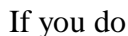


# INTRODUCTION TO THE DOCUMENT OBJECT MODEL

## Nodes

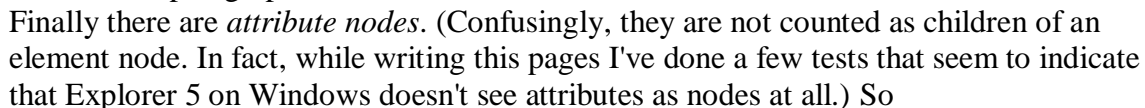
<P>This is a paragraph</P>

<P> <-- element node



<P>This is a <B>paragraph</B></P>

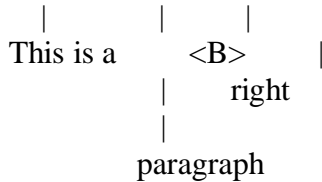
$\langle P \rangle$



**This is a paragraph**

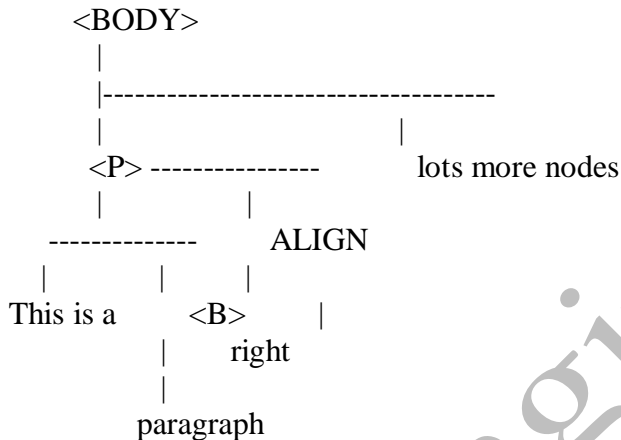
would give something like





So these are element nodes, text nodes and attribute nodes. They constitute about 99% of the content of an HTML page and you'll usually busy yourself with manipulating them. There are more kinds of nodes, but I skip them for the moment.

As you'll understand, the element node P also has its own parent, this is usually the document, sometimes another element like a DIV. So the whole HTML document can be seen as a tree consisting of a lot of nodes, most of them having child nodes (and these, too, can have children).



### Walking through the DOM tree

Knowing the exact structure of the DOM tree, you can walk through it in search of the element you want to influence. For instance, assume the element node P has been stored in the variable x (later on I'll explain how you do this). Then if we want to access the BODY we do

`x.parentNode`

We take the parent node of x and do something with it. To reach the B node:

`x.childNodes[1]`

`childNodes` is an array that contains all children of the node x. Of course numbering starts at zero, so `childNodes[0]` is the text node 'This is a' and `childNodes[1]` is the element node B.

Two special cases: `x.firstChild` accesses the first child of x (the text node), while

`x.lastChild` accesses the last child of x (the element node B).

So supposing the P is the first child of the body, which in turn is the first child of the document, you can reach the element node B by either of these commands:

`document.firstChild.firstChild.lastChild;`

`document.firstChild.childNodes[0].lastChild;`

`document.firstChild.childNodes[0].childNodes[1];`

etc.

or even (though it's a bit silly)

`document.firstChild.childNodes[0].parentNode.firstChild.childNodes[1];`

## **DOM Levels and history**

A short history lesson in order. When JavaScript was first introduced in browsers, some sort of interface was required to allow elements on the page to be accessed via scripting. Each vendor had their own implementation but de facto standards emerged to produce a fairly simple model common to most. For example, most browsers used an array of Image objects to represent all IMG tags on a page. These could then be accessed and manipulated via JavaScript. A simple image rollover might use code like this:

```
document.images[3].src = "graphics/button2.gif"
```

These early models were limited. They only provided access to a few types of element and attributes, like images, links or form elements. As vendors released new versions of browsers, they expanded on the model. But they also were often at odds with one another, leading to compatibility problems among different browsers as vendors tried to outdo each other by adding their own new features. Fortunately, most vendors started adopting the DOM standard set by the World Wide Web Consortium, notably Internet Explorer, Netscape and Opera.

In order to provide some backward compatibility, different levels of the DOM standard are defined. You might find references to DOM Level 0 (or "DOM0") which corresponds to the model used by the first, scriptable browsers - mainly Internet Explorer and Netscape prior to version 4 of each. Then there is DOM1 which was adopted in 1998 and covers some of the features introduced in version 4 browsers. Most of the current browsers (version 5 and up) support the DOM2 standard, or at least a good part of it. They may also continue to support some features of the earlier DOM levels, or their own proprietary extensions, so that they are compatible with older web pages.

This article will focus just the current, DOM2 standard. While this standard applies to XML (extended markup language) documents in general, it discusses how the standard applies to HTML in particular (it being a subset of XML). The good news is that, given the current industry trend, you can expect future versions of browsers to follow this standard. The bad news is that for now, you may find it difficult to code pages that work with both old and new browsers.

One such example is Netscape 6.0, which drops support for many of the proprietary features of Netscape 4, such as the LAYER tag and its corresponding Layer object. They are not recognized in version 6 as they were never adopted as part of the standard.

Also note that Internet Explorer's document.all construct is a proprietary feature, not part of any standard. While it may be supported in many versions of IE - even the latest version - it's generally not supported by other browsers. You should keep in mind that the DOM coding is also indirectly dependent on the standards for HTML and CSS, since the DOM reflects the tags and attributes defined by those standards. It also depends on a standard for JavaScript since the DOM is essentially an API for client-side scripting.

## **INTRINSIC EVENT HANDLING**

Event Handling has been part of JavaScript since the language's inception. As described in our event handler tutorial, they refer to specific, user imitated actions within the webpage, such as the moving of your mouse over a link, the clicking on a link, or submission of a form. Thanks to event handling, our scripts are more interactive and are able to perform certain actions depending on the user's. The DOM of modern web browsers such as IE5+, NS6+, and Firefox provide expanded methods and flexibility

(relative to older browsers) for capturing events. In this tutorial, we explore event handling in the DOM, and the differing support for it in IE5+ and NS6+/Firefox.

## The 2 traditional ways of assigning event handlers

Let's first review (for most of us, at least) the 2 common and conventional ways of setting up an event handler- via HTML, or scripting. In both cases, a function or code is attached at the end, which is executed when the handler detects the specified event.

### 1) Via HTML, using attributes

We can define an event handler directly inside the relevant HTML tag, by embedding it as a attribute. A piece of JavaScript is also included to tell the browser to perform something when the event occurs. For example,

```
<a href="http://freewarejava.com" onMouseover="window.status='Click here for Java applets';return true" onMouseout="window.status=''">Freewarejava.com</a>
```

Here the event handler (onMouseover) is directly added inside the desired element (A), along with the JavaScript to execute.

### 2) Via scripting

You can also assign and set up event handlers to elements using scripting, and inside your script . This allows for the event handlers to be dynamically set up, without having to mess around with the HTML codes on the page.

When setting up event handlers for an element directly inside your script, the code to execute for the events must be defined inside a function. Just look at the below, which does the same thing as above, but with the event handler defined using scripting:

```
<a ID="test" href="http://freewarejava.com">Freewarejava.com</a>
```

```
<script type="text/javascript">
```

```
function changestatus(){  
window.status="Click here for Java applets"  
return true  
}
```

```
function changebackstatus(){  
window.status=""  
}
```

```
document.getElementById("test").onmouseover=changestatus  
document.getElementById("test").onmouseout=changebackstatus
```

```
</script>
```

Notice how we attached the two functions to execute for the two events- the function names without the **parenthesis**. This is called a reference call to the function. When assigning a function to an event via scripting, always use a function call. If you were to include the parenthesis of the function inside the definition, an error will be generated.

## MODIFYING ELEMENT STYLE

HTML elements can be changed using JavaScript, the HTML DOM and events.

### Change an HTML Element

HTML DOM and JavaScript can change the inner content and attributes of HTML elements.

The following example changes the background color of the <body> element:

#### Example

```
<html>
<body>

<script type="text/javascript">
document.body.bgColor="lavender";
</script>

</body>
</html>
```

### Change the Text of an HTML Element - innerHTML

The easiest way to get or modify the content of an element is by using the innerHTML property.

The following example changes the text of a <p> element:

#### Example

```
<html>
<body>

<p id="p1">Hello World!</p>

<script type="text/javascript">
document.getElementById("p1").innerHTML="New text!";
</script>

</body>
</html>
```

### Change an HTML Element Using Events

An event handler allows you to execute code when an event occurs. Events are generated by the browser when the user clicks an element, when the page loads, when a form is submitted, etc. The following example changes the background color of the <body> element when a button is clicked:

#### Example

```
<html>
<body>

<input type="button" onclick="document.body.bgColor='lavender';"
value="Change background color" />

</body>
</html>
```

## Change the Text of an Element - with a Function

The following example uses a function to change the text of the <p> element when a button is clicked:

### Example

```
<html>
<head>
<script type="text/javascript">
function ChangeText()
{
document.getElementById("p1").innerHTML="New text!";
}
</script>
</head>

<body>
<p id="p1">Hello world!</p>
<input type="button" onclick="ChangeText()" value="Change text" />
</body>
</html>
```

## Using the Style Object

The Style object of each HTML element represents its individual style.

The following example uses a function to change the style of the <body> element when a button is clicked:

### Example

```
<html>
<head>
<script type="text/javascript">
function ChangeBackground()
{
document.body.style.backgroundColor="lavender";
}
</script>
</head>

<body>
<input type="button" onclick="ChangeBackground()"
value="Change background color" />
</body>
</html>
```

## Change the font and color of an Element

The following example uses a function to change the style of the <p> element when a button is clicked:

### Example

```
<html>
<head>
<script type="text/javascript">
function ChangeStyle()
{
document.getElementById("p1").style.color="blue";
document.getElementById("p1").style.fontFamily="Arial";
}
</script>
</head>

<body>
<p id="p1">Hello world!</p>
<input type="button" onclick="ChangeStyle()" value="Change style" />
```

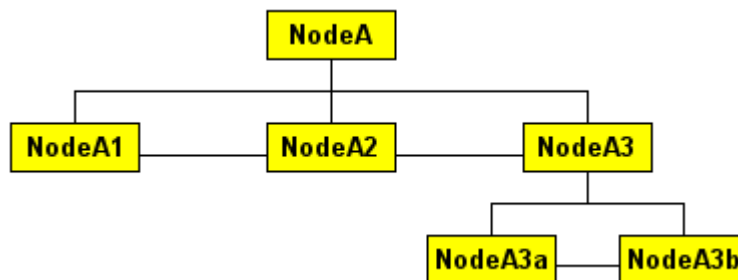
```
</body>
</html>
```

## THE DOCUMENT TREE

When a browser loads a page, it creates a hierarchical representation of its contents which closely resembles its HTML structure. This results in a tree-like organization of nodes, each representing an element, an attribute, content or some other object.

### Nodes

Each of these different object types will have their own unique methods and properties. But each also implements the Node interface. This is a common set of methods and properties related to the document tree structure. To understand this interface better, take a look at the diagram below which represents a simple node tree.



The Node object provides several properties to reflect this structure and allow you to traverse the tree. Here are some relationships using the example above:

- NodeA.firstChild = NodeA1
- NodeA.lastChild = NodeA3
- NodeA.childNodes.length = 3
- NodeA.childNodes[0] = NodeA1
- NodeA.childNodes[1] = NodeA2
- NodeA.childNodes[2] = NodeA3
- NodeA1.parentNode = NodeA
- NodeA1.nextSibling = NodeA2
- NodeA3.prevSibling = NodeA2
- NodeA3.nextSibling = null
- NodeA.lastChild.firstChild = NodeA3a
- NodeA3b.parentNode.parentNode = NodeA

The Node interface also provides methods for dynamically adding, updating and removing nodes such as:

- insertBefore()
- replaceChild()
- removeChild()
- appendChild()
- cloneNode()

These will be covered later. But for now let's look how the document tree reflects the contents of a web page.

## The Document Root

The document object serves as the root of this node tree. It too implements the Node interface. It will have child nodes but no parent node or sibling nodes, as it is the starting node. In addition to being a Node, it also implements the Document interface.

This interface provides methods for accessing and creating other nodes in the document tree. Some methods are:

- getElementById()
- getElementsByTagName()
- createElement()
- createAttribute()
- createTextNode()

Note that unlike other nodes, there is only one document object in a page. All of the above methods (except getElementsByTagName()) can only be used against the document object, i.e., using the syntax `document.methodName()`.

The document object can also have several other properties related to older DOM level support. For example, you'll find that many browsers still have `document.images` and `document.links` arrays or `document.bgColor` and `document.fgColor` properties relating to the BGCOLOR and TEXT attributes of the BODY tag.


These properties are intended to provide some backward compatibility so that pages designed for older browsers will still work properly with newer versions. They can still be used in scripts but they may not be supported in future versions.

## Traversing the Document Tree

As mentioned, the document tree reflects the structure of a page's HTML code. Every tag or tag pair is represented by a element node with other nodes representing attributes or character data (i.e., text).

Technically speaking, the document object has only one child element, given by `document.documentElement`. For web pages, this represents the outer HTML tag and it acts as the root element of the document tree. It will have child elements for HEAD and BODY tags which in turn will have other child elements.

Using this, and the methods of the Node interface, you can traverse the document tree to access individual node within the tree. Consider the following:



```
<html>
<head>
<title></title>
</head>
<body><p>This is a sample paragraph.</p></body>
</html>
```

and this code:

```
alert(document.documentElement.lastChild.firstChild.tagName);
```

which would display "P", the name of the tag represented by that node. The code breaks down as follows:

- `document.documentElement` - gives the page's HTML tag.
- `.lastChild` - gives the BODY tag.
- `.firstChild` - gives the first element in the BODY.
- `.tagName` - gives that element's tag name, "P" in this case.



There are some obvious problems with accessing nodes like this. For one, a simple change to the page source, like adding text or formatting elements or images, will change the tree structure. The path used before may no longer point to the intended node.

Less obvious are some browser compatibility issues. Note that in the sample HTML above, there is no spacing between the BODY tag and the P tag. If some simple line breaks are added,

```
<html>
<head>
<title></title>
</head>
<body>

<p>This is a sample paragraph.</p>

</body>
</html>
```

Netscape will add a node for this data, while IE will not. So in Netscape, the JavaScript code shown above would display "undefined" as it now points to the text node for this white space. Since it's not an element node, it has no tag name. IE, on the other hand, does not add nodes for white space like this, so it would still point to the P tag.

### Accessing Elements Directly

This is where the `document.getElementById()` method comes in handy. By adding an ID attribute to the paragraph tag (or any tag for that matter), you can reference it directly.

```
<p id="myParagraph">This is a sample paragraph.</p>
```

...

```
alert(document.getElementById("myParagraph").tagName);
```

This way, you can avoid compatibility issues and update the page contents at will without worrying about where the node for the paragraph tag is in the document tree. Just remember that each ID needs to be unique to the page.

A less direct method to access element nodes is provided by `document.getElementsByTagName()`. This returns an array of nodes representing all of the elements on a page with the specified HTML tag. For example, you could change color of every link on a page with the following.

```
var nodeList = document.getElementsByTagName("A");
for (var i = 0; i < nodeList.length; i++)
    nodeList[i].style.color = "#ff0000";
```

Which simply updates each link's inline style to set the color parameter to red. Give it a try.

### Node Types

Before going further, it's probably a good time to explain node types in more detail. As mentioned, there are several types of nodes defined in the document object model, but the ones you'll mostly deal with for web pages are *element*, *text* and *attribute*.

*Element* nodes, as we've seen, correspond to individual tags or tag pairs in the HTML code. They can have child nodes, which may be other elements or text nodes.

*Text* nodes represent content, or character data. They will have a parent node and possibly sibling nodes, but they cannot have child nodes.

*Attribute* nodes are a special case. They are not considered a part of the document tree - they do not have a parent, children or siblings. Instead, they are used to allow access to an element node's attributes. That is, they represent the attributes defined in an element's HTML tag, such as the HREF attribute of the A tag or the SRC attribute on the IMG tag. Note that attribute values are always text strings.

### **Attributes vs. Attribute Nodes**

There are several ways to reference the attributes of an element. The reason for this is that the DOM2 standard was designed for many types of structured documents (i.e., XML documents), not just HTML. So it defines a formal node type for attributes.

But for all documents it also provides some more convenient methods for accessing, adding and modifying attributes, as described next.

The `document.createAttribute()` allows you to create a new attribute node, which you can then set a value for and assign to an element node.

```
var attr = document.createAttribute("myAttribute");
attr.value = "myValue";
var el = document.getElementById("myParagraph");
el.setAttributeNode(attr);
```

However, it's usually easier to access an element's attributes directly using the element `getAttribute()` and `setAttribute()` methods instead.

```
var el = document.getElementById("myParagraph");
el.setAttribute("myAttribute", "myValue");
```

An element's attributes are also represented as properties of the element node. In other words, you can simply use

```
var el = document.getElementById("myParagraph");
el.myAttribute = "myValue";
```

It's also interesting to note that you can define your own attributes in the HTML tag itself. For example,

```
<p id="myParagraph" myAttribute="myValue">This is a sample paragraph.</p>
```

...

```
alert(document.getElementById("myParagraph").getAttribute("myAttribute"));
```

will display "myAttribute." But note that you should use

`element.getAttribute(attributeName)` instead of `element.attributeName` to get the value as some browsers may not register user-defined attributes as a properties of the element.

Attributes can also be removed from an element node, using either the `removeAttribute()` or `removeAttributeNode()` methods or setting by setting `element.attributeName` to a null string ("").

Altering attributes is one way to create dynamic effects. Below is a sample paragraph.

Use the links to alter it's ALIGN attribute.

Text in a paragraph element.

Align Left | Align Right

The code is fairly simple:

```
<p id="sample1" align="left">Text in a paragraph element.</p>
```

... code for the links ...

```
document.getElementById('sample1').setAttribute('align', 'left');  
document.getElementById('sample1').setAttribute('align', 'right');
```

### **Style Attributes**

Most attributes for HTML tags are fairly simple, they define a single value for a property specific to that tag. Styles are a little more involved. As you know, CSS can be used to apply style parameters to an individual tag, all tags of a given type or assigned using *classes*. Likewise, styles for a particular element can be inherited from any of these sources.

You can also alter these styles at various levels. For example, you can change the STYLE attribute of an HTML tag, or it's CLASS attribute. But these methods will alter all of the element's style parameters. Often, you may want to change just a single style parameter, or a select few, while retaining the others.

Fortunately, the style attribute of an element node is defined as an object with properties for every possible style parameter. You can access and update these individual parameters as you wish. Here's an example similar to the previous one.

Text in a paragraph element.

[Align Left](#) | [Align Right](#)

But in this case, the text alignment is defined and altered using a style parameter instead of the ALIGN attribute. Here's the code behind it:

```
<p id="sample2" style="text-align:left;">Text in a paragraph  
element.</p>
```

... code for the links ...

```
document.getElementById('sample2').style.textAlign = 'left';  
document.getElementById('sample2').style.textAlign = 'right';
```

### **Dynamic Content**

Changing textual content is relatively simple. Every continuous string of character data in the body of an HTML page is represented by a text node. The nodeValue property of these nodes is the text itself. Changing that value will change the text on the page.

### **Text Nodes**

Here's another example using a simple paragraph tag. Use the links to change the text:

This is the initial text.

[Change Text 1](#) | [Change Text 2](#)

Now look at the code behind it:

```
<p id="sample1">This is the initial text.</p>
```

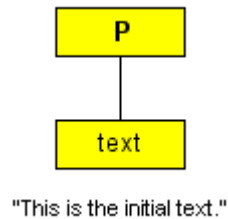
... code for the links ...

```
document.getElementById('sample1').firstChild.nodeValue =  
'Once upon a time...';
```

```
document.getElementById('sample1').firstChild.nodeValue =  
'...in a galaxy far, far away';
```

There are a couple of important things to note here. First, text nodes do not have an ID attribute like element nodes can. So they cannot be accessed directly using methods like `document.getElementById()` or `document.getElementsByTagName()`.

Instead, the code references the text using the parent node, in this case it's the paragraph element with the ID "sample1". This element node has one child node, the text node we want to update. You can see this in the diagram below.

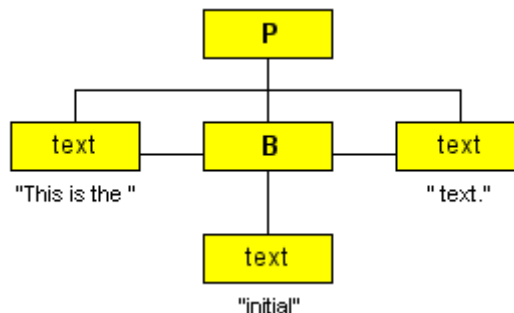


So `document.getElementById('sample1').firstChild.nodeValue` is used to access this text node and read or set its string value.

It's important to remember that text nodes contain just that, text. Even simple markup tags like B or I within a string of text will create a sub tree of element and text nodes. For example, using the example above and adding tags make the word "initial" bold:

```
<p id="sample2">This is the <b>initial</b> text.</p>
```

now gives the "sample2" paragraph element three children instead of one. There is a text node for "This is the ", an element node for the B tag pair and a text node for " text.". The node for the B element has one child node, a text node for "initial". You can see the structure in the diagram below.



To see how this affects scripting code, here's the same example used above but with the additional bold markup:

This is the **initial** text.

[Change Text 1](#) | [Change Text 2](#)

Changing `firstChild` of the P element now only affects the text "This is the ". Conversely, if you attempt to add markup to the value of a text node, the browser will treat it as plain text, as demonstrated below:

This is the **initial** text.

[Change Text 1](#) | [Change Text 2](#)

Where the link code has been changed to this:

```
document.getElementById('sample3').firstChild.nodeValue =  
'<b>Once</b> upon a time...';
```

```
document.getElementById('sample3').firstChild.nodeValue =
```

'...in a galaxy <i>far, far</i> away';

You can avoid problems like this by thinking of text nodes as individual strings of character data located between *any* two HTML tags, not necessarily matching pairs of tags.

### **Adding Nodes**

Nodes can also be added to the DOM. You've already seen how attribute nodes can be created and applied to an element so let's look at adding element and text nodes within the document tree (without using the innerHTML property).

The first step is to create a node object of the type you want using one of document.createElement(), document.createAttribute() or document.createTextNode(). For attributes, however, you'll probably just want to create an element node and assign it attributes directly (recall that IE, as of version 5.5, doesn't support createAttribute()).

### **Working with Text Nodes**

Let's start with a text node. Below is some sample code showing how to create a text node and assign it a value.

```
var myTextNode = document.createTextNode("my text");
```

Now you have a text node. But it's not part of the document tree. To make it appear on the page, you need to add it as a child to an existing node within the tree. Since text nodes cannot have children, you can't attach it to another text node. Attributes nodes are not part of the document tree, so you don't want to attach it to one of them. That leaves element nodes.

Since element nodes can have several children, there are a few different methods provided that allow you to specify where to add the new node among its existing children. These are best illustrated by example.

Here, the appendChild() method is used to add new text to a paragraph element. It also allows you to remove the last node added using the removeChild() method:

Initial text within a paragraph element.

Add Text Node | Remove Text Node

Now take a look at the code:

```
<p id="sample1">Initial text within a paragraph element.</p>
```

... code to add a text node ...

```
var text = document.createTextNode(" new text " + (++counter1));  
var el = document.getElementById("sample1");  
el.appendChild(text);
```

... code to remove the last child node ...

```
var el = document.getElementById("sample1");  
if (el.hasChildNodes())  
    el.removeChild(el.lastChild);
```

Adding text is easy, the code creates a new text node, locates the paragraph element node and calls appendChild() to insert it at the end of its childNodes array. A global counter variable is used on the text itself so you can distinguish each new node in the browser display.

Removing text is almost as easy, using a call to `removeChildNode()`. The only difference in the addition of a reference node, to indicate which of the element's children is to be removed. Here we use the element's `lastChild` property which always points to the last node of the element's `childNodes` array. Note that it will even remove the initial text hard coded in the P tag if that is the only child.

Also note the use of the `hasChildNodes()` method which simply returns true or false to indicate if the given node currently has any children. Here it's used to prevent an error by calling `removeChild` when there are no children left.

## THE DOM EVENT HANDLING

One of the keys to creating dynamic web pages is the use of event handlers. These allow you to execute specific script code in response to user or system initiated actions. Most events relate to the browser GUI, such as mouse movements, button or key clicks and updates to form inputs. These are usually tied to a specific page element. Others relate to browser actions such as when a document or image completes loading. Some objects have default actions defined for certain events, such as clicking on a hypertext link. The browser's normal action in that event is to load the URL associated with the link.

In any case, all events follow the same model. The DOM provides methods for capturing events so you can perform your own actions in response to them. It also provides an Event object which contains information specific to a given event that can be used by your event processing code.

### Assigning Event Handlers

There are several ways to set up the capture of an event on an object using either HTML or scripting. In each case, you assign a function to handle the specific event when it occurs.

### HTML Tag Attributes

Many HTML tags have *intrinsic* events associated with them. You can define script code to be executed when one of these events occurs using the event name as an attribute. For example,

```
<span style="background-color:yellow;"  
  onmouseover="this.style.backgroundColor='black';this.style.color='white'"  
  onmouseout="this.style.backgroundColor='yellow';this.style.color=''">
```

Sample element with mouse event handlers.

```
</span>
```

Here, two different events are captured, `mouseover` and `mouseout`. The value assigned to the corresponding attributes is a string containing JavaScript code. Note the use of single quotes (') for string constants within the double quotes (") used to delimit the attribute value. The script code for both events simply changes the text and background colors on the element's style. The keyword `this` keyword refers to the object that fired the event. Here it refers to the SPAN element. It was stated before that when you set up capturing for an event, you assign a function to be called to handle that event. This seems to contradict the example above where the event code is simply a couple of JavaScript statements. However, the browser actually creates an anonymous function for the handler with those statements in it.

You should see an alert box with a function definition. The actual definition will vary according to browser but it will contain a single statement, the one defined in the HTML:  
function *name(argument\_list)* {

```
    alert(this.onclick);  
}
```

This corresponds to the code set on the ONCLICK attribute:

```
<span style="background-color:yellow;"  
  onclick="alert(this.onclick)">Sample element with an onclick event  
handler.</span>
```

### Scripting

You can also assign event handlers to elements in the DOM using client-side scripting. Like other element attributes, events are represented as properties of the element object so you can set their value just like any other attribute. The main difference is that unlike most attributes, which take a string value, event handlers must be set using a function reference. Here's an example,

```
<span id="sample1" style="background-color:yellow;">Sample element with  
mouse event handlers.</span>
```

... JavaScript code ...

```
function highlight(event) {  
  
    this.style.backgroundColor='black';  
    this.style.color='white';  
}
```

```
function normal(event) {  
  
    this.style.backgroundColor='yellow';  
    this.style.color="";  
}
```

```
document.getElementById('sample1').onmouseover = highlight;  
document.getElementById('sample1').onmouseout = normal;
```

Note that the onmouseover property is set to highlight, i.e., the name of the function without parenthesis. This represents the Function object for highlight which is what the event property expects, a reference to the function to call when the event is fired. If highlight() had been used instead, the browser would call the function and assign whatever value it returned as the event handler. In this particular case, an error would result in the function execution because this has no meaning in the given context.

### Event Listeners

DOM objects can also be registered as event listeners. This feature can be used to assign multiple handlers for a given event. It also allows you to capture events during either one of two phases in the event flow, *capture* or *bubble*.

The difference between these two phases will be covered later on in the discussion on event flow. But for now we'll just look at how to register an event listener.

The basic methods and syntax are

```
node.addEventListener(eventType, function, useCapture);  
node.removeEventListener(eventType, function);
```

where *eventType* is a predefined event name such as "mouseover" or "click" (the same as the corresponding event attribute but without the preceding "on"), *function* is the name of the handler function and *useCapture* is a boolean flag which specifies which phase of the event flow the handler will be called on.

To demonstrate, the previous example could have assigned event handlers for the sample paragraph element with the following:

```
var el = document.getElementById('sample1');  
el.addEventListener("mouseover", highlight, false);  
el.addEventListener("mouseout", normal, false);
```

Additional handlers could also be assigned for a given event:

```
el.addEventListener("mouseover", highlight2, true);  
el.addEventListener("mouseover", highlight3, false);
```

Individual event listeners can be subsequently removed by specifying the same arguments:

```
el.removeEventListener("mouseover", highlight2, true);  
el.removeEventListener("mouseover", highlight3, false);
```

Note that you must specify the same *useCapture* value when removing a listener as when it was added. This is because,

```
el.addEventListener("mouseover", highlight, true);  
el.addEventListener("mouseover", highlight, false);
```

defines two unique event listeners, calling the same function for the same event on the same element but which activate during different phases.

One advantage of this method is that event listeners can be assigned to any node, even text nodes which you could not normally assign event handlers to as they have no attributes.

### Event Flow

Before going into event processing, it's helpful to understand event flow in the DOM. HTML documents (and XML or XHTML documents) are hierarchical in nature. Elements and text are nested within other elements. Because of this, when an event occurs on a particular object, it effectively occurs on any containing objects as well.

To illustrate, consider the following HTML:

```
<div>  
<p>Some text in a paragraph element.</p>  
<p>Another paragraph element with text.</p>  
<p>Yet another paragraph with text and also a  
  <a href="blank.html">link</a>.</p>  
</div>
```

If you click on the link defined in this code sample, it will trigger a onclick event on the A tag. But you're also clicking on the paragraph that contains that link, and the DIV that contains that P element and so on up to the document object itself. Any of the elements in this chain can have an event handler assigned to it to capture the onclick event, regardless of which element it originated at.

### Event Bubbling

The DOM event model provides for this using a concept called *event bubbling*. For example, suppose an onclick event handler were assigned to the DIV tag above. Clicking



on the link would fire the event first on the A element, it would then "bubble" up to the containing P element and then up to the DIV where the handler function would be called. It's possible for the handler to cancel the event, but assuming it doesn't the event would continue bubbling on up to the document root and finally, the browser would follow the default action of loading the link's URL in the window. Note that the P element could also have had an onclick event handler set, as could any elements above the DIV in the document tree. All of these handlers would be called in turn as the event bubbles up to the document root. This is known as the *bubble* phase in the DOM event model. Not all events bubble, for example onfocus and onblur do not. Likewise, not all bubbling events can be canceled, stopping the propagation. You can determine which events bubble and can be canceled either by looking up the documentation for the event or, as we'll see, using the Event object.

### Event Capture

You can also catch events during the *capture* phase using the event listener detailed previously. The capture phase compliments the bubble phase. The capture phase runs first. The event flows down from the root of the document tree to the target element, then it bubbles back up.

In this phase, outer elements will receive the event before it reaches its intended target. This can be useful if you want to intercept an event for some element even if it was initially targeted at one of its children or other descendants.

It should be noted that the term "event capture" is often used loosely to describe the act of setting an event handler or listener, during either phase. Here it specifically means intercepting an event during this downward phase of the event flow, before it reaches its intended target.

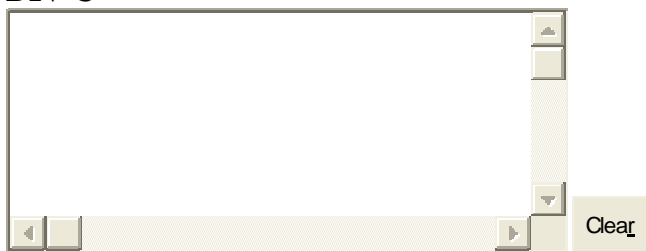
### Event Flow Example

The full process can be illustrated with a demo. Below is a set of nested DIV tags. The onclick event is caught for each element both on the capture phase (if supported by the browser) and the bubble phase. Click anywhere on the boxes and the path of the event will be traced in the text box below it.

**DIV A**

**DIV B**

**DIV C**



Clicking on the innermost DIV fires six event handlers, three going down the document tree and three as it bubbles back up. In this example, the same function happens to be used for all six, but each could be assigned a unique handler function.

In the trace, the "Target" element is the one that the event initiates from while the "Current" element is the one that has the event listener attached. Both of these values are derived using the Event object passed to the handler.

## The Event Object

Within an event handler, you can do pretty much anything you want with your script code. Chances are, you'll want to perform some action related to that event depending on one or more factors.

Recall that event handlers are passed one argument, an Event object. It provides several properties describing the event and its current state. You can use these to determine where an event originated from and where it currently is in the event flow. Or use the methods it provides to stop the event from flowing on and/or cancel the event.

Event Properties and Methods	
Property Name	Description
bubbles	A Boolean value indicating whether or not the event bubbles.
cancelable	A Boolean value indicating whether or not the event can be canceled.
currentTarget	The node that this event handler is assigned to.
eventPhase	An integer value indicating which phase of the event flow this event is being processed in. One of CAPTURING_PHASE (1), AT_TARGET (2) or BUBBLING_PHASE (3).
target	The node that the event originated from.
timeStamp	The time the event occurred.
type	A string indicating the type of event, such as "mouseover", "click", etc.
Method Name	Description
preventDefault()	Can be used to cancel the event, if it is cancelable. This prevents the browser from performing any default action for the event, such as loading a URL when a hypertext link is clicked. Note that the event will continue propagating along the normal event flow.
stopPropagation()	Stops the event flow. This can be used on either the capture or bubble phase.

These properties are common to all event types. Some additional properties are available for specific events related to the mouse and keyboard.

### Mouse Events

Mouse-related events include:

- click
- mousedown
- mouseup
- mouseover
- mouseout
- mousemove

For these events, additional information is provided in the Event object.

### Mouse Event Properties

Property Name	Description
altKey, ctrlKey, metaKey, shiftKey	Boolean values. If true, the associated key was depressed when the event fired.
button	An integer indicating which mouse button was pressed or released, 1 = left, 2 = middle, 3 = right. A one-button mouse will return only 1, a two-button mouse could return either 1 or 3.
clientX, clientY	Give the pixel coordinates of the mouse relative to the client area of the browser, i.e., relative to the viewport, when the event fired.
relatedTarget	On a mouseover this indicates the node that the mouse has left. On a mouseout it indicates the node the mouse has moved onto.
screenX, screenY	Give the pixel coordinates of the mouse relative to the screen when the event fired.

As you can see, these properties can be used to determine the position of the mouse, what buttons are depressed and the elements it is moving over or leaving.

### Notes on Mouse Events

Mouse events are always taken to occur on the lowest element in the document tree. That is, the target of the event will be the most deeply nested element under the mouse when the event occurs.

### Keyboard Events

The DOM2 Event Model does not include specifications for key events. However, the HTML 4 standard does permit the keyup, keydown and keypress events for many elements. Both Netscape 6 and IE 5.5 support these and include properties in the Event object to reflect information on what key or keys were pressed. In Netscape, the ASCII value of the key is given by the charCode property on keypress events and in the keyCode property for keydown and keyup events. Internet Explorer stores the Unicode value of the key in the event keyCode property for all three key events.

In both, the keydown and keyup events will fire when any modifier key is pressed, such as ALT, CTRL and SHIFT. The keypress event can be used instead capture combinations such as SHIFT-A. Some example key combinations with the relevant property values for each key event type are shown below, arranged by browser.

IE Key Event Examples			
Key(s)	keydown	keyup	keypress
A	keyCode=65	keyCode=65	keyCode=97
SHIFT	keyCode=16, shiftKey=true	keyCode=16, shiftKey=false	n/a
SHIFT-A	Whichever key is pressed first fires the event.	Whichever key is released last fires the event.	keyCode=65, shiftKey=true
Z	keyCode=90	keyCode=90	keyCode=122
CTRL	keyCode=17, ctrlKey=true	keyCode=17, ctrlKey=false	n/a

<b>CTRL-Z</b>	Whichever key is pressed first fires the event.	Whichever key is released last fires the event.	keyCode=26, ctrlKey=true
---------------	---	---	--------------------------

#### Netscape Key Event Examples

Key(s)	keydown	keyup	keypress
<b>A</b>	charCode=0, keyCode=65, which=65	charCode=0, keyCode=65, which=65	charCode=97, keyCode=0, which=97
<b>SHIFT</b>	charCode=0, keyCode=16, shiftKey=true	charCode=0, keyCode=16, shiftKey=false	n/a
<b>SHIFT-A</b>	Whichever key is pressed first fires the event.	Fires separate events for each key, shiftKey=true for A key.	charCode=65, keyCode=0, which=65, shiftKey=false (yes, false)
<b>Z</b>	charCode=0, keyCode=90, which=90	charCode=0, keyCode=90, which=90	charCode=122, keyCode=0, which=122
<b>CTRL</b>	charCode=0, keyCode=17, ctrlKey=true	charCode=0, keyCode=17, ctrlKey=false	n/a
<b>CTRL-Z</b>	Whichever key is pressed first fires the event.	Fires separate events for each key, ctrlKey=true for Z key.	charCode=122, keyCode=0, which=122, ctrlKey=true

As you can see, the two browsers differ quite a bit in how they handle key events. Also note that some key combinations are reserved by the browser or operating system for shortcut commands or controls. Depending on the browser and platform, you may or may not be able to capture these.

#### Handling Events

With dozens of different events constantly firing and moving up and down the document tree, the real trick to using event handlers is in figuring out what event to catch, where to attach the handler and how to process it.

Sometimes the answers are obvious and sometimes they aren't. To demonstrate, let's look at a particular effect and see how to implement it using event handlers.

#### Example Using Mouse Events

Menu Item 1	Menu Item 2	Menu Item 3	Menu Item 4	Menu Item 5
-------------	-------------	-------------	-------------	-------------

Suppose you wanted to create a pop up menu like the one at right. It consists of a DIV tag with several A tags for the individual items. CSS styles are used to achieve the look including a hover effect on the item links.

The hover effect could have been done using events but why complicate matters when the CSS :hover pseudo-class does the job? Besides, we'll be using events to handle the hiding and showing of the menu.

Here's what the CSS and HTML code look like:

```
.menu {
  background-color: #40a0c0;
  border-color: #80e0ff #006080 #006080 #80e0ff;
  border-style: solid;
  border-width: 2px;
  position: absolute;
  left: 0px;
  top: 0px;
  visibility: hidden;
}

a.menuItem {
  color: #ffffff;
  cursor: default;
  display: block;
  font-family: MS Sans Serif, Arial, Tahoma,sans-serif;
  font-size: 8pt;
  font-weight: bold;
  padding: 2px 12px 2px 8px;
  text-decoration: none;
}

a.menuItem:hover {
  background-color: #004060;
  color: #ffff00;
}
```

...

```
<div id="menuId" class="menu">
<a class="menuItem" href="blank.html">Menu Item 1</a>
<a class="menuItem" href="blank.html">Menu Item 2</a>
<a class="menuItem" href="blank.html">Menu Item 3</a>
<a class="menuItem" href="blank.html">Menu Item 4</a>
<a class="menuItem" href="blank.html">Menu Item 5</a>
</div>
```

Say we want to make the menu appear at the cursor when the user clicks on a particular link. We set up a function called `openMenu()` and add an `onclick` handler on the A tag like this:

```
<a href=""
  onclick="openMenu(event, 'menuId');return false;">Open Menu</a>
```

Notice that `openMenu()` expects two arguments, an Event object and the ID of the menu DIV to display. Remember that the string value of the `ONCLICK` attribute is actually used to create an anonymous function which, in this case, would look something like this,

```
function anonymous(event) {
  openMenu(event, 'menuId');
```

```

    return false;
}

```

with the menu DIV id hard-coded. So the handler really has only one argument, event, as all event handlers do. Returning false will prevent the browser default action of following the link.

Here's the openMenu() function itself:

```

function openMenu(event, id) {

    var el, x, y;

    el = document.getElementById(id);
    if (window.event) {
        x = window.event.clientX + document.documentElement.scrollLeft
            + document.body.scrollLeft;
        y = window.event.clientY + document.documentElement.scrollTop
            + document.body.scrollTop;
    }
    else {
        x = event.clientX + window.scrollX;
        y = event.clientY + window.scrollY;
    }
    x -= 2; y -= 2;
    el.style.left = x + "px";
    el.style.top = y + "px";
    el.style.visibility = "visible";
}

```

It basically just positions the menu DIV under the cursor and makes it visible. The positioning is complicated a bit by the fact that different methods are needed to determine the cursor position based on the browser.

## ACCOMMODATING NONCOMPLIANT BROWSERS

The navigator object was conceived back in the days when Netscape Navigator reigned supreme. These days it serves as much as an irony of NS's diminished market share as way of probing browser information. The navigator object of JavaScript contains the following core properties:

Properties	Description
appName	The code name of the browser.
appName	The name of the browser (ie: Microsoft Internet Explorer).
appVersion	Version information for the browser (ie: 4.75 [en] (Win98; U) ).
cookieEnabled	Boolean that indicates whether the browser has cookies enabled.
language	Returns the default language of the browser version (ie:

	en-US). <b>NS and Firefox only.</b>
mimeTypes[]	An array of all MIME types supported by the client. <b>NS and Firefox only.</b>
platform[]	The platform of the client's computer (ie: Win32).
plugins	An array of all plug-ins currently installed on the client. <b>NS and Firefox only.</b>
systemLanguage	Returns the default language of the <a href="#">operating system</a> (ie: en-us). <b>IE only.</b>
userAgent	String passed by browser as user-agent header. (ie: <a href="#">Mozilla</a> /4.0 (compatible; MSIE 6.0; Windows NT 5.1))
userLanguage	Returns the preferred language setting of the user (ie: en-ca). <b>IE only.</b>

Let's see exactly what these properties reveal of the browser you're **currently using**:

### At a glance

At a glance at the above table, you may be swayed towards turning to the following two properties to do your browser detection bidding:

```
navigator.appName
navigator.appVersion
```

After all, you are trying to detect a browser's name and version right? However, they both will most likely mislead you. In browsers such as various versions of Netscape and Firefox, these two properties return simply "Netscape" for `appName`, and 5.0 for `appVersion` without any further distinction for Firefox and its version, and hence are pretty much useless in the real world. For example, in both Firefox 1.x and Firefox 2.x, these two properties return:

```
appName: Netscape
appVersion: 5.0 (Windows; en-US)
```

We need to turn to a property that's more thorough in its investigative work if we want more consistency and accuracy, and that turns out to be `navigator.userAgent`.

### Detecting Firefox x.x

In Firefox 2.0.0.13 for example, the `userAgent` property reads:

```
UserAgent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.13)
Gecko/20080311 Firefox/2.0.0.13
```

The detail we're interested in apparently lies at the very end, or `Firefox/2.0.0.13`. Different versions of Firefox will contain a different version number, but the pattern is consistent enough. The part we're interested in occurs after the string "Firefox/", or the exact version number. There are many ways to get to it using either standard [String](#) or [RegExp](#) methods- I'm opting for the later here:

```
<script type="text/javascript">

if
(/Firefox[\/\s](\d+\.\d+)/.test(navigator.userAgent)){ //te
st for Firefox/x.x or Firefox x.x (ignoring remaining
digits);
var ffversion=new Number(RegExp.$1) // capture x.x portion
and store as a number
if (ffversion>=3)
document.write("You're using FF 3.x or above")
}
```

```

else if (ffversion>=2)
    document.write("You're using FF 2.x")
else if (ffversion>=1)
    document.write("You're using FF 1.x")
}
else
    document.write("n/a")

</script>

```

## Detecting IE x.x

In IE 7.0 for example, the `userAgent` property reads:

**UserAgent:** Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 2.0.50727)

So the part we're interested in lies in the middle, or `MSIE 7.0`; . If you try a shortcut and use `parseFloat` on the entire string to get to the 7.0 portion, it won't work. This is due to the way `parseFloat` works- by returning the first number it encounters, which in this case is 4.0. Once again we need to use either standard `String` or `RegExp` methods again to get to the actual version number; below I'm using `RegExp` as well:

```
<script type="text/javascript">
```

```

if (/MSIE (\d+\.\d+);/.test(navigator.userAgent)){ //test for
MSIE x.x;
    var ieversion=new Number(RegExp.$1) // capture x.x portion and
store as a number
    if (ieversion>=8)
        document.write("You're using IE8 or above")
    else if (ieversion>=7)
        document.write("You're using IE7.x")
    else if (ieversion>=6)
        document.write("You're using IE6.x")
    else if (ieversion>=5)
        document.write("You're using IE5.x")
    }
else
    document.write("n/a")
</script>

```

## ADDITIONAL PROPERTIES OF WINDOWS

### Window Object

The window object represents an open window in a browser.

If a document contain frames (`<frame>` or `<iframe>` tags), the browser creates one window object for the HTML document, and one additional window object for each frame.

**Note:** There is no public standard that applies to the Window object, but all major browsers support it.

### Window Object Properties

Property	Description
<a href="#">closed</a>	Returns a Boolean value indicating whether a window has been closed or not
<a href="#">defaultStatus</a>	Sets or returns the default text in the statusbar of a window
<a href="#">document</a>	Returns the Document object for the window ( <a href="#">See Document object</a> )
<a href="#">frames</a>	Returns an array of all the frames (including iframes) in the current window



<a href="#">history</a>	Returns the History object for the window ( <a href="#">See History object</a> )
<a href="#">innerHeight</a>	Sets or returns the the inner height of a window's content area
<a href="#">innerWidth</a>	Sets or returns the the inner width of a window's content area
<a href="#">length</a>	Returns the number of frames (including iframes) in a window
<a href="#">location</a>	Returns the Location object for the window ( <a href="#">See Location object</a> )
<a href="#">name</a>	Sets or returns the name of a window
<a href="#">navigator</a>	Returns the Navigator object for the window ( <a href="#">See Navigator object</a> )
<a href="#">opener</a>	Returns a reference to the window that created the window
<a href="#">outerHeight</a>	Sets or returns the outer height of a window, including toolbars/scrollbars
<a href="#">outerWidth</a>	Sets or returns the outer width of a window, including toolbars/scrollbars
<a href="#">pageXOffset</a>	Returns the pixels the current document has been scrolled (horizontally) from the upper left corner of the window
<a href="#">pageYOffset</a>	Returns the pixels the current document has been scrolled (vertically) from the upper left corner of the window
<a href="#">parent</a>	Returns the parent window of the current window
<a href="#">screen</a>	Returns the Screen object for the window ( <a href="#">See Screen object</a> )
<a href="#">screenLeft</a>	Returns the x coordinate of the window relative to the screen
<a href="#">screenTop</a>	Returns the y coordinate of the window relative to the screen
<a href="#">screenX</a>	Returns the x coordinate of the window relative to the screen
<a href="#">screenY</a>	Returns the y coordinate of the window relative to the screen
<a href="#">self</a>	Returns the current window
<a href="#">status</a>	Sets the text in the statusbar of a window
<a href="#">top</a>	Returns the topmost browser window

## Window Object Methods

Method	Description
<a href="#">alert()</a>	Displays an alert box with a message and an OK button
<a href="#">blur()</a>	Removes focus from the current window
<a href="#">clearInterval()</a>	Clears a timer set with setInterval()
<a href="#">clearTimeout()</a>	Clears a timer set with setTimeout()
<a href="#">close()</a>	Closes the current window
<a href="#">confirm()</a>	Displays a dialog box with a message and an OK and a Cancel button
<a href="#">createPopup()</a>	Creates a pop-up window
<a href="#">focus()</a>	Sets focus to the current window
<a href="#">moveBy()</a>	Moves a window relative to its current position
<a href="#">moveTo()</a>	Moves a window to the specified position
<a href="#">open()</a>	Opens a new browser window
<a href="#">print()</a>	Prints the content of the current window
<a href="#">prompt()</a>	Displays a dialog box that prompts the visitor for input
<a href="#">resizeBy()</a>	Resizes the window by the specified pixels
<a href="#">resizeTo()</a>	Resizes the window to the specified width and height

scroll()	
scrollBy()	Scrolls the content by the specified number of pixels
scrollTo()	Scrolls the content to the specified coordinates
setInterval()	Calls a function or evaluates an expression at specified intervals (in milliseconds)
setTimeout()	Calls a function or evaluates an expression after a specified number of milliseconds

The DOM API provided by the Web browser contains a object called "window", which has the following features:

- The "window" object is a browser built-in object that represents the browser window that contains the current HTML document.
- The "window" object offers a number of properties and methods to allow you
- The "document" object and its "node" objects offers various methods to allow you to manipulate the window.

To illustrate some nice features of the "document" object, I wrote this JavaScript tutorial example: <html>

```
<!-- Window_Object.html
Copyright (c) 2008 by Dr. Herong Yang, http://www.herongyang.com/
-->
<head>
<title>Window Object</title>
<script type="text/javascript">
var sizeID = 0;
function changeSize() {
    if (sizeID == 0 )
        window.resizeTo(300,300);
    else if (sizeID == 1 )
        window.resizeTo(400,400);
    else if (sizeID == 2 )
        window.resizeTo(500,500);
    else if (sizeID == 3 )
        window.resizeTo(600,600);
    sizeID = (sizeID+1)%4;
}
</script>
</head>
<body>
<p>Hello World!</p>
<p><form>
    <input type="button" value="Change" onClick="changeSize();" />
</form></p>
</body>
</html>
```

Run this JavaScript example in a Web browser, and click the "Change" button. The browser window will change its size each time you click the button.

For more tutorial examples on the "window" object, see the DOM API chapters in this book.

## CASE STUDY

### SERVER-SIDE PROGRAMMING: JAVA SERVLETS

#### What is a Servlet?

Servlets are modules of Java code that run in a server application (hence the name "Servlets", similar to "Applets" on the client side) to answer client requests. Servlets are not tied to a specific client-server protocol but they are most commonly used with HTTP and the word "Servlet" is often used in the meaning of "HTTP Servlet".

Servlets make use of the Java standard extension classes in the packages `javax.servlet` (the basic Servlet framework) and `javax.servlet.http` (extensions of the Servlet framework for Servlets that answer HTTP requests). Since Servlets are written in the highly portable Java language and follow a standard framework, they provide a means to create sophisticated server extensions in a server and operating system independent way.

Typical uses for HTTP Servlets include:

- Processing and/or storing data submitted by an HTML form.
- Providing dynamic content, e.g. returning the results of a database query to the client.
- Managing state information on top of the stateless HTTP, e.g. for an online shopping cart system which manages shopping carts for many concurrent customers and maps every request to the right customer.

### Servlets vs CGI

The traditional way of adding functionality to a Web Server is the [Common Gateway Interface \(CGI\)](#), a language-independent interface that allows a server to start an external process which gets information about a request through environment variables, the command line and its standard input stream and writes response data to its standard output stream. Each request is answered in a separate process by a separate instance of the CGI program, or CGI script (as it is often called because CGI programs are usually written in interpreted languages like Perl).

Servlets have several advantages over CGI:

- A Servlet does not run in a separate process. This removes the overhead of creating a new process for each request.
- A Servlet stays in memory between requests. A CGI program (and probably also an extensive runtime system or interpreter) needs to be loaded and started for each CGI request.
- There is only a single instance which answers all requests concurrently. This saves memory and allows a Servlet to easily manage persistent data.
- A Servlet can be run by a Servlet Engine in a restrictive [Sandbox](#) (just like an Applet runs in a Web Browser's Sandbox) which allows secure use of untrusted and potentially harmful Servlets.

### THE BASIC SERVLET ARCHITECTURE

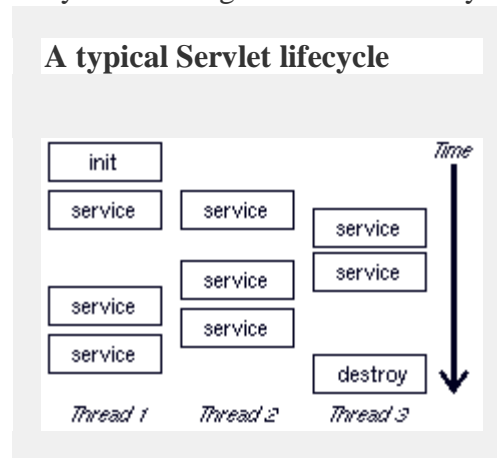
A Servlet, in its most general form, is an instance of a class which implements the `javax.servlet.Servlet` interface. Most Servlets, however, extend one of the standard implementations of that interface, namely `javax.servlet.GenericServlet` and `javax.servlet.http.HttpServlet`. In this tutorial we'll be discussing only HTTP Servlets which extend the `javax.servlet.http.HttpServlet` class.

In order to initialize a Servlet, a server application loads the Servlet class (and probably other classes which are referenced by the Servlet) and creates an instance by calling the no-args constructor. Then it calls the Servlet's `init(ServletConfig config)` method. The Servlet should perform one-time setup procedures in this method and store the `ServletConfig` object so that it can be retrieved later by calling the Servlet's `getServletConfig()` method. This is handled by `GenericServlet`. Servlets which extend `GenericServlet` (or its subclass `HttpServlet`) should call `super.init(config)` at the beginning of the `init` method to make use of this feature. The `ServletConfig` object contains Servlet parameters and a reference to the Servlet's `ServletContext`. The `init` method is guaranteed

to be called only once during the Servlet's lifecycle. It does not need to be thread-safe because the `service` method will not be called until the call to `init` returns.

When the Servlet is initialized, its `service(ServletRequest req, ServletResponse res)` method is called for every request to the Servlet. The method is called concurrently (i.e. multiple threads may call this method at the same time) so it should be implemented in a thread-safe manner. Techniques for ensuring that the `service` method is not called concurrently, for the cases where this is not possible, are described in [section 4.1](#).

When the Servlet needs to be unloaded (e.g. because a new version should be loaded or the server is shutting down) the `destroy()` method is called. There may still be threads that execute the `service` method when `destroy` is called, so `destroy` has to be thread-safe. All resources which were allocated in `init` should be released in `destroy`. This method is guaranteed to be called only once during the Servlet's lifecycle.



## HTTP

Before we can start writing the first Servlet, we need to know some basics of HTTP ("HyperText Transfer Protocol"), the protocol which is used by a WWW client (e.g. a browser) to send a request to a Web Server.

HTTP is a request-response oriented protocol. An HTTP request consists of a request method, a URI, header fields and a body (which can be empty). An HTTP response contains a result code and again header fields and a body.

The `service` method of `HttpServlet` dispatches a request to different Java methods for different HTTP request methods. It recognizes the standard HTTP/1.1 methods and should not be overridden in subclasses unless you need to implement additional methods. The recognized methods are GET, HEAD, PUT, POST, DELETE, OPTIONS and TRACE. Other methods are answered with a *Bad Request* HTTP error. An HTTP method XXX is dispatched to a Java method `doXxx`, e.g. `GET` -> `doGet`. All these methods expect the parameters "(`HttpServletRequest req`, `HttpServletResponse res`)". The methods `doOptions` and `doTrace` have suitable default implementations and are usually not overridden. The HEAD method (which is supposed to return the same header lines that a GET method would return, but doesn't include a body) is performed by calling `doGet` and ignoring any output that is written by this method. That leaves us with the methods `doGet`, `doPut`, `doPost` and `doDelete` whose default implementations in `HttpServlet` return a *Bad Request* HTTP error. A subclass of `HttpServlet` overrides one or more of these methods to provide a meaningful implementation.

The request data is passed to all methods through the first argument of type `HttpServletRequest` (which is a subclass of the more general `ServletRequest` class). The response can be created with methods of the second argument of type `HttpServletResponse` (a subclass of `ServletResponse`).

When you request a URL in a Web Browser, the GET method is used for the request. A GET request does not have a body (i.e. the body is empty). The response should contain a body with the response data and header fields which describe the body (especially `Content-Type` and `Content-Encoding`). When you send an HTML form, either GET or POST can be used. With a GET request the parameters are encoded in the URL, with a POST request they are transmitted in the body. HTML editors and upload tools use PUT requests to upload resources to a Web Server and DELETE requests to delete resources. The complete HTTP specifications can be found in [RFCs 1945](#) (HTTP/1.0) and [2068](#) (HTTP/1.1).

## A "HELLO WORLD" SERVLET

### Hello World!

*This section shows how to*

- *use the framework that makes up a simple Servlet*
- *write a Servlet that provides static content (i.e. it produces the same output every time it is called by a client)*

We start our venture into Servlet programming with the well-known "Hello World" example, this time named more suitably "Hello Client":

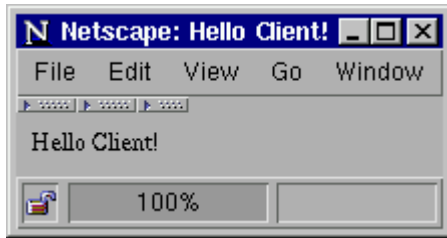
`HelloClientServlet.java`

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloClientServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest req,
                          HttpServletResponse res)
        throws ServletException, IOException
    {
        res.setContentType("text/html");
        PrintWriter out = res.getWriter();
        out.println("<HTML><HEAD><TITLE>Hello Client!</TITLE>"+
                    "</HEAD><BODY>Hello Client!</BODY></HTML>");
        out.close();
    }

    public String getServletInfo()
    {
        return "HelloClientServlet 1.0 by Stefan Zeiger";
    }
}
```

then you compile this Servlet and run it by requesting a URL which is assigned to it in a Web Browser it produces the following output:



Let's have a look at how the Servlet works.

Lines 1 to 3 import some packages which contain many classes which are used by the Servlet (almost every Servlet needs classes from these packages).

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

The Servlet class is declared in line 5. Our Servlet extends `javax.servlet.http.HttpServlet`, the standard base class for HTTP Servlets.

```
public class HelloClientServlet extends HttpServlet
```

In lines 7 through 16 `HttpServlet`'s `doGet` method is getting overridden.

```
protected void doGet(HttpServletRequest req,
                      HttpServletResponse res)
    throws ServletException, IOException
: {
    ...
}
```

In line 11 we use a method of the `HttpServletResponse` object to set the content type of the response that we are going to send. All response headers must be set *before* a `PrintWriter` or `ServletOutputStream` is requested to write body data to the response.

```
res.setContentType("text/html");
```

In line 12 we request a `PrintWriter` object to write text to the response message.

```
PrintWriter out = res.getWriter();
```

**API 2.0** `ServletResponse.getWriter()` is a new feature of JSDK version 2.0. If your Servlet engine does not support JSDK 2.0 you can replace the above line by `"ServletOutputStream out = res.getOutputStream();" . This change can be made in most of the example Servlets in this tutorial. The advantages of using ServletResponse.getWriter() are discussed in section 4.4.`

In lines 13 and 14 we use the `PrintWriter` to write the text of type *text/html* (as specified through the content type).

```
out.println("<HTML><HEAD><TITLE>Hello Client!</TITLE>"+
            "</HEAD><BODY>Hello Client!</BODY></HTML>");
```

The `PrintWriter` gets closed in line 15 when we are finished writing to it.

```
out.close();
```

This line is included for completeness. It is not strictly necessary. The Web Server closes the `PrintWriter` or `ServletOutputStream` automatically when a service call returns. An explicit call to `close()` is useful when you want to do some post-processing after the response to the client has been fully written. Calling `close()` tells the Web Server that the response is finished and the connection to the client may be closed as well.

In lines 18 through 21 we override the `getServletInfo()` method which is supposed to return information about the Servlet, e.g. the Servlet name, version, author and copyright

notice. This is not required for the function of the `HelloClientServlet` but can provide valuable information to the user of a Servlet who sees the returned text in the administration tool of the Web Server.

```
public String getServletInfo()
{
    return "HelloClientServlet 1.0 by Stefan Zeiger";
}
```

## **SERVLETS GENERATING DYNAMIC CONTENT**

This section shows how to

- process form data
- manage persistent data
- use init parameters

The next Servlet that we are going to write provides a user interface to a mailing list through HTML forms. A user should be able to enter an email address in a text field and press a button to subscribe to the list or another button to unsubscribe.

The Servlet consists of two major parts: Data management and client interaction.

### **Data management**

The data management is rather straight-forward for an experienced Java programmer. We use a `java.lang.Vector` object which contains the email addresses as Strings. Since a Servlet can have data which persists between requests we load the address list only once, when the Servlet is initialized, and save it every time it has been changed by a request.

An alternative approach would be keeping the list in memory while the Servlet is active and writing it to disk in the destroy method. This would avoid the overhead of saving the address list after every change but is less fail-safe. If for some reason the address file can't be written to disk or the server crashes and cannot destroy the Servlet, all changes to the list will be lost even though the users who submitted the requests to change the list received positive responses.

The following parts of the Servlet are related to data management:

```
private Vector addresses;
private String filename;
public void init(ServletConfig config) throws ServletException
{
    super.init(config);
    filename = config.getInitParameter("addressfile");
    if(filename == null)
        throw new UnavailableException(this,
            "The \"addressfile\" property "+
            "must be set to a file name");
    try
    {
        ObjectInputStream in =
            new ObjectInputStream(new FileInputStream(filename));
        addresses = (Vector)in.readObject();
        in.close();
    }
```

```
26: catch(FileNotFoundException e) { addresses = new Vector(); }
```

```

27:    catch(Exception e)
28:    {
        throw new UnavailableException(this,
                                      "Error reading address file: "+e);
    }
}

private synchronized boolean subscribe(String email) throws IOException
{
    if(addresses.contains(email)) return false;
    addresses.addElement(email);
    save();
    return true;
}
private synchronized boolean unsubscribe(String email) throws IOException
{
    if(!addresses.removeElement(email)) return false;
    save();
return true;
}
private void save() throws IOException
{
    ObjectOutputStream out =
        new ObjectOutputStream(new FileOutputStream(filename));
out.writeObject(addresses);
    out.close();
}

```

In `init` we first call `super.init(config)` to leave the `ServletConfig` management to the superclass (`HttpServlet`), then we get the name of the address file from an *init parameter* (which is set up in the Web Server configuration). If the parameter is not available the Servlet throws a `javax.servlet.UnavailableException` (a subclass of `javax.servlet.ServletException`) which indicates that a Servlet is temporarily (if a duration is specified) or permanently (as in this case) unavailable. Finally, the `init` method deserializes the address file or creates an empty `Vector` if the address file does not exist yet. All exceptions that occur during the deserialization are transformed into `UnavailableExceptions`.

**API 2.1** Version 2.1 of the Servlet API offers a no-args `init` method which is called by `GenericServlet`'s `init(ServletConfig)` method. By using this new method you don't have to worry about passing the `ServletConfig` object to the superclass yourself. Note that even though code that uses the no-args `init` method can be compiled without problems using the JSDK 1.0 or 2.0 interface classes and run in a 1.0 or 2.0 compliant web server, the initialization code *will never be executed* in such an environment.

### Client interaction

The client interaction is handled by two of the standard `HttpServlet` methods, `doGet` and `doPost`.

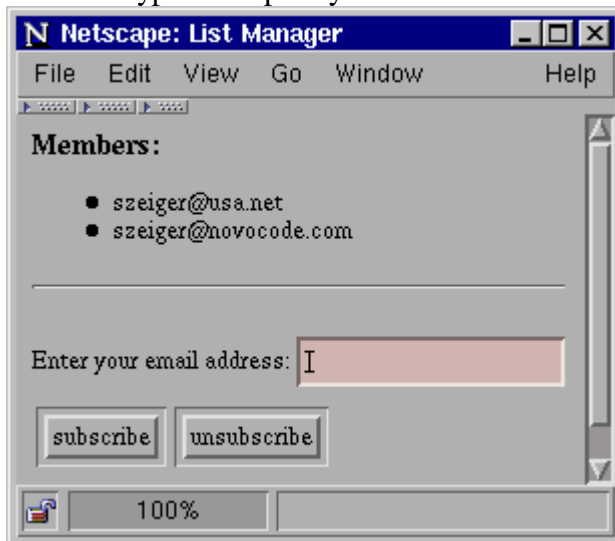


The `doGet` method replies to GET requests by sending an HTML page which contains the list of the currently subscribed addresses and the form that is used to subscribe or unsubscribe an address:

```
protected void doGet(HttpServletRequest req,
                    HttpServletResponse res)
    throws ServletException, IOException
{
    res.setContentType("text/html");
    res.setHeader("pragma", "no-cache");
    PrintWriter out = res.getWriter();
    out.print("<HTML><HEAD><TITLE>List Manager</TITLE></HEAD>");
    out.print("<BODY><H3>Members:</H3><UL>");
    for(int i=0; i<addresses.size(); i++)
        out.print("<LI>" + addresses.elementAt(i));
    out.print("</UL><HR><FORM METHOD=POST>");
    out.print("Enter your email address: <INPUT TYPE=TEXT NAME=email><BR>");
    out.print("<INPUT TYPE=SUBMIT NAME=action VALUE=subscribe>");
    out.print("<INPUT TYPE=SUBMIT NAME=action VALUE=unsubscribe>");
    out.print("</FORM></BODY></HTML>");
    out.close();
}
```

The response content type is again set to *text/html* and the response is marked as not cacheable to proxy servers and clients (because it is dynamically created) by setting an HTTP header "pragma: no-cache". The form asks the client to use the POST method for submitting form data.

Here is a typical output by this method:



## SERVLET LIFE CYCLE

The life cycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps.

1. If an instance of the servlet does not exist, the Web container
  - a. Loads the servlet class.
  - b. Creates an instance of the servlet class.
  - c. Initializes the servlet instance by calling the `init` method. Initialization is covered in [Initializing a Servlet](#).
2. Invokes the `service` method, passing a request and response object. Service methods are discussed in the section [Writing Service Methods](#).

If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's `destroy` method. Finalization is discussed in [Finalizing a Servlet](#).

### Handling Servlet Life-Cycle Events

You can monitor and react to events in a servlet's life cycle by defining listener objects whose methods get invoked when life cycle events occur. To use these listener objects, you must define the listener class and specify the listener class.

### Defining The Listener Class

You define a listener class as an implementation of a listener interface. Table 10-3 lists the events that can be monitored and the corresponding interface that must be implemented. When a listener method is invoked, it is passed an event that contains information appropriate to the event. For example, the methods in the `HttpSessionListener` interface are passed an `HttpSessionEvent`, which contains an `HttpSession`.

**Table 10-3 Servlet Life-Cycle Events**

Object	Event	Listener Interface and Event Class
Web context (See <a href="#">Accessing the Web Context</a> )	Initialization and destruction	<code>javax.servlet.ServletContextListener</code> and <code>ServletContextEvent</code>
	Attribute added, removed, or replaced	<code>javax.servlet.ServletContextAttributeListener</code> and <code>ServletContextAttributeEvent</code>
Session (See <a href="#">Maintaining Client State</a> )	Creation, invalidation, and timeout	<code>javax.servlet.http.HttpSessionListener</code> and <code>HttpSessionEvent</code>
	Attribute added, removed, or replaced	<code>javax.servlet.http.HttpSessionAttributeListener</code> and <code>HttpSessionBindingEvent</code>

The `listeners.ContextListener` class creates and removes the database helper and counter objects used in the Duke's Bookstore application. The methods retrieve the Web context object from `ServletContextEvent` and then store (and remove) the objects as servlet context attributes.

```
import database.BookDB;
import javax.servlet.*;
import util.Counter;
```

```

public final class ContextListener
    implements ServletContextListener {
    private ServletContext context = null;
    public void contextInitialized(ServletContextEvent event) {
        context = event.getServletContext();
        try {
            BookDB bookDB = new BookDB();
            context.setAttribute("bookDB", bookDB);
        } catch (Exception ex) {
            System.out.println(
                "Couldn't create database: " + ex.getMessage());
        }
        Counter counter = new Counter();
        context.setAttribute("hitCounter", counter);
        context.log("Created hitCounter" +
            counter.getCounter());
        counter = new Counter();
        context.setAttribute("orderCounter", counter);
        context.log("Created orderCounter" +
            counter.getCounter());
    }

    public void contextDestroyed(ServletContextEvent event) {
        context = event.getServletContext();
        BookDB bookDB = context.getAttribute(
            "bookDB");
        bookDB.remove();
        context.removeAttribute("bookDB");
        context.removeAttribute("hitCounter");
        context.removeAttribute("orderCounter");
    }
}

```

## PARAMETER DATA

### Reading Parameters

Here's a simple example that reads parameters named `param1`, `param2`, and `param3`, listing their values in a bulleted list. Note that, although you are required to specify response settings (content type, status line, other HTTP headings) before beginning to generate the content, there is no requirement that you read the request parameters at any particular time.

Also note you can easily make servlets that can handle both `GET` and `POST` data, simply by having its `doPost` method call `doGet` or by overriding `service` (which calls `doGet`, `doPost`, `doHead`, etc.). This is good standard practice, since it requires very little extra work and permits flexibility on the part of the client. If you're used to the traditional CGI approach where you read `POST` data via the standard input, you should note that there is a similar way with servlets by first calling `getReader` or `getInputStream` on the `HttpServletRequest`. This is a bad idea for regular parameters, but might be of use for uploaded files or `POST` data being sent by custom clients rather than via HTML forms. Note, however, that if you read the `POST` data in that manner, it might no longer be found by `getParameter`.

### ThreeParams.java

You can also [download the source](#) or [try it on-line](#). Note: also uses [ServletUtilities.java](#), shown earlier.

```
package hall;
```

```

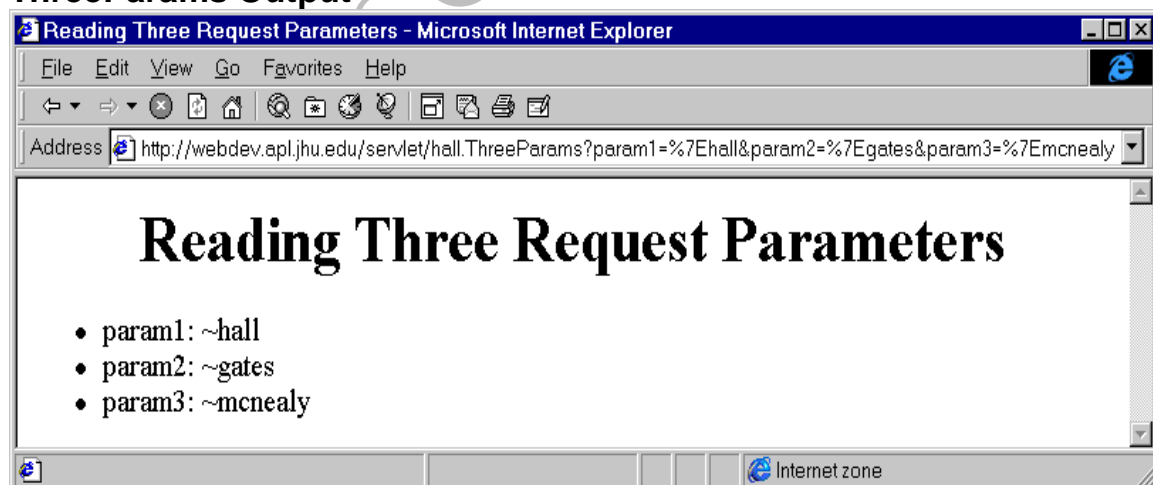
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class ThreeParams extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Reading Three Request Parameters";
        out.println(ServletUtilities.headWithTitle(title) +
            "<BODY>\n" +
            "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
            "<UL>\n" +
            "  <LI>param1: "
            + request.getParameter("param1") + "\n" +
            "  <LI>param2: "
            + request.getParameter("param2") + "\n" +
            "  <LI>param3: "
            + request.getParameter("param3") + "\n" +
            "</UL>\n" +
            "</BODY></HTML>");
    }

    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}

```

### ThreeParams Output



### Example: Listing All Form Data

Here's an example that looks up *all* the parameter names that were sent and puts them in a table. It highlights parameters that have zero values as well as ones that have multiple values. First, it looks up all the parameter names via the `getParameterNames` method of `HttpServletRequest`. This returns an `Enumeration`. Next, it loops down the

Enumeration in the standard manner, using `hasMoreElements` to determine when to stop and using `nextElement` to get each entry. Since `nextElement` returns an `Object`, it casts the result to a `String` and passes that to `getParameterValues`, yielding an array of `Strings`. If that array is one entry long and contains only an empty string, then the parameter had no values, and the servlet generates an italicized "No Value" entry. If the array is more than one entry long, then the parameter had multiple values, and they are displayed in a bulleted list. Otherwise the one main value is just placed into the table.

### ShowParameters.java

You can also [download the source](#) or [try it on-line](#). Note: also uses [ServletUtilities.java](#), shown earlier.

```
package hall;
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
```

```
public class ShowParameters extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String title = "Reading All Request Parameters";
        out.println(ServletUtilities.headWithTitle(title) +
            "<BODY BGCOLOR=\"#FDF5E6\">\n" +
            "<H1 ALIGN=CENTER>" + title + "</H1>\n" +
            "<TABLE BORDER=1 ALIGN=CENTER>\n" +
            "<TR BGCOLOR=\"#FFAD00\">\n" +
            "<TH>Parameter Name<TH>Parameter Value(s)");

        Enumeration paramNames = request.getParameterNames();
        while(paramNames.hasMoreElements()) {
            String paramName = (String)paramNames.nextElement();
            out.println("<TR><TD>" + paramName + "\n<TD>");
            String[] paramValues = request.getParameterValues(paramName);
            if (paramValues.length == 1) {
                String paramValue = paramValues[0];
                if (paramValue.length() == 0)
                    out.print("<I>No Value</I>");
                else
                    out.print(paramValue);
            } else {
                out.println("<UL>");
                for(int i=0; i<paramValues.length; i++) {
                    out.println("<LI>" + paramValues[i]);
                }
                out.println("</UL>");
            }
        }
        out.println("</TABLE>\n</BODY></HTML>");
    }
}
```

```

    }

    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        doGet(request, response);
    }
}

```

## Front End to ShowParameters

Here's an HTML form that sends a number of parameters to this servlet. Right click on the [source code link](#) to download the HTML. Left click on the link to try it out on-line. It uses POST to send the data (as should *all* forms that have PASSWORD entries), demonstrating the value of having servlets include both a doGet and a doPost. However, just for the sake of illustration, a [version using GET](#) can also be downloaded or tried out on-line.

### PostForm.html

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>
    <TITLE>A Sample FORM using POST</TITLE>
</HEAD>

<BODY BGCOLOR="#FDF5E6">
<H1 ALIGN="CENTER">A Sample FORM using POST</H1>

<FORM ACTION="/servlet/hall.ShowParameters"
    METHOD="POST">
    Item Number:
    <INPUT TYPE="TEXT" NAME="itemNum"><BR>
    Quantity:
    <INPUT TYPE="TEXT" NAME="quantity"><BR>
    Price Each:
    <INPUT TYPE="TEXT" NAME="price" VALUE="$"><BR>
    <HR>
    First Name:
    <INPUT TYPE="TEXT" NAME="firstName"><BR>
    Last Name:
    <INPUT TYPE="TEXT" NAME="lastName"><BR>
    Middle Initial:
    <INPUT TYPE="TEXT" NAME="initial"><BR>
    Shipping Address:
    <TEXTAREA NAME="address" ROWS=3 COLS=40></TEXTAREA><BR>
    Credit Card:<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
        VALUE="Visa">Visa<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
        VALUE="Master Card">Master Card<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
        VALUE="Amex">American Express<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
        VALUE="Discover">Discover<BR>
    <INPUT TYPE="RADIO" NAME="cardType"
        VALUE="Java SmartCard">Java SmartCard<BR>
    Credit Card Number:

```

```

<INPUT TYPE="PASSWORD" NAME="cardNum"><BR>
Repeat Credit Card Number:
<INPUT TYPE="PASSWORD" NAME="cardNum"><BR><BR>
<CENTER>
  <INPUT TYPE="SUBMIT" VALUE="Submit Order">
</CENTER>
</FORM>

</BODY>
</HTML>

```

**A Sample FORM using POST**

Item Number:

Quantity:

Price Each:

---

First Name:

Last Name:

Middle Initial:

Shipping Address:

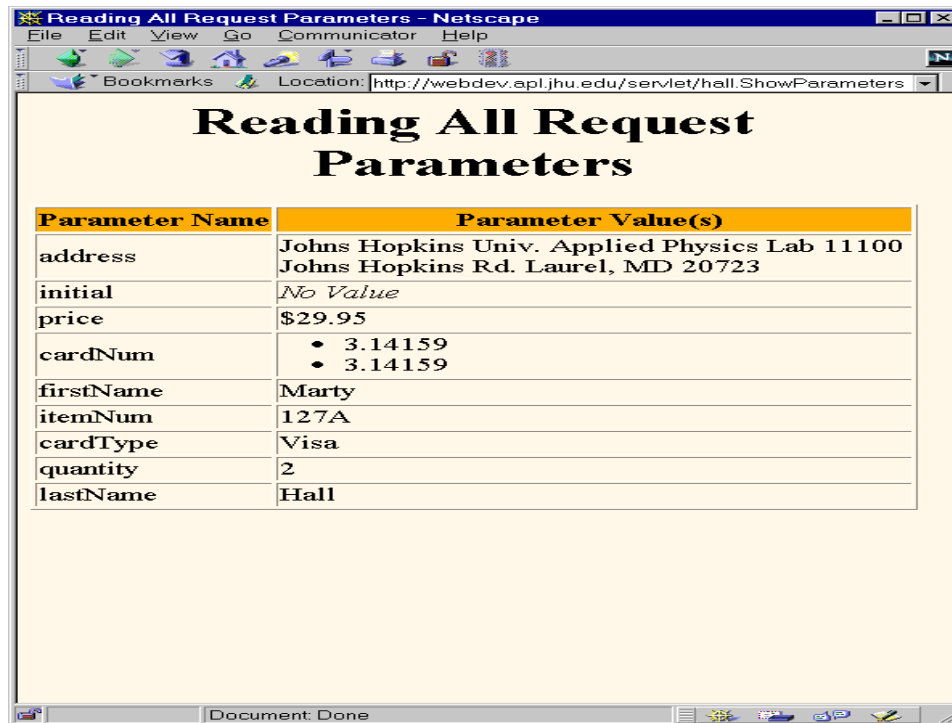
Credit Card:

☒ Visa  
☐ Master Card  
☐ American Express  
☐ Discover  
☐ Java SmartCard

Credit Card Number:

Repeat Credit Card Number:

## Submission Result



Parameter Name	Parameter Value(s)
address	Johns Hopkins Univ. Applied Physics Lab 11100 Johns Hopkins Rd. Laurel, MD 20723
initial	No Value
price	\$29.95
cardNum	• 3.14159 • 3.14159
firstName	Marty
itemNum	127A
cardType	Visa
quantity	2
lastName	Hall

## SESSIONS

### Session Tracking

This section shows how to

- use Session Tracking capabilities

Session Tracking allows a Servlet to associate a request with a user. A session can extend across requests and connections of the [stateless HTTP](#). Sessions can be maintained in two ways:

1. By using *Cookies*. A Cookie is a string (in this case that string is the *session ID*) which is sent to a client to start a session. If the client wants to continue the session it sends back the Cookie with subsequent requests. This is the most common way to implement session tracking.
2. By rewriting URLs. All links and redirections which are created by a Servlet have to be encoded to include the session ID. This is a less elegant solution (both, for Servlet implementors and users) because the session cannot be maintained by requesting a well-known URL oder selecting a URL which was created in a different (or no) session. It also does not allow the use of static pages. All HTML pages which are sent within a session have to be created dynamically.

Our next Servlet manages a virtual shopping cart. Users can add various items to their shopping cart via HTML forms. The shopping cart contents are stored on the server and each user gets his own shopping cart which is selected automatically whenever he makes a request to the Servlet.

In the simplified version that we implement in class `ShoppingCartServlet` there are only two kinds of items, named *FOO* and *BAR*. By pressing a button in an HTML form a single *FOO* or *BAR* item can be put into the shopping cart. There's another button to see



the current contents of the shopping cart and a button to order the selected items, thus clearing the shopping cart.

The first version of the Servlet, called `ShoppingCartServlet`, which works with Cookie-style sessions only, consists of the two standard methods, `doGet` and `doPost`:

- A form with the buttons is created by the Servlet's `doGet` method.
- ```
7: protected void doGet(HttpServletRequest req, HttpServletResponse res)
8:     throws ServletException, IOException
9: {
10:    res.setContentType("text/html");
11:    PrintWriter out = res.getWriter();
12:    out.print("<HTML><HEAD><TITLE>Online Shop</TITLE>" +
13:            "</HEAD><BODY><FORM METHOD=POST>" +
14:            "<INPUT TYPE=SUBMIT NAME=foo VALUE=" +
15:            "\"Put a FOO into the shopping cart\">" +
16:            "<INPUT TYPE=SUBMIT NAME=bar VALUE=" +
17:            "\"Put a BAR into the shopping cart\">" +
18:            "<INPUT TYPE=SUBMIT NAME=see VALUE=" +
19:            "\"See the shopping cart contents\">" +
20:            "<INPUT TYPE=SUBMIT NAME=buy VALUE=" +
21:            "\"Buy the shopping cart contents\">" +
22:            "</FORM></BODY></HTML>");
23:    out.close();
24: }
```

First we get the `HttpSession` object which is associated with the request by calling `req.getSession`. The argument `true` forces the creation of a new session if the request doesn't contain a valid session key.

If the session is indeed new (determined by calling `HttpSession`'s `isNew()` method) we add some custom data to the session: Two counters, one for the *FOOs* and one for the *BARs* in the shopping cart. The session object can be used like a `Dictionary`. That means we can only add *Objects*, not instances of primitive types like *int*. We could use an instance of `java.lang.Integer` for each counter, but these objects are immutable which makes incrementing inefficient and difficult to implement. Instead we use an *array of int* (`int[]`) with only one element as a mutable wrapper object. The element is initialized to 0.

Next we retrieve the values for "foo" and "bar" from the session, no matter if they were just added or carried over from a previous request.

In the `ListManagerServlet` both buttons had the same name but different values so we could use `getParameter` to retrieve the value from the request and then do a string compare to the possible values. This time we use a different approach which can be implemented more efficiently. All buttons have different names and we can find out which button was used to submit the form by checking which name has a non-null value.

A new *FOO* or *BAR* item can be put into the shopping cart by simply incrementing the counter in the array. Note that the array does not need to be put back into the session because it has not changed itself, only the contents have been modified.

When the user chooses to buy the contents of the shopping cart we call `session.invalidate()` to delete the session on the server side and tell the client to remove the session ID Cookie. The session data is lost and when a new *POST* request is made to the Servlet, a new session will be created.

The rest of the `doPost` method is basically the same as in the `ListManagerServlet`.

## URL REWRITING

To make the Servlet usable with URL rewriting (for clients without Cookie support or with Cookie support turned off) we have to make some modifications.

The formerly static "Online Shop" page which is created by `doGet` needs to be modified to include an *ACTION* URL which contains an encoded session ID in the HTML form. This is done with the `encodeUrl` method of `HttpServletResponse`. We also need to call `req.getSession` to keep the session alive. The additional code to check for a new session can be avoided by using `getSession(false)`. If there is no session or an invalid session we do not force the creation of a new session. This is deferred until the `doPost` method is called. Finally, the response has to be marked as not cachable by calling `res.setHeader("pragma", "no-cache")`, as usual.

These changes lead us to the following revised implementation of `doGet`:

```
protected void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
{
    HttpSession session = req.getSession(false);
    res.setContentType("text/html");
    res.setHeader("pragma", "no-cache");
    PrintWriter out = res.getWriter();
    out.print("<HTML><HEAD><TITLE>Online Shop</TITLE>"+
        "</HEAD><BODY><FORM METHOD=POST ACTION="+
    out.print(res.encodeUrl(req.getRequestURI()));
    out.print("><INPUT TYPE=SUBMIT NAME=foo VALUE="+
        "\"Put a FOO into the shopping cart\">"+
        "<INPUT TYPE=SUBMIT NAME=bar VALUE="+
        "\"Put a BAR into the shopping cart\">"+
        "<INPUT TYPE=SUBMIT NAME=see VALUE="+
        "\"See the shopping cart contents\">"+
        "<INPUT TYPE=SUBMIT NAME=buy VALUE="+
        "\"Buy the shopping cart contents\">"+
        "</FORM></BODY></HTML>");
    out.close();
}
```

## COOKIES

We've already used Cookies indirectly through *Sessions*. While session data is stored on the server side and only an index to the server-side data (the session ID) is transmitted to the client, Cookies can also be used directly to store data on the client side.

The Servlet API provides the class `javax.servlet.http.Cookie` for a convenient object-oriented representation of Cookies so you don't need to compose and decompose `Cookie` and `Set-Cookie` HTTP headers yourself.

Even if you don't care where the data is stored it is sometimes useful to manipulate Cookies directly via the `Cookie` class to get more control over Cookie parameters like *Domain* and *Path*, e.g. to share Cookies between different Servlets or even servers.

**Example.** Imagine an authentication Servlet which receives a username and password from a login form via `doPost` and verifies them against a central authentication database. It then computes an authentication string (e.g. a Base64-encoded "user:password" combination, as used by *HTTP Basic Authentication*). This string is now put into a Cookie and sent back to the client:

```
Cookie authCookie = new Cookie("xyz-Auth", credentials);
authCookie.setVersion(1);
authCookie.setDomain(".xyz.com");
res.addCookie(authCookie);
```

The Cookie's domain is set to ".xyz.com" so it will be sent to all hosts in domain "xyz.com" like "a.xyz.com" and "foo.xyz.com" (but not "c.d.xyz.com"). Note that the *Domain* attribute is supported by RFC2109-style Cookies (version 1) but not by old Netscape Cookies (version 0, the default for newly created `Cookie` objects).

All Web Servers on hosts in xyz.com are running an instance of another Servlet which serves protected data after verifying the authentication credentials:

```
boolean verified = false;
Cookie[] cookies = req.getCookies();

for(int i=0; i<cookies.length; i++)
{
    String n = cookies[i].getName(),
          d = cookies[i].getDomain();
    if(n != null && n.equals("xyz-Auth") &&
       d != null && d.equals(".xyz.com"))
    {
        String credentials = cookies[i].getValue();
        verified = verifyCredentials(credentials);
        break;
    }
}
if(!verified)
{
    res.sendRedirect(...);
    return;
}
```

The credentials are retrieved from the Cookie and verified by the authentication database. If the credentials are invalid or missing the client is redirected to the login page on the authentication server, otherwise the protected content is returned.

## OTHER SERVLETS CAPABILITIES

### Servlet Interface and Life Cycle

Let's implement our first servlet. A servlet is a Java class that implements the `Servlet` interface. This interface has three methods that define the servlet's life cycle:

- `public void init(ServletConfig config) throws ServletException`  
This method is called once when the servlet is loaded into the servlet engine, before the servlet is asked to process its first request.
- `public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException`  
This method is called to process a request. It can be called zero, one or many times until the servlet is unloaded. Multiple threads (one per request) can execute this method in parallel so it must be thread safe.
- `public void destroy()`  
This method is called once just before the servlet is unloaded and taken out of service.

The `init` method has a `ServletConfig` attribute. The servlet can read its initialization arguments through the `ServletConfig` object. How the initialization arguments are set is servlet engine dependent but they are usually defined in a configuration file.

A typical example of an initialization argument is a database identifier. A servlet can read this argument from the `ServletConfig` at initialization and then use it later to open a connection to the database during processing of a request:

...

```
private String databaseURL;
```

```
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    databaseURL = config.getInitParameter("database");
}
```

The Servlet API is structured to make servlets that use a different protocol than HTTP possible. The `javax.servlet` package contains interfaces and classes intended to be protocol independent and the `javax.servlet.http` package contains HTTP specific interfaces and classes. Since this is just an introduction to servlets I will ignore this distinction here and focus on HTTP servlets. Our first servlet, named `ReqInfoServlet`, will therefore extend a class named `HttpServlet`. `HttpServlet` is part of the JSDK and implements the `Servlet` interface plus a number of convenience methods. We define our class like this:

```
import javax.servlet.*;
import javax.servlet.http.*;
```

```
public class ReqInfoServlet extends HttpServlet {
```

...

```
}
```

An important set of methods in `HttpServlet` are the ones that specialize the `service` method in the `Servlet` interface. The implementation of `service` in `HttpServlet` looks at the type of request it's asked to handle (GET, POST, HEAD, etc.) and calls a specific method for each type. This way the servlet developer is relieved from handling the details about obscure requests like HEAD, TRACE and OPTIONS and can focus on taking care of the

more common request types, i.e. GET and POST. In this first example we will only implement the doGet method.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
```

```
...
```

```
}
```

### Request and Response Objects

The doGet method has two interesting parameters: HttpServletRequest and HttpServletResponse. These two objects give you full access to all information about the request and let you control the output sent to the client as the response to the request.

With CGI you read environment variables and stdin to get information about the request, but the names of the environment variables may vary between implementations and some are not provided by all Web servers. The HttpServletRequest object provides the same information as the CGI environment variables, plus more, in a standardized way. It also provides methods for extracting HTTP parameters from the query string or the request body depending on the type of request (GET or POST). As a servlet developer you access parameters the same way for both types of requests. Other methods give you access to all request headers and help you parse date and cookie headers.

Instead of writing the response to stdout as you do with CGI, you get an OutputStream or a PrintWriter from the HttpServletResponse. The OutputStream is intended for binary data, such as a GIF or JPEG image, and the PrintWriter for text output. You can also set all response headers and the status code, without having to rely on special Web server CGI configurations such as Non Parsed Headers (NPH). This makes your servlet easier to install.

Let's implement the body of our doGet method and see how we can use these methods. We will read most of the information we can get from the HttpServletRequest (saving some methods for the next example) and send the values as the response to the request.

```
protected void doGet(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
```

```
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    // Print the HTML header
    out.println("<HTML><HEAD><TITLE>");
    out.println("Request info");
    out.println("</TITLE></HEAD>");

    // Print the HTML body
    out.println("<BODY><H1>Request info</H1><PRE>");
    out.println("getCharacterEncoding: " + request.getCharacterEncoding());
    out.println("getContentLength: " + request.getContentLength());
    out.println("getContentType: " + request.getContentType());
    out.println("getProtocol: " + request.getProtocol());
    out.println("getRemoteAddr: " + request.getRemoteAddr());
    out.println("getRemoteHost: " + request.getRemoteHost());
```

```

out.println("getScheme: " + request.getScheme());
out.println("getServerName: " + request.getServerName());
out.println("getServerPort: " + request.getServerPort());
out.println("getAuthType: " + request.getAuthType());
out.println("getMethod: " + request.getMethod());
out.println("getPathInfo: " + request.getPathInfo());
out.println("getPathTranslated: " + request.getPathTranslated());
out.println("getQueryString: " + request.getQueryString());
out.println("getRemoteUser: " + request.getRemoteUser());
out.println("getRequestURI: " + request.getRequestURI());
out.println("getServletPath: " + request.getServletPath());

out.println();
out.println("Parameters:");
Enumeration paramNames = request.getParameterNames();
while (paramNames.hasMoreElements()) {
    String name = (String) paramNames.nextElement();
    String[] values = request.getParameterValues(name);
    out.println("    " + name + ":" );
    for (int i = 0; i < values.length; i++) {
        out.println("        " + values[i]);
    }
}

out.println();
out.println("Request headers:");
Enumeration headerNames = request.getHeaderNames();
while (headerNames.hasMoreElements()) {
    String name = (String) headerNames.nextElement();
    String value = request.getHeader(name);
    out.println("    " + name + " : " + value);
}

out.println();
out.println("Cookies:");
Cookie[] cookies = request.getCookies();
for (int i = 0; i < cookies.length; i++) {
    String name = cookies[i].getName();
    String value = cookies[i].getValue();
    out.println("    " + name + " : " + value);
}

// Print the HTML footer
out.println("</PRE></BODY></HTML>");
out.close();
}

```

The `doGet` method above uses most of the methods in `HttpServletRequest` that provide information about the request. You can read all about them in the Servlet API documentation so here we'll just look at the most interesting ones.

`getParameterNames` and `getParameterValues` help you access HTTP parameters no matter if the servlet was requested with the GET or the POST method. `getParameterValues` returns a `String` array because an HTTP parameter may have multiple values. For instance, if you request the servlet with a URL like `http://company.com/servlet/ReqInfoServlet?foo=bar&foo=baz` you'll see that the

`foo` parameter has two values: `bar` and `baz`. The same is true if you use the same name for more than one HTML FORM element and use the POST method in the ACTION tag. If you're sure that an HTTP parameter only can have one value you can use the `getParameter` method instead of `getParameterValues`. It returns a single String and if there are multiple values it returns the first value received with the request.

You have access to all HTTP request headers with the `getHeaderNames` and `getHeader` methods. `getHeader` returns the String value of the header. If you know that the header has a date value or an integer value you can get help converting the header to an appropriate format. `getDateHeader` returns a date as the number of milliseconds since January 1, 1970, 00:00:00 GMT. This is the standard numeric representation of a timestamp in Java and you can use it to construct a Date object for further manipulation. `getIntHeader` returns the header value as an `int`.

`getCookies` parses the Cookie header and returns all cookies as an array of Cookie objects. To add a cookie to a response the `HttpServletResponse` class provides an `addCookie` method that takes a Cookie object as its argument. This saves you from dealing with the format for different versions of cookie header strings.

If you compile the `ReqInfoServlet` and install it in your servlet engine you can now invoke it through a browser with a URL like

`http://company.com/servlet/ReqInfoServlet/foo/bar?fee=baz`. If everything goes as planned you will see something like this in your browser:

## DATA STORAGE

One of the blessings of the JVM is that there are many “pure” databases available for it; Sesame (RDF) and Apache Derby (relational) come to mind. Alas, these databases need to save their files somewhere. This becomes a challenge with Servlets: Where should one put these files? Pre-packaged, read-only, files usually reside in `SERVER/webapps/app/WEB-INF/`, but mutable files? Putting them in `WEB-INF` makes upgrading a web application more difficult. I can easily imagine myself accidentally removing such data during an upgrade. The best solution, that I was able to come up with, is:

- By default, the data directory of a web application “app” is `SERVER/webapps/app.data/WEB-INF/` (`WEB-INF` is necessary to protect against serving secret files).
- This can optionally be configured via a file `SERVER/webapps/app.properties`. There, one can specify the servlet data directory via a relative or an absolute path. The former case is useful if you want this directory to be a sibling of `SERVER/webapps`. The latter case is useful for putting this directory anywhere on a system.

I do realize that there are security issues with this scheme: While it worked for me on Tomcat, I would expect other servlet containers to prevent servlets from writing files in this manner. Other options are:

- Make the data directory a servlet parameter in `web.xml`: Complicates upgrades.
- Specify a directory via the Java preference API: Prevents installing the same web application multiple times on the same system.
- Storing the data in a temporary (as in `File.createTempFile()`) directory: Not a solution for long-term persistence.

## SERVLETS AND CONCURRENCY

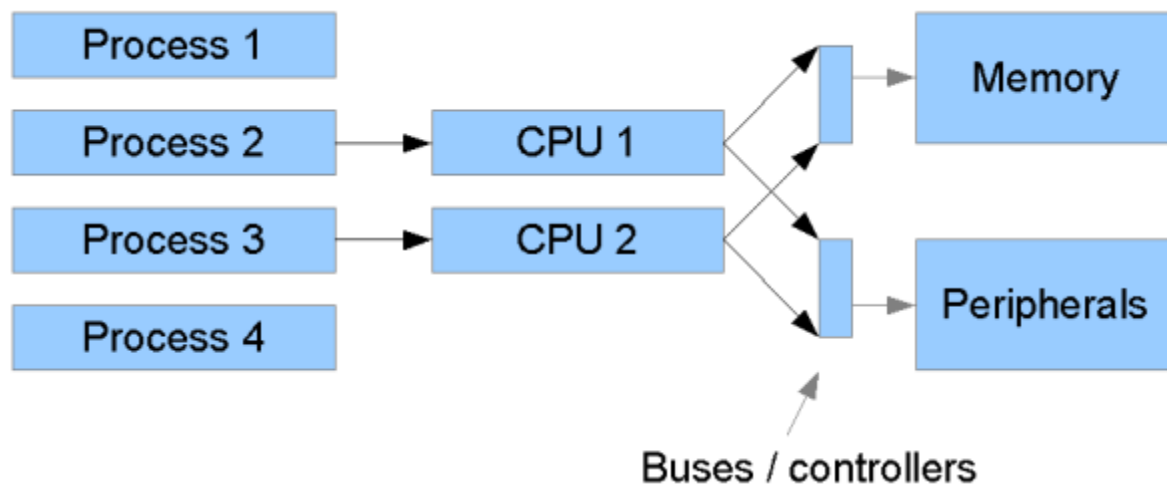
## CONCURRENCY IN WEB SERVERS

Essentially, multi-processor systems are difficult to program because:

- they involve multiple processes competing for shared resources;
- these resources perform differently. (For example, a hard disk is much slower to access than RAM.)

A highly concurrent application needs to make sure that each process is "in the right place at the right time" to take advantage of what resources are available at any given moment.

A typical situation is illustrated in Figure 1. The architectural details have been heavily simplified in this diagram, but it shows four processes "running" on a machine with two CPUs. Since there are only two CPUs, only two processes can actually be running simultaneously at any one time; the operating system must periodically allocate CPU time to different processes– or "swap them in and out" of the CPUs– so that over time all four make progress.



Additionally– and usually more significantly– each running process, via the CPU, may access main memory, shared between the two CPUs, and other shared peripherals such as hard disks. These "external" resources are much slower to access than registers and cached instructions "internal" to the CPU. Additionally, only one value can physically be written to a memory address at once, and only one part of a hard disk can physically be accessed at once. These factors pose two main problems:

- How to avoid "wasting" CPUs when they need to wait for a response from one of the slower resources (such as memory or hard disk);
- What to do when two CPUs access want to access a shared resource simultaneously.

The ideal solution to the first problem is that while one process is waiting on a slow resource, it is "swapped out" of the CPU and another process that requires the CPU runs in the meantime. Moving towards this ideal generally falls under the heading of **concurrency**. In the second case, what physically happens (e.g. which processor "gets in there first" when the two CPUs simultaneously want to write different values to a memory location) is determined by the hardware and can be unpredictable. So when data is being shared in this way, we need to take special steps in software to manage simultaneous access and give predictable results. This second issue is generally the realm of **synchronization**.



## Concurrency on a single-processor machine

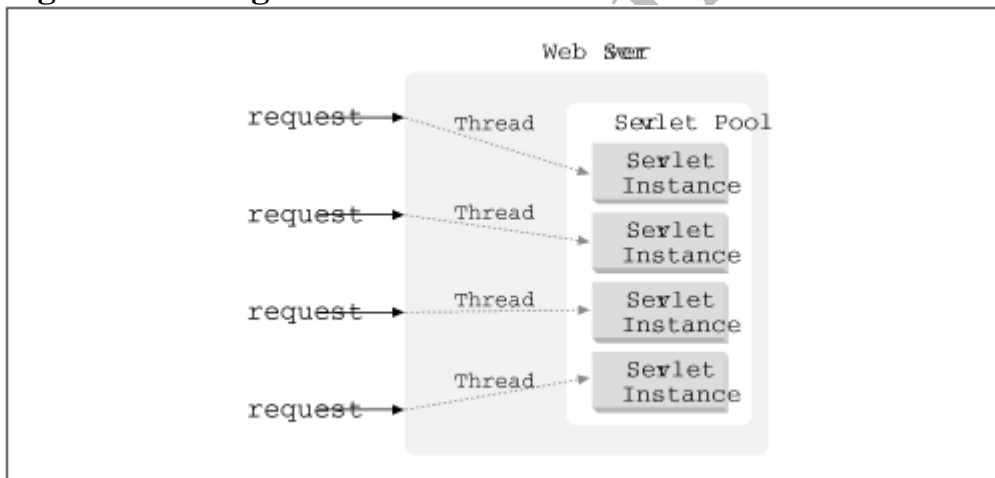
Concurrency is still often an issue on a single-processor machine because processes are usually swapped in and out of the available processor under *interrupt*. One process cannot necessarily predict when another one is going to "step in" and potentially change some data that it was in the middle of accessing. And the issue of not "wasting" the CPU still remains: if process 1 is waiting for data from the hard disk, and thus cannot usefully use the CPU, it would be useful to let process 2 use the CPU in the meantime

## Single-Thread Model

Although it is standard to have one servlet instance per registered servlet name, it is possible for a servlet to elect instead to have a pool of instances created for each of its names, all sharing the duty of handling requests. Such servlets indicate this desire by implementing the `javax.servlet.SingleThreadModel` interface. This is an empty, tag interface that defines no methods or variables and serves only to flag the servlet as wanting the alternate life cycle.

A server that loads a `SingleThreadModel` servlet must guarantee, according to the Servlet API documentation, "that no two threads will execute concurrently the service method of that servlet." To accomplish this, each thread uses a free servlet instance from the pool, as shown in Figure 3-4. Thus, any servlet implementing `SingleThreadModel` can be considered thread safe and isn't required to synchronize access to its instance variables.

**Figure The Single Thread Model**



Such a life cycle is pointless for a counter or other servlet application that requires central state maintenance. The life cycle can be useful, however, in avoiding synchronization while still performing efficient request handling.

For example, a servlet that connects to a database sometimes needs to perform several database commands atomically as part of a single transaction. Normally, this would require the servlet to synchronize around the database commands (letting it manage just one request at a time) or to manage a pool of database connections where it can "check out" and "check in" connections (letting it support multiple concurrent requests). By instead implementing `SingleThreadModel` and having one "connection" instance variable per servlet, a servlet can easily handle concurrent requests by letting its server manage the servlet instance pool (which doubles as a connection pool). The skeleton code is shown in Example 3-5.

## Example Handling database connections using SingleThreadModel

```
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SingleThreadConnection extends HttpServlet
    implements SingleThreadModel {

    Connection con = null;    // database connection, one per pooled
    servlet instance

    public void init(ServletConfig config) throws ServletException {
        super.init(config);

        // Establish the connection for this instance
        con = establishConnection();
        con.setAutoCommit(false);
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();

        // Use the connection uniquely assigned to this instance
        Statement stmt = con.createStatement();

        // Update the database any number of ways

        // Commit the transaction
        con.commit();
    }

    public void destroy() {
        if (con != null) con.close();
    }

    private Connection establishConnection() {
        // Not implemented. See .
    }
}
```

## Background Processing

Servlets can do more than simply persist between accesses. They can also *execute* between accesses. Any thread started by a servlet can continue executing even after the response has been sent. This ability proves most useful for long-running tasks whose incremental results should be made available to multiple clients. A background thread started in `init()` performs continuous work while request-handling threads display the current status with `doGet()`. It's a similar technique to that used in animation applets, where one thread changes the picture and another paints the display.

Example 3-6 shows a servlet that searches for prime numbers above one quadrillion. It starts with such a large number to make the calculation slow enough to adequately demonstrate caching effects--something we need for the next section. The algorithm it

uses couldn't be simpler: it selects odd-numbered candidates and attempts to divide them by every odd integer between 3 and their square root. If none of the integers evenly divides the candidate, it is declared prime.

### Example On the hunt for primes

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class PrimeSearcher extends HttpServlet implements Runnable {

    long lastprime = 0; // last prime found
    Date lastprimeModified = new Date(); // when it was found
    Thread searcher; // background search thread

    public void init(ServletConfig config) throws ServletException {
        super.init(config); // always!
        searcher = new Thread(this);
        searcher.setPriority(Thread.MIN_PRIORITY); // be a good citizen
        searcher.start();
    }

    public void run() {
        // QTTTBBBMMMTTTOOO
        long candidate = 1000000000000001L; // one quadrillion and one

        // Begin loop searching for primes
        while (true) { // search forever
            if (isPrime(candidate)) {
                lastprime = candidate; // new prime
                lastprimeModified = new Date(); // new "prime time"
            }
            candidate += 2; // evens aren't prime

            // Between candidates take a 0.2 second break.
            // Another way to be a good citizen with system resources.
            try {
                searcher.sleep(200);
            }
            catch (InterruptedException ignored) { }
        }
    }

    private static boolean isPrime(long candidate) {
        // Try dividing the number by all odd numbers between 3 and its
        sqrt
        double sqrt = Math.sqrt(candidate);
        for (long i = 3; i <= sqrt; i += 2) {
            if (candidate % i == 0) return false; // found a factor
        }

        // Wasn't evenly divisible, so it's prime
        return true;
    }
}
```

```

public void doGet(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    res.setContentType("text/plain");
    PrintWriter out = res.getWriter();
    if (lastprime == 0) {
        out.println("Still searching for first prime...");
    }
    else {
        out.println("The last prime discovered was " + lastprime);
        out.println(" at " + lastprimeModified);
    }
}

public void destroy() {
    searcher.stop();
}
}

```

The searcher thread begins its search in the `init()` method. Its latest find is saved in `lastprime`, along with the time it was found in `lastprimeModified`. Each time a client accesses the servlet, the `doGet()` method reports the largest prime found so far and the time it was found. The searcher runs independently of client accesses; even if no one accesses the servlet it continues to find primes silently. If several clients access the servlet at the same time, they all see the same current status.

Notice that the `destroy()` method stops the searcher thread.[6] This is very important! If a servlet does not stop its background threads, they continue to run until the virtual machine exits. Even when a servlet is reloaded (either explicitly or because its class file changed), its threads won't be stopped. Instead, it's likely that the new servlet will create extra copies of the background threads. And, at least with the Java Web Server, even explicitly restarting the web server service doesn't stop background threads because the Java Web Server virtual machine continues its execution.

### ***Last Modified Times***

Most web servers, when they return a document, include as part of their response a `Last-Modified` header. An example `Last-Modified` header value might be:

Tue, 06-May-98 15:41:02 GMT

This header tells the client the time the page was last changed. That information alone is only marginally interesting, but it proves useful when a browser reloads a page.

Most web browsers, when they reload a page, include in their request an `If-Modified-Since` header. Its structure is identical to the `Last-Modified` header:

Tue, 06-May-98 15:41:02 GMT

This header tells the server the `Last-Modified` time of the page when it was last downloaded by the browser. The server can read this header and determine if the file has changed since the given time. Here's a `getLastModified()` method for our `PrimeSearcher` example that returns when the last prime was found.

```

public long getLastModified(HttpServletRequest req) {
    return lastprimeModified.getTime() / 1000 * 1000;
}

```

milliseconds later than the time given by `If-Modified-Since`. For example, let's assume `PrimeSearcher` found a prime exactly 869127442359 milliseconds since the beginning of the Disco Decade. This fact is told to the browser, but only to the nearest second:

Thu, 17-Jul-97 09:17:22 GMT

Now let's assume that the user reloads the page and the browser tells the server, via the `If-Modified-Since` header, the time it believes its cached page was last modified:

Thu, 17-Jul-97 09:17:22 GMT

Some servers have been known to receive this time, convert it to exactly 869127442000 milliseconds, find that this time is 359 milliseconds earlier than the time returned by `getLastModified()`, and falsely assume that the servlet's content has changed. This is s

## CASE STUDY

### Deploying Servlets on Smart Cards: Portable Web Servers with Java Card 3.0



#### **Background**

In previous versions of the Java Card specification, Java developers were significantly limited in the breadth and depth of the applications that could be created for Java Card 2 devices, such as:

- A very strict subset of Java APIs, with only JDK 1.0 features
- No support for `java.lang.String` objects
- No threads
- No garbage collection
- No support for networking
- No native support for `.class` files
- No dynamic class loading
- 1-D Arrays
- 8-bit CPUs
- 8kB of RAM

Despite these limitations, Java Card technology became the most widely deployed Java platform in the world with over 3.5 billion deployments. Java Card 3.0 technology has several significant enhancements over the Java Card 2 platform, as explained in the following sections.

#### **Comparing JavaCard 2 to JavaCard 3**

As you can see from the sequence diagrams in Figures 1 and 2 located below, creating a complete application that involved using Java Card 2 technology could be tedious.



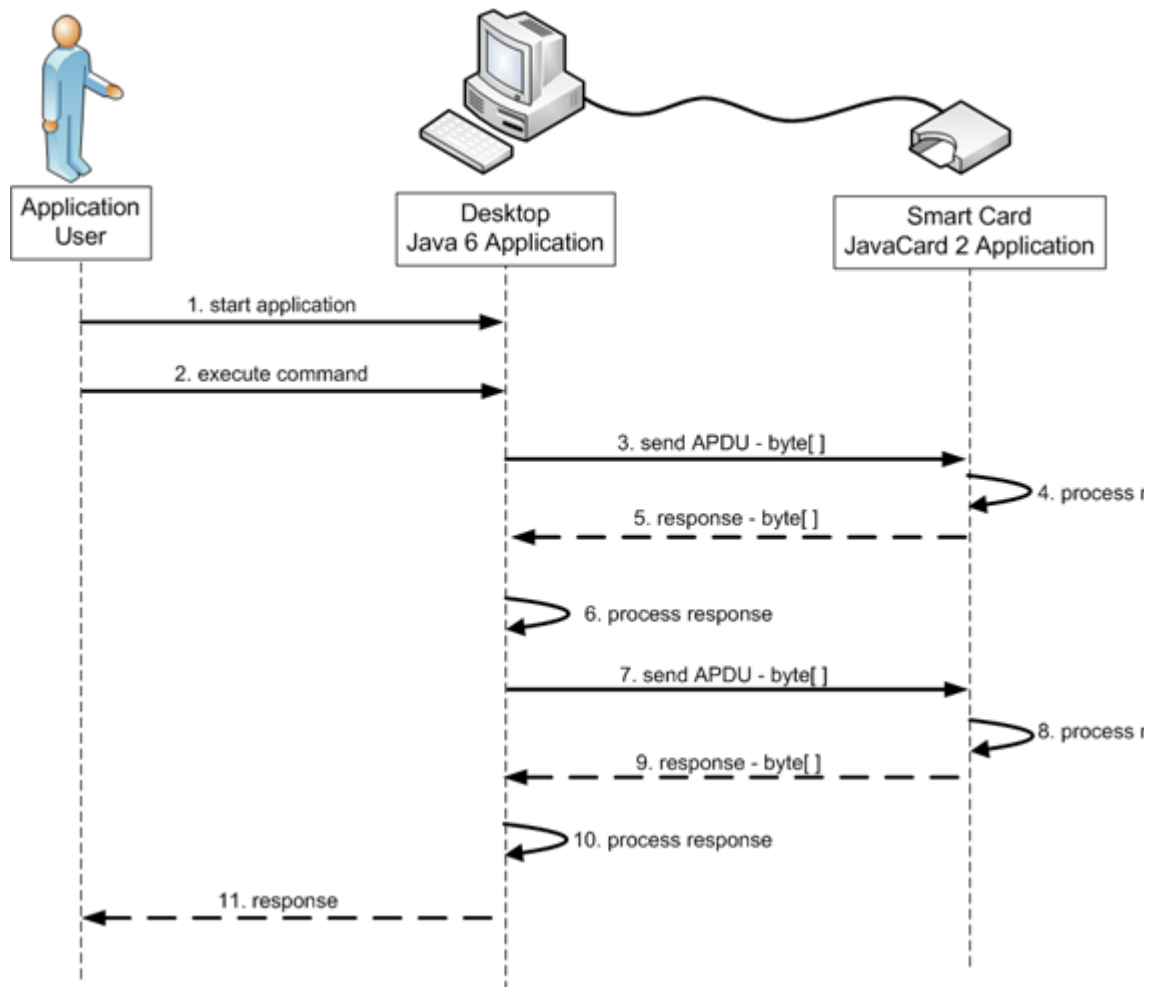


Figure 1. A Sequence Diagram Depicting the Interactions Between a Desktop Application and a Java Card 2 Application

Note

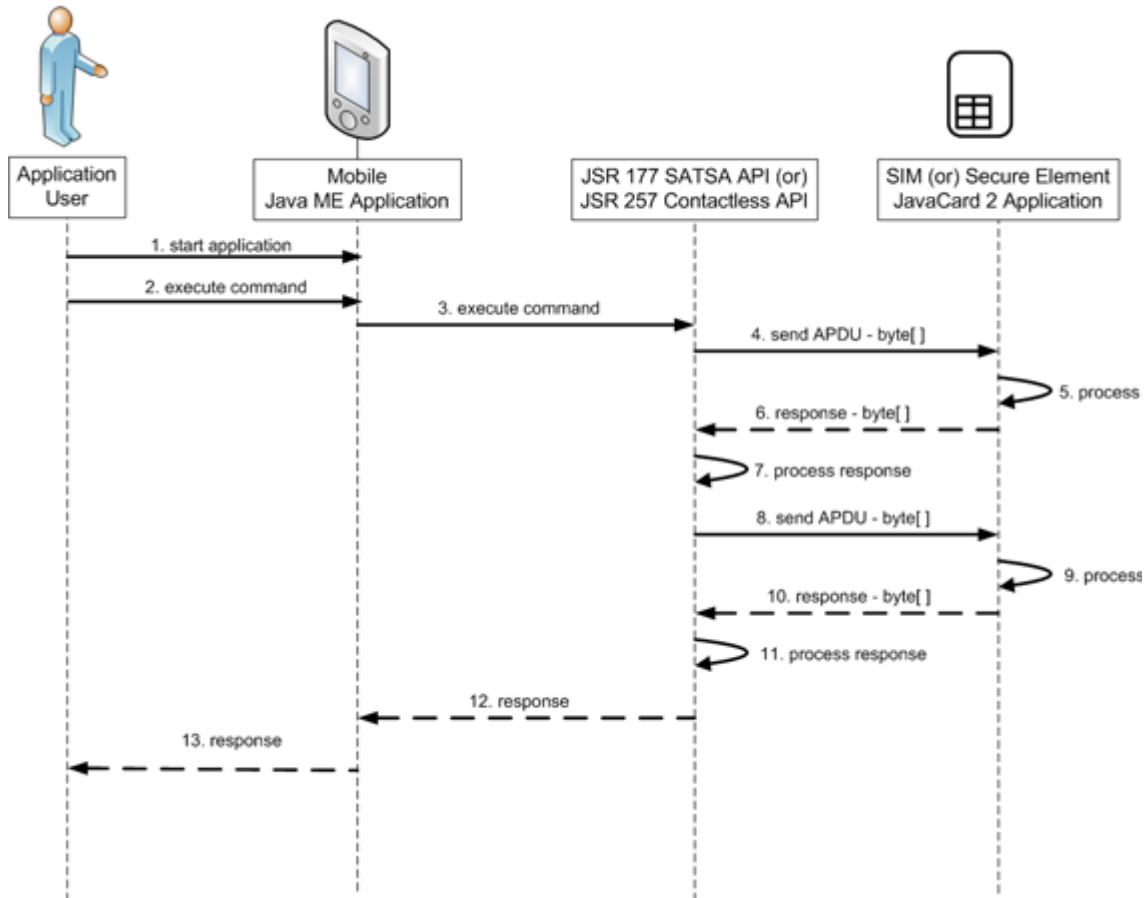


Figure 2. A Sequence Diagram Depicting the Interactions Between a Mobile Application and a Java Card 2 Application

As you can see from the preceding figures, both the desktop and mobile applications need to eventually use the APDU (application protocol data unit) protocol to communicate with applications (also known as applets) located on the Java Card 2 smart card. In some cases, however, the API restrictions -- combined with having to use byte arrays for all interaction with the smart card -- were major deterrents to attracting enterprise Java developers to the Java Card platform.

However, the Java Card 3 platform provides significant improvements to its predecessor, and those changes should appeal not only to enterprise and desktop developers, but to mobile developers as well. The Java Card 3 platform includes support for the following features:

- An API based upon a CLDC subset, with Java 6 features

- Annotations

- byte, short, int, long, char and `java.lang.String`

- Threads

- Servlets

- Garbage collection of objects

- TCP/IP networking

- HTTP and HTTPS

- .class files

.war deployment files  
Dynamic class loading  
2-D Arrays  
32-bit CPUs  
24kB of RAM

Figure 3, shown below is a sequence diagram showing the steps and actions involved in creating a complete application using the Java Card 3 platform.

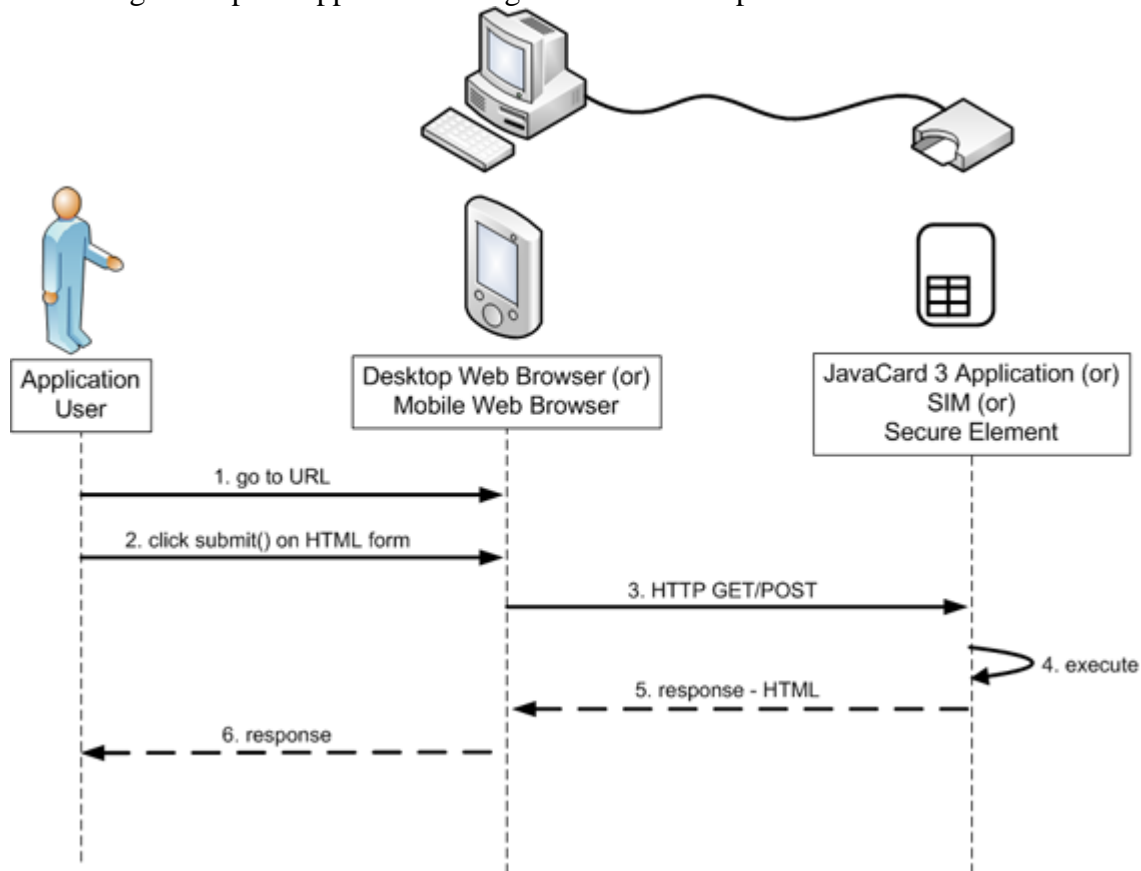


Figure 3. A Sequence Diagram Depicting the Interactions Between a Mobile or Desktop Application and a Java Card 3 Application

With Java Card 3 technology, smart card application development has instantly become a lot easier, and has now taken a significant leap-ahead in capability so that the following scenarios are possible:

Developers no longer need to create individual client applications to access the data and resources on the smart card. The only client interface needed is an ordinary web browser.

Smart Card applications are now fully functioning TCP-based servers. These server applications are Java servlets, and they have a full HTTP stack to allow them to process GET requests, POST requests, headers, cookies, sessions, and so on.

You can secure the data between the client (the browser) and the server (the smart card) by using the industry-standard SSL (secure sockets layer).



## Getting Started the Java Card Connected Development Kit and the Persistence Example

Did you know that Java Card applications have access to two heap spaces? One of them is volatile, which means that it acts like RAM, so all information stored in that heap location is lost after the system is reset (that is, when the card is removed and inserted). The other heap area is non-volatile, so data stored in that location is truly persistent, even after a system reset.

To get clear understanding of how to store data on the card in the volatile heap allocation, let's take a look at the Persistence example from the Java Card Connected Development Kit 3.0.2. Table 1, shown below, provides a detailed breakdown of the files contained within

the %folder\_root%\JCDK3.0.2\_ConnectedEdition\samples\web\Persistence folder.

**Table 1. Description of Files and Folders Within the Persistence Example**

| Folder/File Name            | Description                                                                                                                                                                                                              |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| /build.xml                  | This is the primary ant build xml file. The project name, base directory, and import file are specified in this file.                                                                                                    |
| /build/                     | This is the parent folder that contains the build artifacts that will be automatically created by ant. These artifacts include the intermediary /WEB-INF/classes folder, which will be packaged into the final .war file |
| /config/                    | This is the parent folder that contains the configuration files.                                                                                                                                                         |
| /config/build-impl.xml      | This is an autogenerated ant build file that is imported from the primary build.xml file. It is not recommended for developers to modify this file.                                                                      |
| /config/genfiles.properties | This is an autogenerated file used by NetBeans. It is not recommended for developers to modify this file.                                                                                                                |
| /config/project.properties  | This is the main property file that sets several configurable properties such as the main servlet, the main URL, as well as the web context path.                                                                        |
| /config/project.xml         | This the NetBeans project XML file.                                                                                                                                                                                      |
| /dist                       | This is the parent folder containing the redistributable items / artifacts after the build process is complete.                                                                                                          |
| /dist/Persistence.signature | This is a binary file that contains a digital signature.                                                                                                                                                                 |
| /dist/Persistence.war       | This is the deployable .war file that will be loaded on the Java Card.                                                                                                                                                   |

|                                 |                                                                                                                                                                                        |
|---------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| /html/                          | This is the parent folder containing the index.html as well as the accompanying images.                                                                                                |
| /html/index.html                | This is a simple placeholder html page that redirects to the "index" path of the JavaCard WAR application.                                                                             |
| /html/images/                   | This is the folder that contains the images referenced by the JavaCard servlet.                                                                                                        |
| /META-INF/                      | This is the parent folder containing manifest files.                                                                                                                                   |
| /META-INF/javacard.xml          | This is the JavaCard application descriptor file.                                                                                                                                      |
| /META-INF/MANIFEST.MF           | This is a simple manifest file that's common for JAR and WAR files.                                                                                                                    |
| /src                            | This is the parent folder for all Java source code.                                                                                                                                    |
| src/.../PersistenceServlet.java | This is a servlet that shows the contents of the database, adds new items to the database, and deletes existing items.                                                                 |
| src/.../IndexServlet.java       | This is a servlet that has been mapped to the "index" path of the Java Card WAR application. This servlet presents an HTML form, which invokes the HelloWorldServlet card application. |
| src/.../Database.java           | JavaCard 3 applications have two heap spaces (volatile and non-volatile). This is a simple class that store data in volatile heap.                                                     |
| /WEB-INF                        | This is the parent folder that contains the assets used by the servlets.                                                                                                               |
| /WEB-INF/footer.i               | This is a part of an HTML file.                                                                                                                                                        |
| /WEB-INF/header.i               | This is a part of an HTML file.                                                                                                                                                        |
| /WEB-INF/index.i                | This is a part of an HTML file.                                                                                                                                                        |
| /WEB-INF/web.xml                | This is a typical web.xml that defines and maps the servlets with URLs.                                                                                                                |

Listing 1 is the code for Database.java, a class that allows multiple users in different browser sessions to store and retrieve data objects from the volatile heap space of a Java Card application.

**Listing 1. Database.java**

```
package com.sun.jchowto.persistence;

import java.util.Vector;

/**
 *
 */
```

```

public class Database {
    private static Vector<String> items = new Vector<String>();

    /**
     * @param item
     */
    public static void addItem(String item) {
        if (!items.contains(item)) {
            items.addElement(item);
        }
    }

    /**
     * @return
     */
    public static Vector<String> getItems() {
        return items;
    }

    /**
     * @param item
     * @return
     */
    public static boolean delete(String item) {
        return items.removeElement(item);
    }
}

```

As you can see, the code for the database is pretty simple as it is primarily a wrapper class that allows users to store, retrieve, and delete objects from a String-typed Vector. It demonstrates the capabilities of the Java Card 3 platform well, since it shows that you can use generics, first introduced in the Java SE 5.0.

Listing 2 is the code for the Java Card 3 servlet that accesses the database.

***Listing 2. PersistenceServlet.java***

```

package com.sun.jchowto.persistence;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * A Simple Persistence Servlet.
 */
public class PersistenceServlet extends HttpServlet {    /* (non-
Javadoc)
    * @see
    javax.servlet.http.HttpServlet#doPost(javax.servlet.http.HttpServletR
equest, javax.servlet.http.HttpServletResponse)

```

```

*/

@Override
public void doGet(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException {

    String op = request.getParameter("op");
    String item = request.getParameter("item");

    if (op != null) {
        if (op.equals("Add")) {
            if (item != null) {
                Database.addItem(item);
            }
        } else if (op.equals("Delete")) {
            if (item != null) {
                Database.delete(item);
            }
        }
    }

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    RequestDispatcher dispatcher = null;

    dispatcher = request.getRequestDispatcher("/WEB-
INF/header.i");
    dispatcher.include(request, response);

    out.println("        <tr>");
    out.println("        <td bgcolor=\"#FFFFFF\"
align=\"center\" valign=\"middle\">");
    out.println("        <table bgcolor=\"#000000\"
border=\"0\" width=\"100%\" cellspacing=\"1\" cellpadding=\"15\">");
    out.println("        <tr>");
    out.println("        <td align=\"center\"
bgcolor=\"#FFFFFF\">");

    out.println("</b><big><big>Simple Database</big></big></b>");
    out.println("<form method=\"get\" action=\"database\">");
    out.println("    <table border = 0>");
    out.println("        <tr>");
    out.println("        <td>Item : </td><td colspan=2><input
type=\"text\" name=\"item\"></input></td>");
    out.println("        </tr>");
    out.println("        <tr>");
    out.println("        <td> </td>");
    out.println("        <td><input type=\"submit\"
name=\"op\" value=\"Add\"></td>");
    out.println("        <td><input type=\"submit\"
name=\"op\" value=\"Delete\"></td>");
    out.println("        </tr>");
    out.println("        <tr>");
    out.println("        <td colspan=3><hr></td>");

```

```

        out.println("                </tr>");
        out.println("                <tr>");
        out.println("                    <td colspan=3><b>Items in
Database</b></td>");
        out.println("                </tr>");
        out.println("            <tr>");
        out.println("                <td colspan=3><textarea readonly rows=\"5\"
cols=\"20\">");
        if(Database.getItems().size() <= 0) {
            out.print(" \n \nDatabase is Empty;\n \n ");
        } else {
            for (String str : Database.getItems()) {
                out.println(str);
            }
        }
        out.println("</textarea></td>");
        out.println("</tr>");
        out.println("    </table>");
        out.println("</form>");

        out.println("                                </td>");
        out.println("                                </tr>");
        out.println("                                </table>");
        out.println("                                </td>");
        out.println("                                </tr>");

        dispatcher = request.getRequestDispatcher("/WEB-
INF/footer.i");
        dispatcher.include(request, response);

    }
}

```

As you can see from the preceding code listing, `PersistenceServlet.java` works like any other subclass of `HTTPServlet`. All you need to do is override the HTTP methods that you want, and replace them with your own code. In this case, we're only supporting HTTP GET requests. If the HTTP parameter "op" is set to "Add", then we'll add a new item to the database. If, however, "op" is set to "Delete", then the specified item will be removed from the database.

In either case, the servlet completes its work by showing the final list of contents in the database. The Java Card 3 platform doesn't support JSP files, so you should notice in Listing 2 that much of the application is dedicated to creating the HTML response. Figure 4 is the `PersistenceServlet` in action.

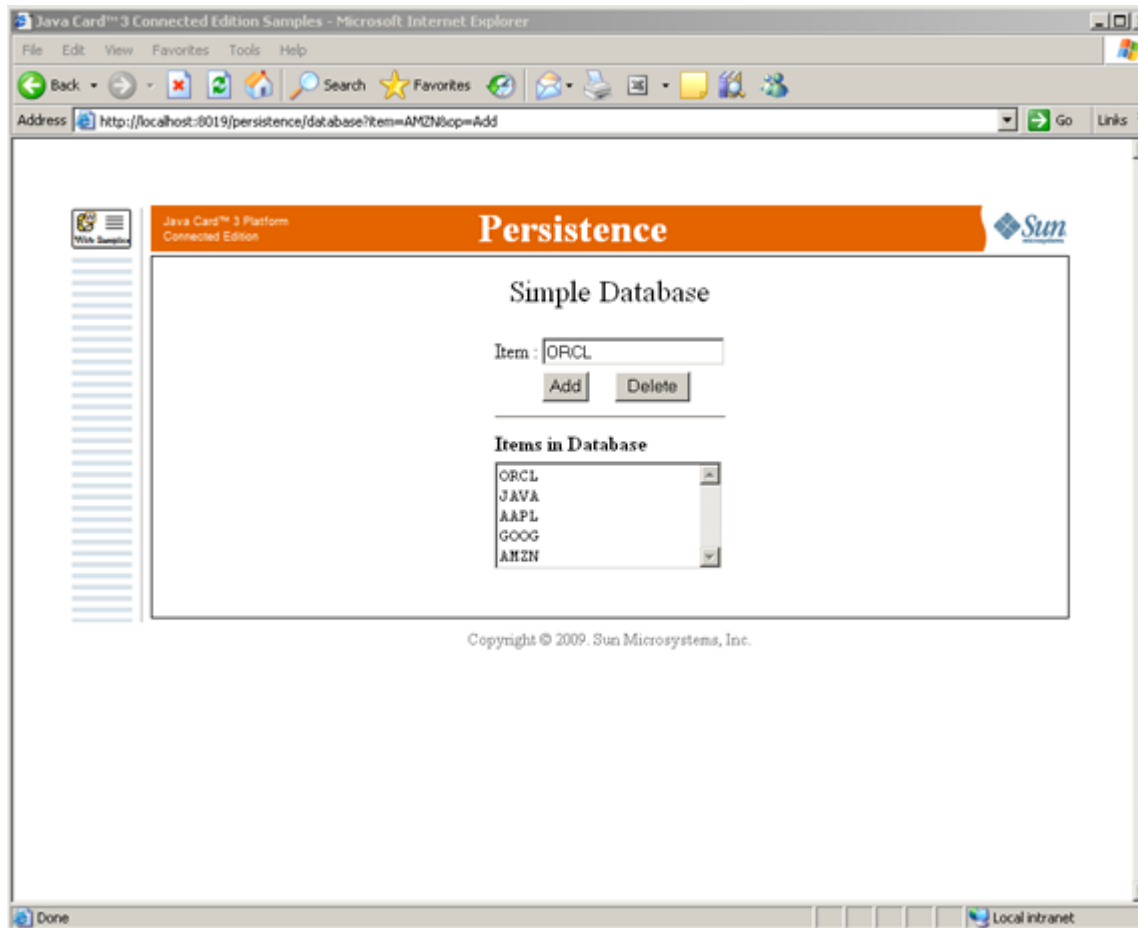


Figure 4. Adding My Favorite Stock Symbols to the PersistenceServlet.java

## RELATED TECHNOLOGIES

### How CGI Scripting Works

The article [How Web Pages Work](#) discusses the basic features of HTML and shows you how to create Web pages that contain text and graphics. It also shows you how to get your page "on the air" with a hosting service. One of the questions frequently asked by new Web site designers once they get their site up is, "What is CGI Scripting and how can I use it on my site?" or, "How do I create interactive forms on my site?"

In this article, we will answer your questions about CGI scripting and show you how to create your own scripts.

You'll also learn a bit about Web servers in the process. Let's get started!

### Web Servers

As described in the article [How Web Servers Work](#), Web servers can be pretty simple. At their most basic, Web servers simply retrieve a file off the disk and send it down the wire to the requesting browser. Let's say you type in the URL

### Shipping Information

Country:  (Billing and shipping address must be in the same country.)

First Name:

Last Name:

Shipping Address 1:

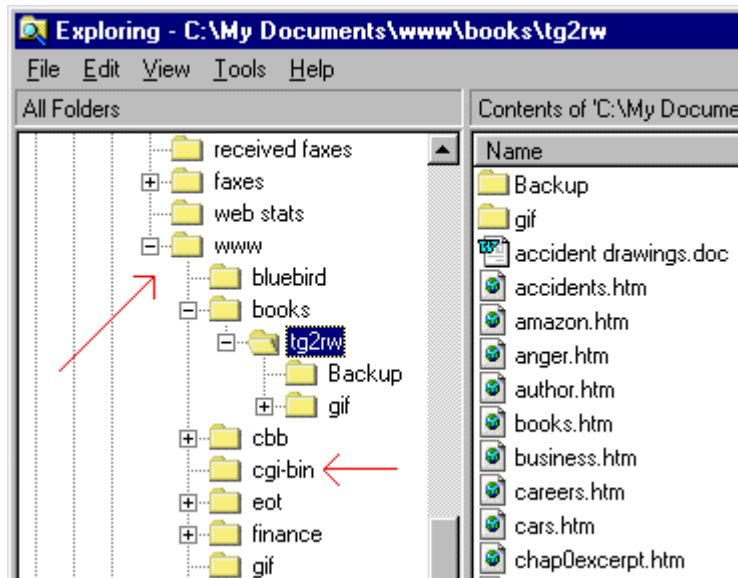
Shipping Address 2:  (optional)

City:

State:

ZIP Code:  (5 or 9 digits)

<http://www.byggpub.com/books/tg2rw/author.htm>. The server gets a request for the file `/books/tg2rw/author.htm`. If you look at the following figure, you can see how the server resolves that request:



During setup, the Web server has been instructed to understand that `c:\My Documents\www` is the server's root directory. It then looks for `/books/tg2rw/author.htm` off of that root. When you ask for the URL <http://www.byggpub.com/books/tg2rw/>, the server understands that you are looking for the default file for that directory. It looks for several different file names to try to find the default file: `index.html`, `index.htm`, `default.html`, `default.htm`. Depending on the server, it may look for others as well. So the server turns <http://www.byggpub.com/books/tg2rw/> into <http://www.byggpub.com/books/tg2rw/index.htm> and delivers that file. All other files must be specified by naming the files explicitly.

This is how all Web servers handle *static* files. Most Web servers also handle **dynamic files** -- through a mechanism called the **Common Gateway Interface**, or CGI. You have seen CGI in all sorts of places on the Web, although you may not have known it at the time. For example:

Any guest book allows you to enter a message in an HTML form and then, the next time the guest book is viewed, the page will contain your new entry.

The WHOIS form at Network Solutions allows you to enter a domain name on a form, and the page returned is different depending on the domain name entered.

Any search engine lets you enter keywords on an HTML form, and then it dynamically creates a page based on the keywords you enter.

All of these dynamic pages use CGI.

The CGI Mechanism

On most Web servers, the CGI mechanism has been standardized in the following way. In the normal directory tree that the server considers to be the root, you create a subdirectory named **cgi-bin**. (You can see this directory in the figure on the previous page.) The server then understands that any file requested from the special `cgi-bin` directory should not simply be read and sent, but instead should be **executed**. The output

of the executed program is what it actually sent to the browser that requested the page. The executable is generally either a pure executable, like the output of a C compiler, or it is a PERL script. PERL is an extremely popular language for CGI scripting.

Imagine that you type the following URL into your browser: **<http://www.howstuffworks.com/cgi-bin/search.pl>**. The server recognized that **search.pl** is in the cgi-bin directory, so it executes **search.pl** (which is a PERL script) and sends the output from the execution to your browser.

You can write your own scripts and try out CGI yourself provided that:

You know a programming language such as C or PERL.

You have access to a Web server that handles CGI scripts. If you have paid a Web hosting service to host your Web site, then chances are you have access to CGI scripting through your host. Check with the hosting service for details. If not, then you can experiment by installing a Web server on your home machine and learning to use it. The second option is a bit more complicated, but you are guaranteed to learn a lot in the process!

#### Simple CGI Scripts

Assuming that you have access to a cgi-bin directory (see the previous section), and assuming that you know either the C programming language or PERL, you can do a whole bunch of interesting experiments with CGI to get your feet wet. Let's start by creating the simplest possible CGI script.

In the article *How Web Pages Work*, we examined the simplest possible HTML Web page. It looked something like this:

```
<html>
<body>
  <h1>Hello there!</h1>
</body>
</html>
```

The simplest possible CGI script would, upon execution, create this simple, static page as its output. Here is how this CGI program would look if you wrote it in C:

```
#include <stdio.h>
```

```
int main()
{
    printf("Content-type: text/html\n\n");
    printf("<html>\n");
    printf("<body>\n");
    printf("<h1>Hello there!</h1>\n");
    printf("</body>\n");
    printf("</html>\n");
    return 0;
}
```

On my Web server, I entered this program into the file **simplest.c** and then compiled it by saying:

```
gcc simplest.c -o simplest.cgi
```

(See *How C Programming Works* for details on compiling C programs.)



By placing **simplest.cgi** in the **cgi-bin** directory, it can be executed. You can try it out now by typing in or clicking on this URL: <http://www.howstuffworks.com/cgi-bin/simplest.cgi>. As you can see, all that the script does is generate a page that says, "Hello there!" The only part that is unexpected is the line that says:

```
printf("Content-type: text/html\n\n");
```

The line "Content-type: text/html\n\n" is special piece of text that must be the first thing sent to the browser by any CGI script. As long as you remember to do that, everything will be fine. If you forget, the browser will reject the output of the script.

You can do the same thing in PERL. Type this PERL code into a file named **simplest.pl**:

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print "<html><body><h1>Hello World!";
print "</h1></body></html>\n";
```

Place the file into your cgi-bin directory. On a UNIX machine, it may help to also type:

```
chmod 755 simplest.pl
```

This tells UNIX that the script is executable. You can try it out now by typing in or clicking on this URL: <http://www.howstuffworks.com/cgi-bin/simplest.pl>.

You have just seen the basic idea behind CGI scripting. It is really that simple! A program executes and its *output* is sent to the browser that called the script. Normal output sent to stdout is what gets sent to the browser.

The whole point of CGI scripting, however, is to create **dynamic content** -- each time the script executes, the output should be different. After all, if the output is the same every time you run the script, then you might as well use a static page. The following C program demonstrates very simple dynamic content:

```
#include <stdio.h>
```

```
int incrementcount()
{
    FILE *f;
    int i;

    f=fopen("count.txt", "r+");
    if (!f)
    {
        sleep(1);
        f=fopen("count.txt", "r+");
        if (!f)
            return -1;
    }
    fscanf(f, "%d", &i);
    i++;
    fseek(f,0,SEEK_SET);
    fprintf(f, "%d", i);
    fclose(f);
    return i;
}
```

Notesengine.com