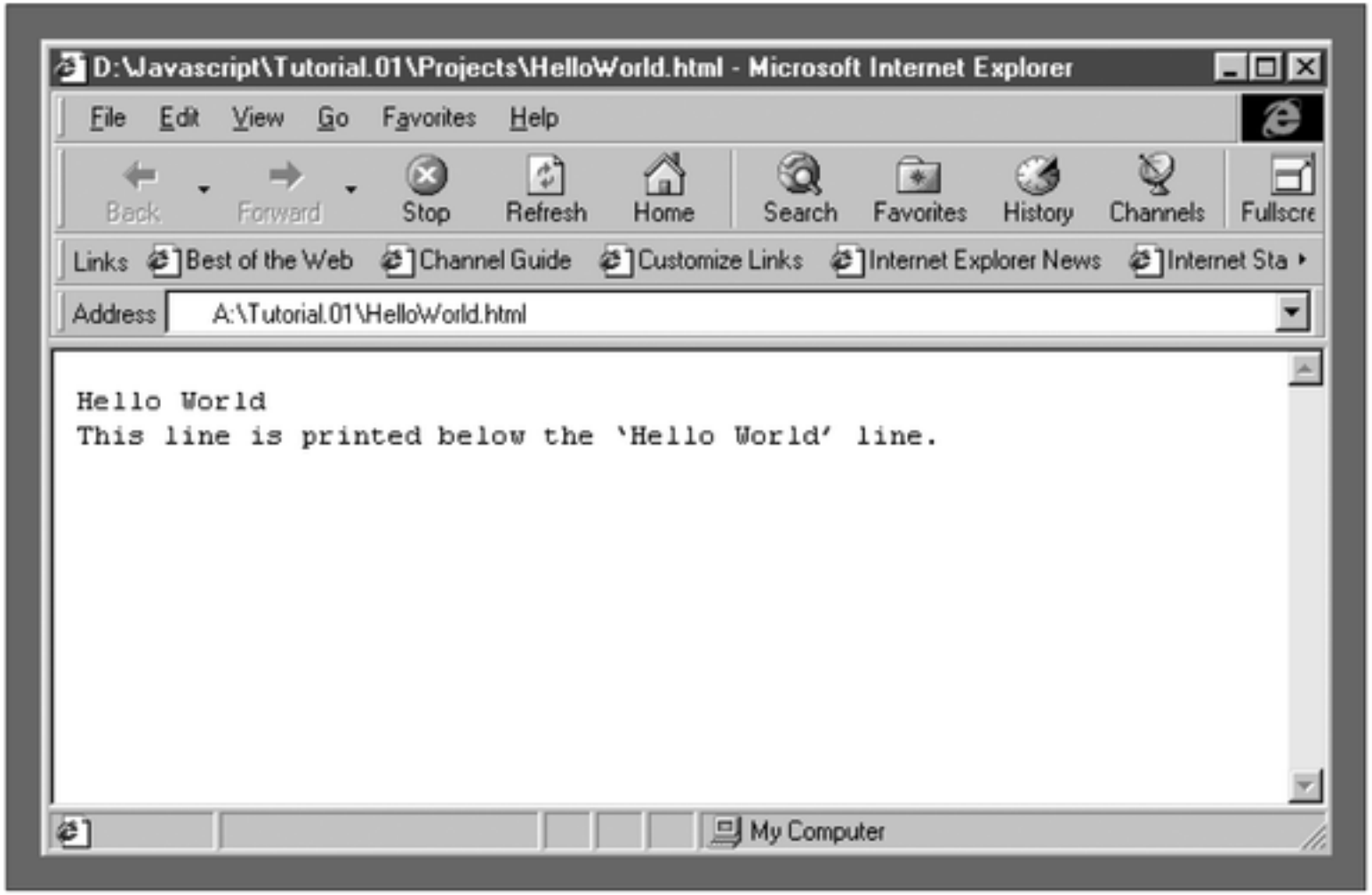


Figure 1-14: Output of the Hello World script using the writeln() method of the Document object

A FIRST JAVASCRIPT PROGRAM

- The <script> Tag
  - Document object
    - Considered a top-level object
    - Naming convention
      - Capitalize first letter of object
  - Unlike HTML, JavaScript is **case sensitive**

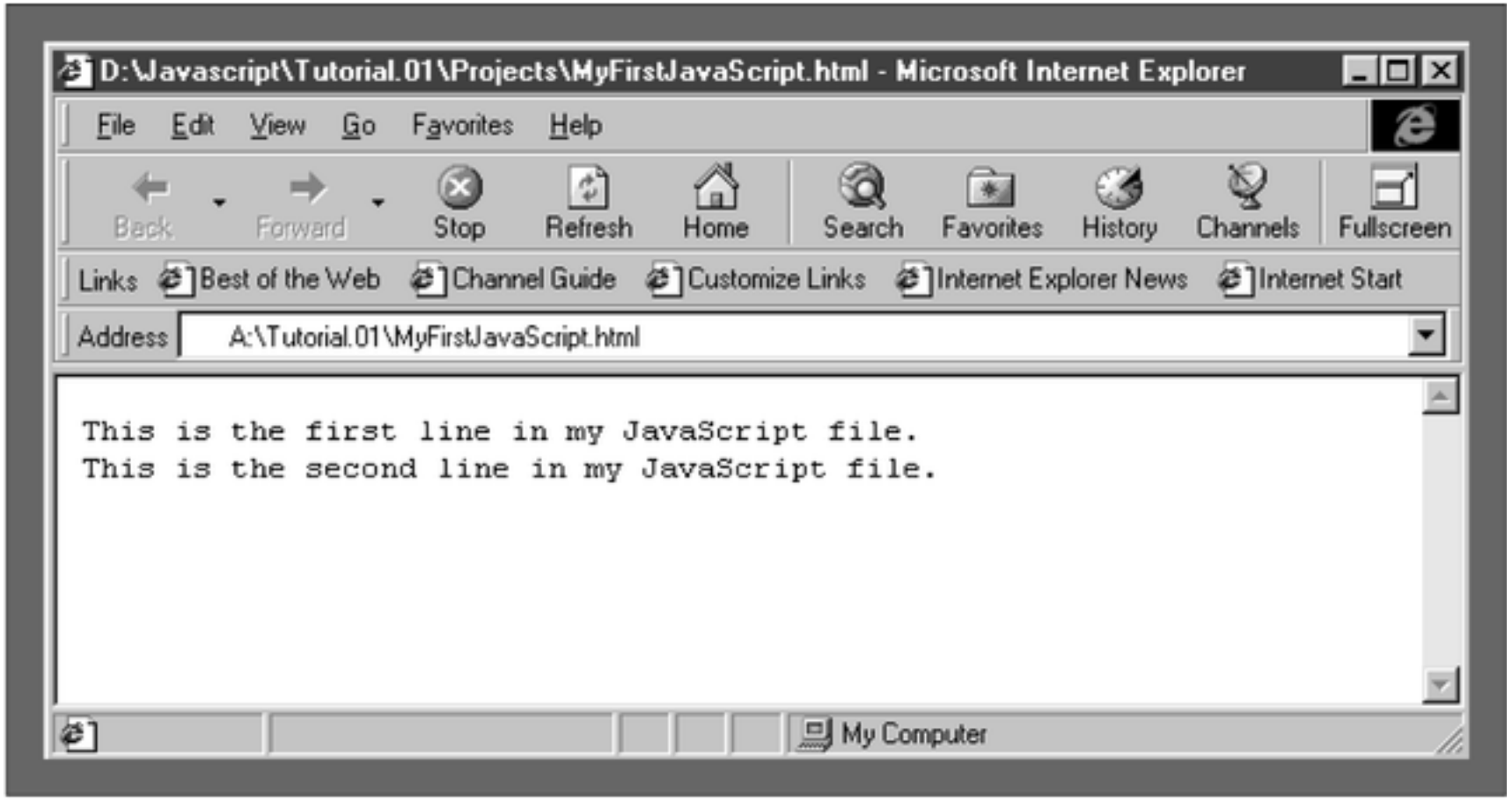


Figure 1-17: MyFirstJavaScript.html in Internet Explorer

http://old.jccc.net/~srao/JavaScript/Tutorial01/MyFirstJavaScript.html

A FIRST JAVASCRIPT PROGRAM

- Creating a JavaScript Source File
  - JavaScript programs can be used in two ways:
    - Incorporated directly into an HTML file
      - Using <script> tag
    - Placed in an external (source) file
      - Has file extension .js
      - Contains only JavaScript statements

A FIRST JAVASCRIPT PROGRAM

- Creating a JavaScript Source File
  - JavaScript source files
    - Use src attribute of <script> tag to denote source of JavaScript statements
      - Browser will ignore any JavaScript statements inside <script> and </script> if src attribute is used
    - **Cannot** include HTML tags in source file

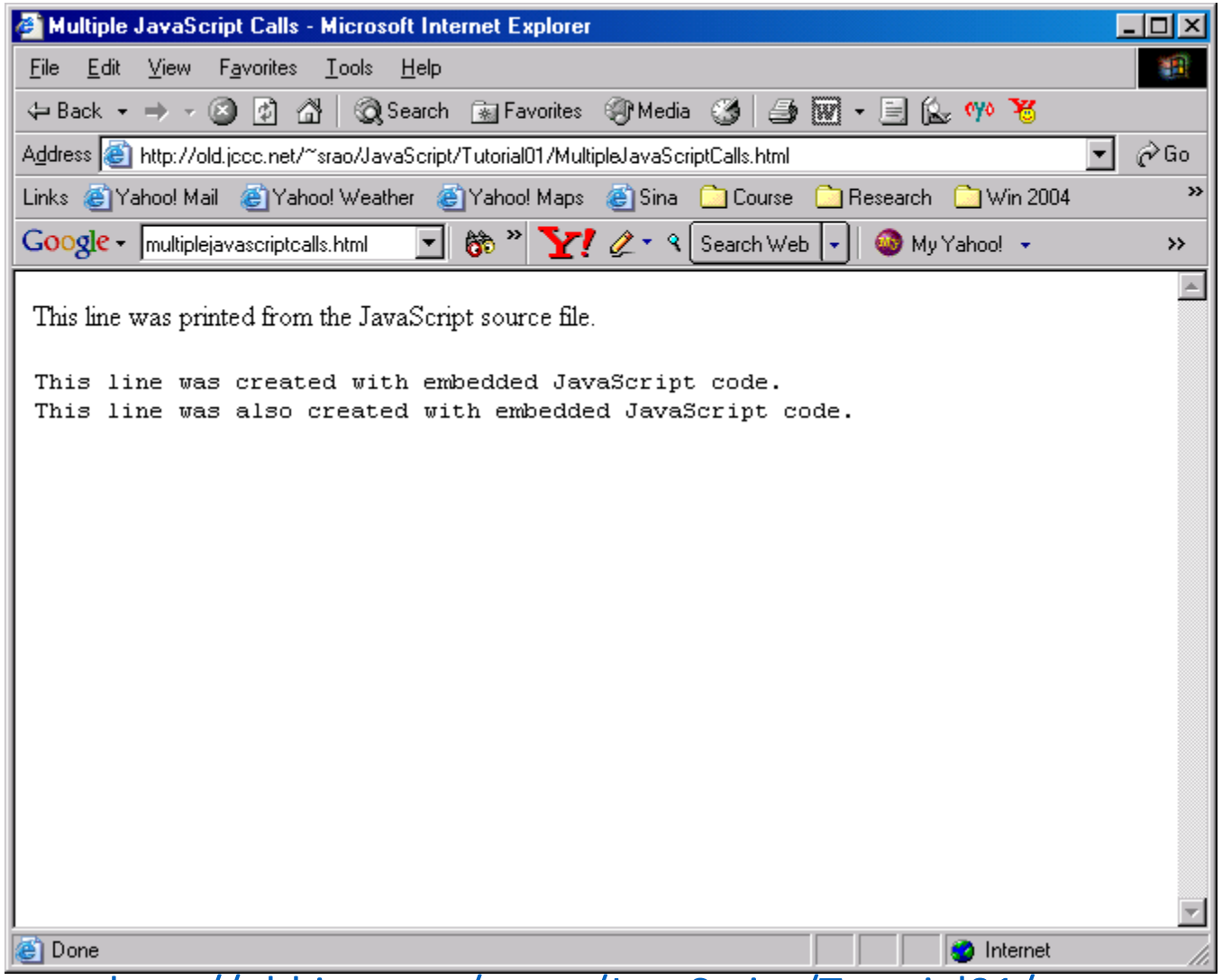
```
<script language="JavaScript" src="/home/hai/source.js">
</script>
```

A FIRST JAVASCRIPT PROGRAM

- Creating a JavaScript Source File
  - Advantages of JavaScript source files
    - Makes HTML document neater (less confusing)
    - JavaScript can be shared among multiple HTML files
    - Hides JavaScript code from incompatible browsers

A FIRST JAVASCRIPT PROGRAM

- Creating a JavaScript Source File
  - Can use a combination of embedded and non–embedded code
    - Allows finer granularity in coding functionality
    - JavaScript sections executed in order of location within HTML document



http://old.jccc.net/~srao/JavaScript/Tutorial01/MultipleJavaScriptCalls.html

A FIRST JAVASCRIPT PROGRAM

- Adding comments to a JavaScript Program
  - Comments
    - Non-printing lines that are placed in the code to contain various remarks about the code
      - Makes it easier to understand the code operation
    - Two types
      - Line comments
        - // ignore all text to the end of the line
      - Block comments
        - /* ignore all text between these symbols */

```
<SCRIPT LANGUAGE="JavaScript">
/*
This line is part of the block comment.
This line is also part of the block comment.
*/
document.writeln("Comments Example");  // Line comments can
follow code statements
// This line comment takes up an entire line.
/* This is another way of creating a block comment. */
</SCRIPT>
```

Figure 1-20: JavaScript file with line and block comments

A FIRST JAVASCRIPT PROGRAM

- Hiding JavaScript from Incompatible Browsers
  - Two methods
    - Place JavaScript in external source file
    - Enclose the code within HTML comments

```
<!-- beginning of HTML block comment
    end of HTML block comments -->
```

```
<SCRIPT LANGUAGE="JavaScript">
<!--  This line starts the HTML comment block
document.writeln("Your order has been confirmed.");
document.writeln("Thank you for your business.");
// This line ends the HTML comment block -->
</SCRIPT>
```

Figure 1-23: JavaScript code hidden from incompatible browsers, using HTML comments

A FIRST JAVASCRIPT PROGRAM

- Hiding JavaScript from Incompatible Browsers
  - Alternate message display
    - If browser doesn’t support JavaScript
      - Use <noscript> & </noscript> to place alternate message to users of back-level browsers

```
<SCRIPT LANGUAGE="JavaScript">

<!--  This line starts the HTML comment block

document.writeln("Your order has been confirmed.");

document.writeln("Thank you for your business.");

// This line ends the HTML comment block -->

</SCRIPT>

<NOSCRIPT>

Your browser does not support JavaScript or JavaScript is
disabled.<BR>

</NOSCRIPT>
```

Figure 1-24: JavaScript code with <NOSCRIPT> tag

A FIRST JAVASCRIPT PROGRAM

- Placing JavaScript in <head> or <body> sections
  - Script statements interpreted in order of document rendering
  - <head> section rendered before <body> section
    - Good practice to place as much code as possible in <head> section

Working with Variables, Functions, and Objects

- How to declare and use variables
- How to define and call functions
- About built-in JavaScript functions
- How to use JavaScript objects
- How to use object inheritance and prototypes
- How to use object methods
- About built-in JavaScript objects
- About variable scope

Working With Variables

- Variables (or identifiers)
  - Values stored in computer memory locations
  - Value can vary over time
  - Cannot use *reserved words* as variables
    - Reserved Words or Keywords are part of the JavaScript language syntax
  - Variable Example:
    - employeeName

abstract	char	delete	extends
boolean	class	do	false
break	const	double	final
byte	continue	else	finally
case	debugger	enum	float
catch	default	export	for

Figure 2-2: JavaScript reserved words

function	long	short	true
goto	native	static	try
if	new	super	typeof
implements	null	switch	var
import	package	synchronized	void
in	private	this	volatile
instanceof	protected	throw	while
int	public	throws	with
interface	return	transient	

Figure 2-2: JavaScript reserved words (continued)

Working With Variables

- Variables
  - To create:
    - Use keyword *var* to *declare* the variable
    - Use the assignment operator to assign the variable a value
  - Declare, then assign value (initialize)
    - var employeeName;
    - employeeName = “Ric”;
  - Declare and assign variable in a single statement
    - var employeeName = “Ric”;

Working With Variables

- Variables
  - Once created:
    - May be changed at any point in the program
  - Use variable name and assignment operator
    - employeeName = “Althea”;

Working With Variables

- Variables
  - Syntax rules
    - Cannot use:
      - Reserved words & spaces
    - Must begin with one of the following:
      - Uppercase or lowercase ASCII letter
      - Dollar sign ($) or underscore (_)
    - Can use numbers, but not as first character
    - Variables are case-sensitive

Working With Variables

- Variables
  - Conventions
    - Use underscore or capitalization to separate words of an identifier
      - employee_first_name
      - employeeFirstName

```
my_variable

$my_variable

_my_variable

my_variable_example

myVariableExample
```

Figure 2-3: Examples of legal variable names

```
%my_variable

1my_variable

#my_variable

@my_variable

~my_variable

+my_variable
```

Figure 2-4: Examples of illegal variable names

Working With Variables

- Can write the value contained in a variable to a web page:

```
var myName = “john”;
document.writeln(“Hello “);
document.writeln(myName);
```

Working With Variables

- Can use the + concatenation operator:

```
var myName = “john”;
document.writeln(“Hello “ + myName);
```

Working With Functions

- Defining Custom Functions
  - Function:
    - Individual statements grouped together to form a specific procedure
    - Allows you to treat the group of statements as a single unit
    - Must be contained between <script> and </script> tags
    - Must be formally composed (function definition)

Working With Functions

- Defining Custom Functions
  - A function definition consists of three parts:
    - Reserved word *function* followed by the function name (identifier)
    - Parameters required by the function, contained within parentheses following the name
      - Parameters → variables used within the function
      - Zero or more may be used
    - Function statements, delimited by curly braces { }

```
function print_company_name(company1, company2, company3) {
     document.writeln(company1);
     document.writeln(company2);
     document.writeln(company3);
}
```

Figure 2-5: Function that prints the name of multiple companies

Working With Functions

- Calling Functions
  - Function invocation or call
    - Statement including function name followed by a list of *arguments* in parentheses
    - Parameter of function definition takes on value of argument *passed* to function in function call

Working With Functions

- Calling Functions
  - Code placement
    - Functions must be created (defined) before called
      - <head> rendered by browser before <body>
    - Function definition
      - Place in <head> section
    - Function call
      - Place in <body> section

Working With Functions

- Variable Scope
  - Defines where in the program a declared variable can be used
    - Global variable
      - Declared outside a function and is available to all parts of the program
      - var keyword optional
    - Local variable
      - Declared inside a function and is only available within the function it is declared
    - Global and local variables can use same identifier

Working With Functions

```
<head>
<script language="JavaScript">
function putStuff(myName, myNum){
  document.writeln("Hello " + myName + ", how are you?");
  var rslt = myNum * 2;
  document.writeln (myNum + " * 2 = ", rslt);
}
</script>
</head>
<body>
<script>
putStuff("John", 5);
</script>
</body>
```

Working With Functions

- Returning a value from a Function
  - A function can return nothing
    - Just performing some task
  - A function can return a value
    - Perform a calculation and return the result
      - Var returnValue = functionCall(opOne, opTwo);
    - A *return* statement must be added to function definition

EXAMPLES

- TwoFunctionsProgram.html
- CompanyName.html
- SingleCompanyName.html

Working With Functions

- Built-in JavaScript Functions
  - Functions defined as part of the JavaScript language
  - Function call identical to the custom functions

Built-in JavaScript

```
alert("some string");
var rslt = prompt("question", "default");
var rslt = confirm("some string");
```

alert displays an alert box with the message.

prompt displays the "question" and allows the user to input a response. The "default" value shows up in the response area of the prompt box.

Returns the value the user enters.

confirm displays the string in a box and then has "ok" and "cancel" buttons.

Returns TRUE if the user hits the "ok" button and returns **FALSE** if the user hits the "cancel" button.

Built-in JavaScript

- To create multiple line messages: use the ‘\n’ character.

- No html formatting (or other types of formatting) can be done on the message that is displayed in these boxes.

- **prompt** always returns a string, even if the user enters a number into the response area of the box.
  - **If you want to use the response as a number you must use the** parseInt **method to change the string into a number.**

BUILT-IN JAVASCRIPT

```
<head>
        <title>Prompt Box</title>

        <script type = "text/javascript">
          <!--
        function putWelcome()
        {
          var myName=prompt("What's your name?");
          confirm("Is your name really " + myName + "?");
            document.writeln(“Hello “ + myName);
            alert(“Hello” + myName);
        }
          // -->
        </script>
    </head>
    <body>
    <script>
        putWelcome( );
    </script>
    </body>
```

http://www.ithaca.edu/barr/Student/CS205/Examples/Tutorial02/prompt2.html

Built-in JavaScript

- There are other built-in functions that extend the capabilities of javascript.

Function	Description
eval()	Evaluates expressions contained within strings
isFinite()	Determines whether a number is finite
isNaN()	Determines whether a value is the special value NaN (Not a Number)
parseInt()	Converts string literals to integers
parseFloat()	Converts string literals to floating-point numbers
encodeURI()	Encodes a text string into a valid URI
encodeURIComponent()	Encodes a text string into a valid URI component
decodeURI()	Decodes text strings encoded with encodeURI()
decodeURIComponent()	Decodes text strings encoded with encodeURIComponent()

Figure 2-9: **Built-in JavaScript functions**

Working With

- Objects are similar to built-in programs that contain their own functions and variables.
- You access a function or variable that belongs to an object using the dot syntax:

```
document.writeln(``my message'');
```

- There are built-in objects that relate to the browser and its contents:
  - Document
  - Window
  - Navigator
  - Screen
  - Element
  - Event
  - Form

Working With

- There are also built-in JavaScript objects that extend the functionality of the language:
    - JavaScript includes 11 built-in objects that relate to the language itself
    - Each object contains various methods and properties for performing a particular task
    - Can be used directly in program without instantiating a new object

Object	Description
Array	Creates new array objects
Boolean	Creates new Boolean objects
Date	Retrieves and manipulates dates and times
Error	Returns run-time error information
Function	Creates new function objects
Global	Represents the JavaScript built-in methods
Math	Contains methods and properties for performing mathematical calculations
Number	Contains methods and properties for manipulating numbers
Object	Provides common functionality to all built-in JavaScript objects
RegExp	Contains properties for finding and replacing in text strings
String	Contains methods and properties for manipulating text strings

Figure 2-10: Built-in JavaScript objects

Working With

- Understanding JavaScript Objects
  - In OO languages (Java, C++)
    - Class structures contain associated variables, methods (functions) and statements
    - Objects are instances of classes
      - (i.e., objects are instantiated from classes)
    - Classes can be inherited from other classes
  - JavaScript is not truly object-oriented
    - Cannot create classes

(NOT REQUIRED SINCE THIS SLIDE)

- Understanding JavaScript Objects
  - Custom JavaScript objects
    - Based on constructor functions
      - Instantiate a new object or extending an old object
      - Objects inherit all variables and statements of constructor function
    - Any JavaScript function can serve as a constructor

- Understanding JavaScript Objects
  - Constructor function
    - Has two types of elements
      - Property (field)
        - Variable within a constructor function
        - Data of the object
      - Method
        - Function called from within an object
        - Can be custom or built-in function

- Understanding JavaScript Objects
  - Constructor function
    - Identifier naming convention
      - First word uppercase to differentiate from non-constructor functions
        - BankEmployee
    - The *this* keyword
      - Used in constructor functions to refer to the current object that called the constructor function

- Understanding JavaScript Objects
  - Constructor function
    - The *new* keyword
      - Used to create new objects from constructor functions
      - Example:
        - Achmed = new BankEmployee(name, empNum);

- Understanding JavaScript Objects
  - Custom object inheritance and prototypes
    - Objects inherit the properties and methods of their constructor function
    - Constructor functions:
      - Do not require parameters
      - Do not require passed arguments
        - Properties may set the value at a later time
        - If used before set, will return an *undefined* value

- Understanding JavaScript Objects
  - Custom object inheritance and prototypes
    - Adding properties after object is instantiated
      - Properties can be added to objects using dot operator (.)
      - These properties available only to that specific object
    - Prototype properties
      - Properties added using *prototype* keyword
      - Available to all objects that extend the constructor function
        - BankEmployee.prototype.department = “”;

- Understanding JavaScript Objects
  - Custom object inheritance and prototypes
    - Object definitions can extend other object definitions
      - Extend original definition and add new properties or function calls

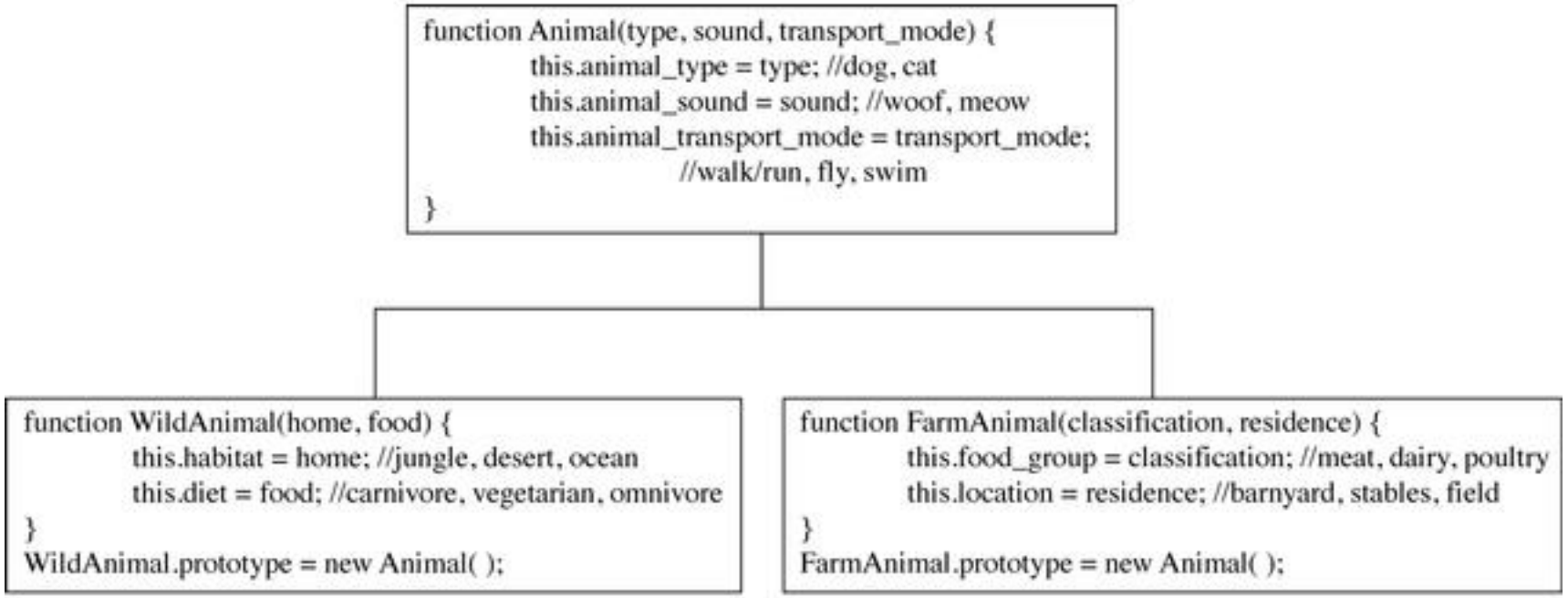


Figure 2-11: Two object definitions extending another object definition

- Understanding JavaScript Objects
  - Custom object methods
    - Functions associated with a particular object
      - Define the function
      - Use the *this* reference to refer to object properties
      - Add it to the constructor function
        - this.methodName = functionName
    - Call the method using the dot operator (.)

EXERCISE 1