03 노드의 자바스크립트와 친해지기

노드는 자바스크립트가 기본 언어입니다. 따라서 자바스크립트를 모르면 노드 프로그램을 만들기 어렵습니다. 이 장에서는 자바스크립트의 다양한 기능이나 코드 패턴 중에서 노드 프로그램을 만들 때 알아야 할 몇 가지 내용에 대해 설명합니다. 만약 자바스크립트를 잘 알고 있다면 이 장을 넘어가도 좋습니다. 하지만 자바스크립트를 전혀 몰라서 이 장의 내용도 잘 이해하기 어렵다면 자바스크립트 입문서를 먼저 보기를 권합니다.

질문으로 시작하기

질문1. 객체와 함수는 어떻게 만드나요? ➡ 03-1. 자바스크립트의 객체와 함수 이해하기

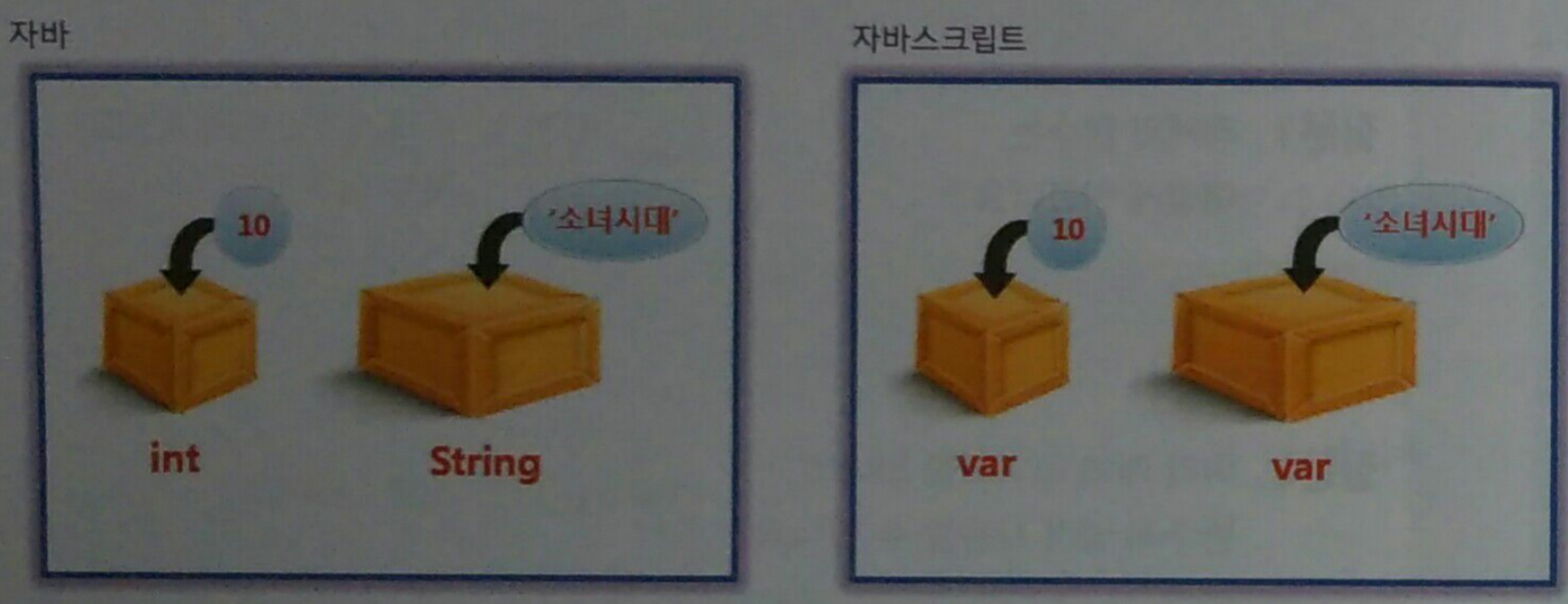
질문2. 여러 개의 데이터를 하나의 변수에 넣어 사용할 수 있나요? ➡ 03-2. 배열 이해하기

질문3. 비동기 방식에서 콜백 함수는 어떻게 사용하나요? ➡ 03-3. 콜백 함수 이해하기

질문4. 프로토타입 객체라는 것은 어떻게 만드나요? ➡ 03-4. 프로토타입 객체 만들기

03-1 | 자바스크립트의 객체와 함수 이해하기

우선 변수라는 것은 데이터를 담아 두는 상자와 같습니다. 그래서 변수를 변수 상자라고 부르기도 하며, 이런 변수 상자는 안에 들어가는 데이터의 크기에 따라 상자의 크기를 다르게 만들 수 있습니다. 자바나 C 언어와 같은 **타입 기반의 언어(Type based language)**는 메모리를 절약하기 위해 정수와 문자열을 만들 때 다른 크기의 변수 상자를 만들고 변수 앞에 int, String과 같은 타입(Type, 자료형)을 지정합니다. 그러나 자바스크립트는 자료형을 명시하지 않습니다. 이 때문에 자바스크립트는 모든 변수를 var 키워드로 선언하고 사용합니다. 자바라는 언어에서 변수를 만들 때 사용하는 방식과 자바스크립트에서 변수를 만들 때 사용하는 방식을 비교하면 다음과 같습니다.



▲ 자바와 자바스크립트의 변수 타입 비교

자바는 숫자 '10'을 넣어두기 위해 변수 상자를 a라는 이름으로 만들고 '소녀시대'라는 글자를 넣어두기 위해 변수 상자를 b라는 이름으로 만듭니다. 이때 두 개의 변수 상자 크기는 서로 다르며, 크기가 서로 다르다는 것을 명확히 알 수 있도록 변수 앞에 int나 String과 같은 타입을 명시합니다. 이것을 자바 코드로 만들 때는 다음과 같이 입력합니다.

```
int a = 10;
String b = '소녀시대';
```

그런데 자바스크립트에서는 int나 String으로 타입을 명시하지 않기 때문에 두 개의 변수 모두 var 키워드로 만듭니다. 자바스크립트 코드로 만들 때는 다음과 같이 입력합니다.

```
var a = 10;
var b = '소녀시대';
```

자바스크립트에서도 숫자를 넣어 둘 때는 문자열보다 작은 크기의 변수 상자를 만들게 되므로 실제로 변수 상자의 크기는 다를 수 있습니다. 따라서 두 개의 변수 모두 var 키워드를 사용했지만 실제로 변수 상자의 크기는 다를 수 있다는 점을 생각해야 합니다. 하지만 변수를 만들 때는 var 키워드만 사용하기 때문에 자바스크립트로 만든 코드는 간결하고 쉬워 보입니다.

자료형	설명
Boolean	[기본 자료형] true와 false의 두 가지 값을 가지는 자료형
Number	[기본 자료형] 64비트 형식의 IEEE 754 값이며 정수나 부동소수 값을 가지는 자료형 몇 가지 상징적인 값을 가질 수 있음: NaN(숫자가 아님), +무한대(Number.MAX_VALUE로 확인), -무한대(Number.MIN_VALUE로 확인)
String	[기본 자료형] 문자열 값을 가지는 자료형
undefined	값을 할당하지 않은 변수의 값
null	존재하지 않는 값을 가리키는 값
Object	객체를 값으로 가지는 자료형 객체는 속성들을 담고 있는 가방(Collection)으로 볼 수 있으며, 대표적인 객체로 Array나 Date를 들 수 있음

[표] 자바스크립트의 자료형

자바스크립트에서 사용할 수 있는 자료형으로 Boolean, Number, String을 들 수 있으며 특별한 값인 undefined와 null에 대해서도 알아두어야 합니다. 새로운 변수를 만들었을 때 실제로는 자료형에 따라 다른 크기의 변수 상자가 만들어지지만 코드에서는 var 키워드만 사용하므로 코드만 보아서는 변수 상자의 크기를 명확히 알 수 없습니다. 이 때문에 타입을 확인할 수 있도록 typeof 연산자를 제공합니다. 또한 문자열을 숫자로 변환할 수 있도록 parseInt( ), parseFloat( ) 함수도 제공합니다.

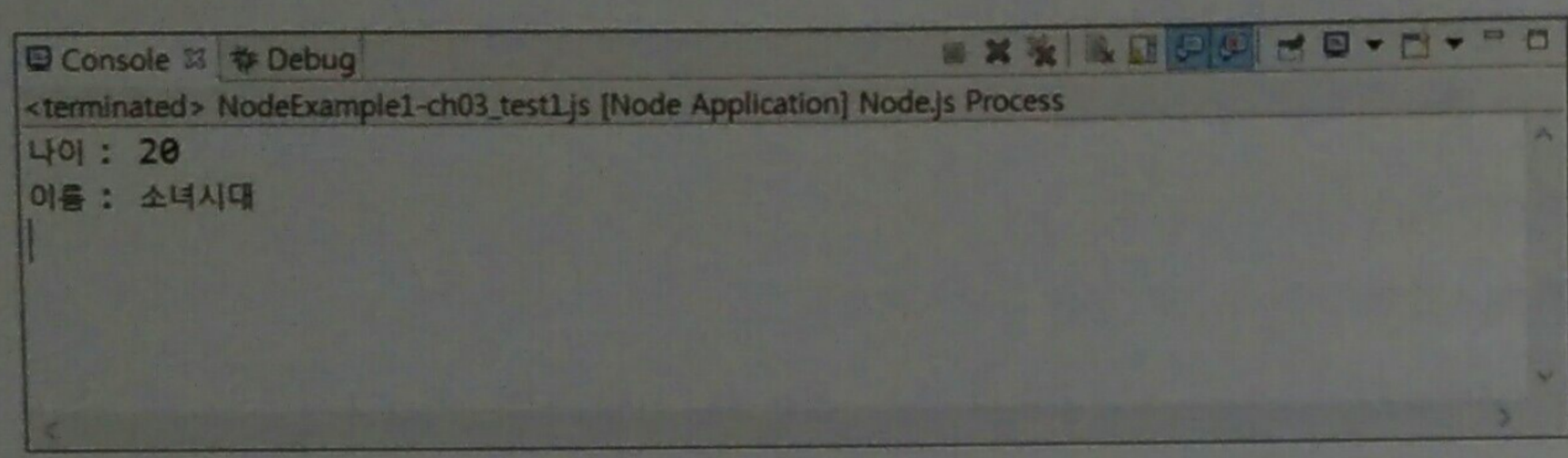
변수로 자료형 알아보기

변수를 만들고 자료형이란 무엇인지 알아보기 위해 코드를 만들어 실행해 보겠습니다. NodeExample1 프로젝트 안에 새로운 자바스크립트 파일 ch03_test1.js를 만들고 다음 코드를 입력합니다.

참조 파일 | NodeExample1>ch03_test1.js

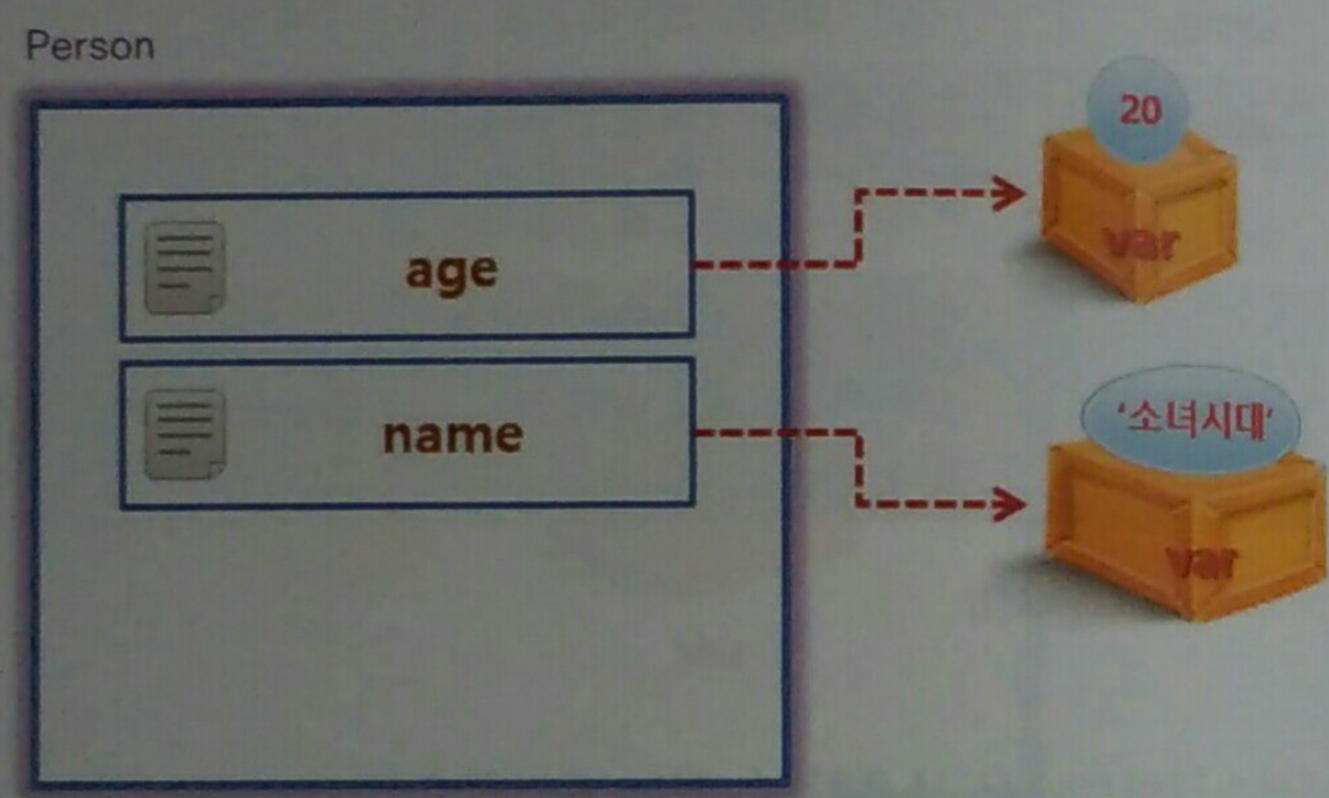
```
var age = 20;
console.log('나이 : %d', age);

var name = '소녀시대';
console.log('이름 : %s', name);
```

숫자 '20'을 담고 있는 age 변수 상자와 '소녀시대'라는 문자열을 담고 있는 name 변수 상자를 만들었습니다. 두 개의 변수 모두 console.log( ) 메소드를 호출하여 콘솔 창에 뿌려 줍니다. 파일을 실행하면 나이와 이름이 표시됩니다.



▲ 자바스크립트의 변수에 들어 있는 값을 콘솔 창에 뿌리기

이렇게 숫자나 문자열을 변수 상자에 넣어 두는 방법은 어렵지 않습니다. 자바스크립트의 변수에는 숫자, 문자열, 이진(boolean)과 같은 값을 넣을 수 있으며 객체를 넣을 수도 있습니다. 자바스크립트에서 객체를 만들고 싶다면 중괄호를 이용해 만들 수 있으며, 중괄호로 묶은 객체 안에는 속성을 추가할 수 있습니다.



▲ 자바스크립트의 객체

객체 안에 들어 있는 속성의 이름은 하나의 변수로 생각할 수 있으며, 변수의 이름과 변수 값 또는 속성의 이름과 속성 값이라는 형태로 조합됩니다. 이렇게 객체에 넣을 수 있는 속성은 점(.) 연산자와 속성의 이름을 이용해 접근할 수도 있고 대괄호([ ])와 속성의 이름을 이용해서 접근할 수도 있습니다.

ch03_test2.js 파일을 새로 만들고 다음 코드를 입력합니다.

참조 파일 | NodeExample1>ch03_test2.js

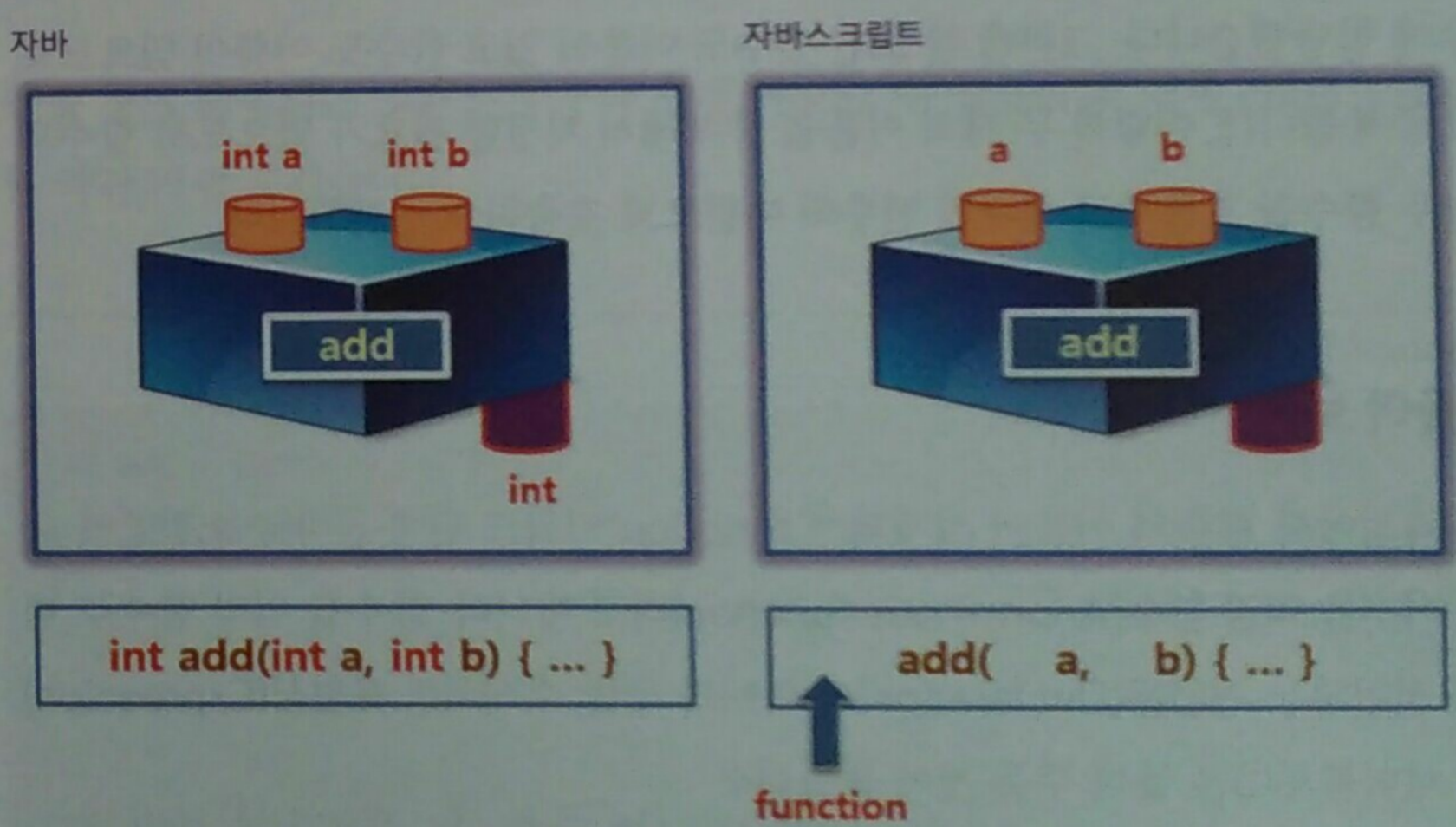
```
var Person = {};

Person["age"] = 20;
Person["name"] = '소녀시대';
Person.mobile = '010-1000-1000';

console.log('나이 : %d', Person.age);
console.log('이름 : %s', Person.name);
console.log('전화 : %s', Person["mobile"]);
```

Person이라는 이름으로 변수를 선언했으며 이 변수에는 빈 객체가 하나 할당됩니다. 객체는 중괄호만 사용해서 만들고 이 객체에 속성을 넣고 싶다면 객체 이름 뒤에 대괄호를 붙인 후 그 안에 속성의 이름을 넣어 줍니다. 여기에서는 age 속성과 name 속성을 추가했습니다. 속성은 점 연산자를 사용해서 접근할 수도 있는데 위의 코드에서 mobile 속성은 점 연산자를 사용해 추가했습니다. 이렇게 추가한 세 개의 속성이 가지는 값을 콘솔 창에 출력하고 싶은 경우에도 대괄호와 속성 이름 또는 점 연산자와 속성 이름을 사용합니다. 파일을 실행하면 세 개의 속성 값이 출력됩니다.

```
Console  Debug
<terminated> NodeExample1-ch03_test2.js [Node Application] Node.js Process
나이 : 20
이름 : 소녀시대
전화 : 010-1000-1000
```

▲ 객체에 들어 있는 속성 값을 출력한 경우

지금까지 타입을 명시적으로 표시하지 않는 자바스크립트의 특징을 생각하면서 변수와 객체를 가장 간단한 형태로 만들어 보았습니다. 그러면 함수를 사용할 때는 어떨까요? 자바스크립트가 타입을 표시하지 않기 때문에 함수를 선언하고 호출하는 형태도 약간 달라집니다. 다음 그림은 타입 기반 언어인 자바와 비교한 자바스크립트의 함수 형태입니다.

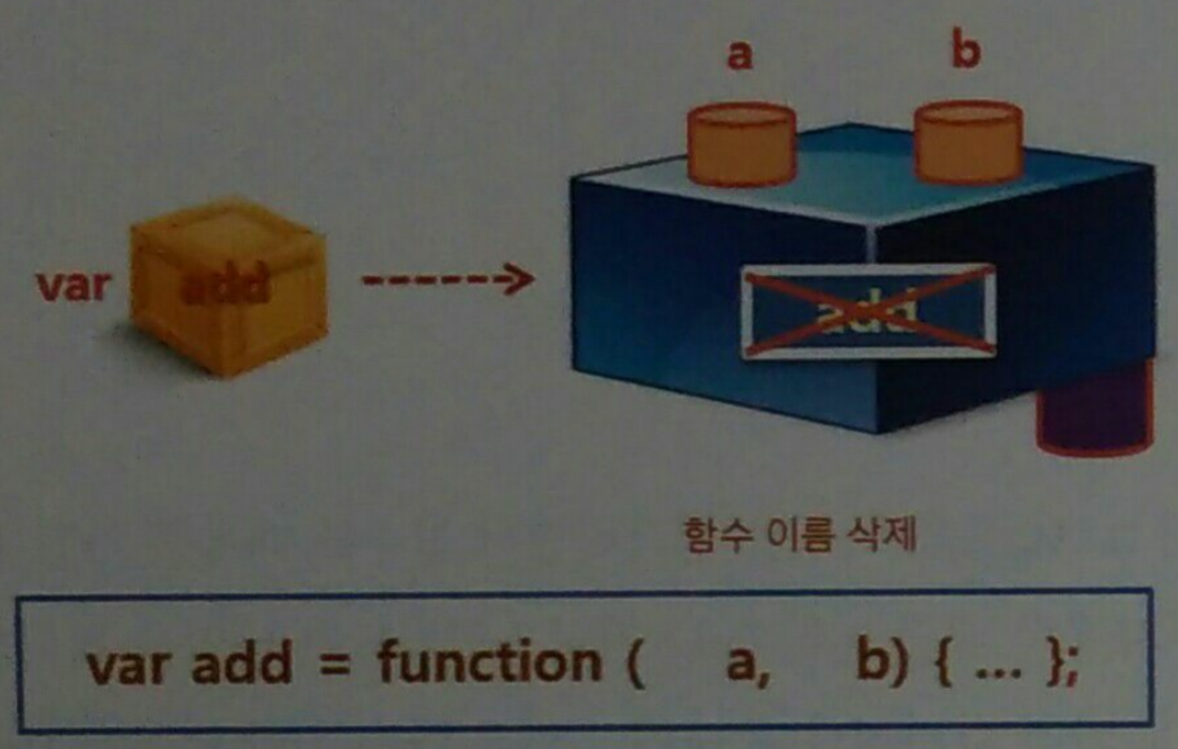


▲ 자바와 자바스크립트의 함수 비교

자바에서 더하기 함수를 만드는 경우, 먼저 함수 상자의 이름을 add라고 붙일 수 있습니다. 함수 상자는 입력 값과 출력 값을 갖는데 이 입력 값을 a와 b라는 변수로 받아 함수 상자에 넣어 준다면 add라는 함수 이름 뒤에 소괄호를 붙인 후 파라미터(매개변수) 이름과 타입을 적어야 합니다. 또한 함수의 출력 값이 정수라면 함수 이름 앞에도 int와 같은 타입을 표시해야 합니다.

그러나 자바스크립트에서는 타입을 표시하지 않으므로 타입이 표시되었던 부분은 모두 사라집니다. 심지어 파라미터로 전달되는 변수에는 var 키워드도 붙일 필요가 없습니다. 이 때문에 함수를 정의하는 형태가 더 단순해집니다. 다만 함수라는 것을 명시적으로 구별하기 위해 function 키워드를 함수 이름 앞에 붙여 줍니다.

중요한 것은 앞에서 사용했던 자바스크립트의 변수에는 일반 데이터나 객체 이외에 함수도 할당할 수 있다는 점입니다. 이것이 자바스크립트의 가장 중요한 특징 중 하나입니다. 변수에 함수를 할당한다는 것, 어떤 의미일까요? 다음 그림은 더하기 함수를 add 변수에 할당하는 과정을 보여 줍니다.



▲ 변수에 함수를 할당하는 경우

함수를 만든 후 add 변수에 할당했습니다. 그런데 잘 보면 변수도 이름이 있고 함수도 이름이 있으므로 변수 이름과 함수 이름이 중복됩니다. 이렇게 두 개의 이름을 중복해서 지정할 필요가 없으므로 함수의 이름을 삭제합니다. 이 경우 함수를 호출하고 싶다면 변수의 이름으로 호출하여 실행합니다.

자바스크립트 함수 만들어 보기

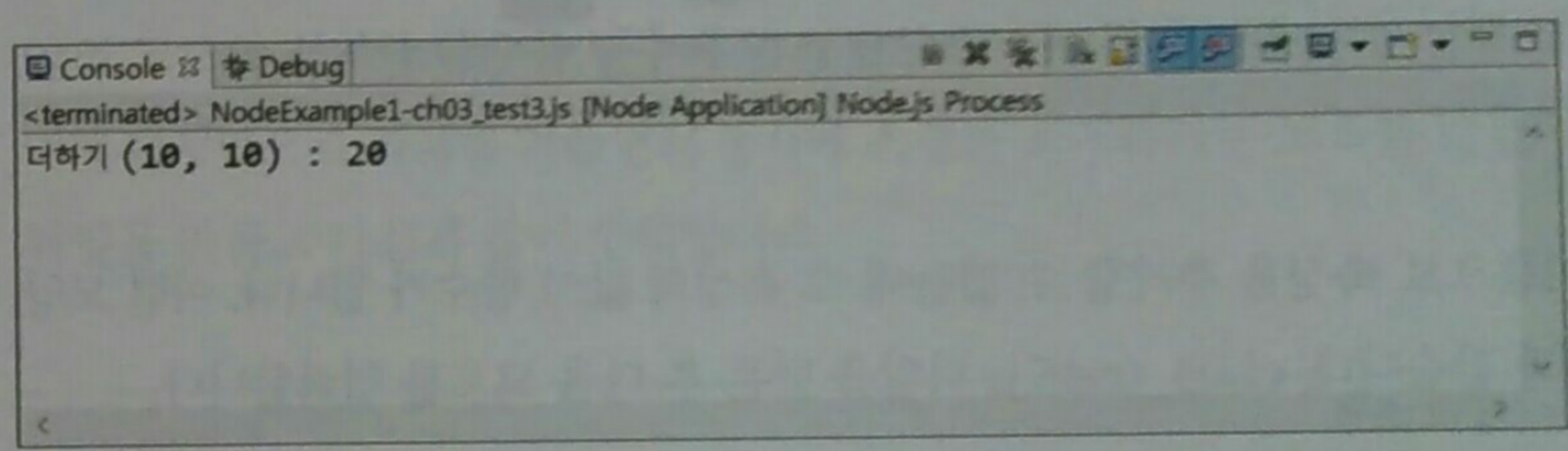
함수를 변수에 할당하는 과정에서 함수의 이름이 삭제되어 function 키워드 뒤에 곧바로 소괄호가 붙는 형태로 만들어지는데, 이것을 **익명 함수**(Anonymous function)라고 합니다. 함수를 익명 함수로 변수에 할당할 때는 함수를 선언하는 **선언문**(Declaration)이 아니라 일반 수식처럼 **표현식**(Expression)이 되므로 가장 마지막에 세미콜론(;)을 붙여 주는 것이 좋습니다.

자바스크립트의 함수를 만들어보기 위해 ch03_test3.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch03_test3.js

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

var result = add(10, 10);

console.log('더하기 (10, 10) : %d', result);
```

먼저 add 함수를 정의한 후 함수를 호출하고 그 결과 값을 result 변수에 넣었습니다. 함수를 정의할 때는 function 뒤에 함수의 이름이 온다는 것을 쉽게 확인할 수 있습니다. 그리고 함수를 호출할 때는 함수 이름과 소괄호를 사용한다는 것도 알 수 있습니다. 이 파일을 실행하면 다음과 같은 결과가 나타납니다.



▲ 더하기 함수를 만들어 실행한 결과

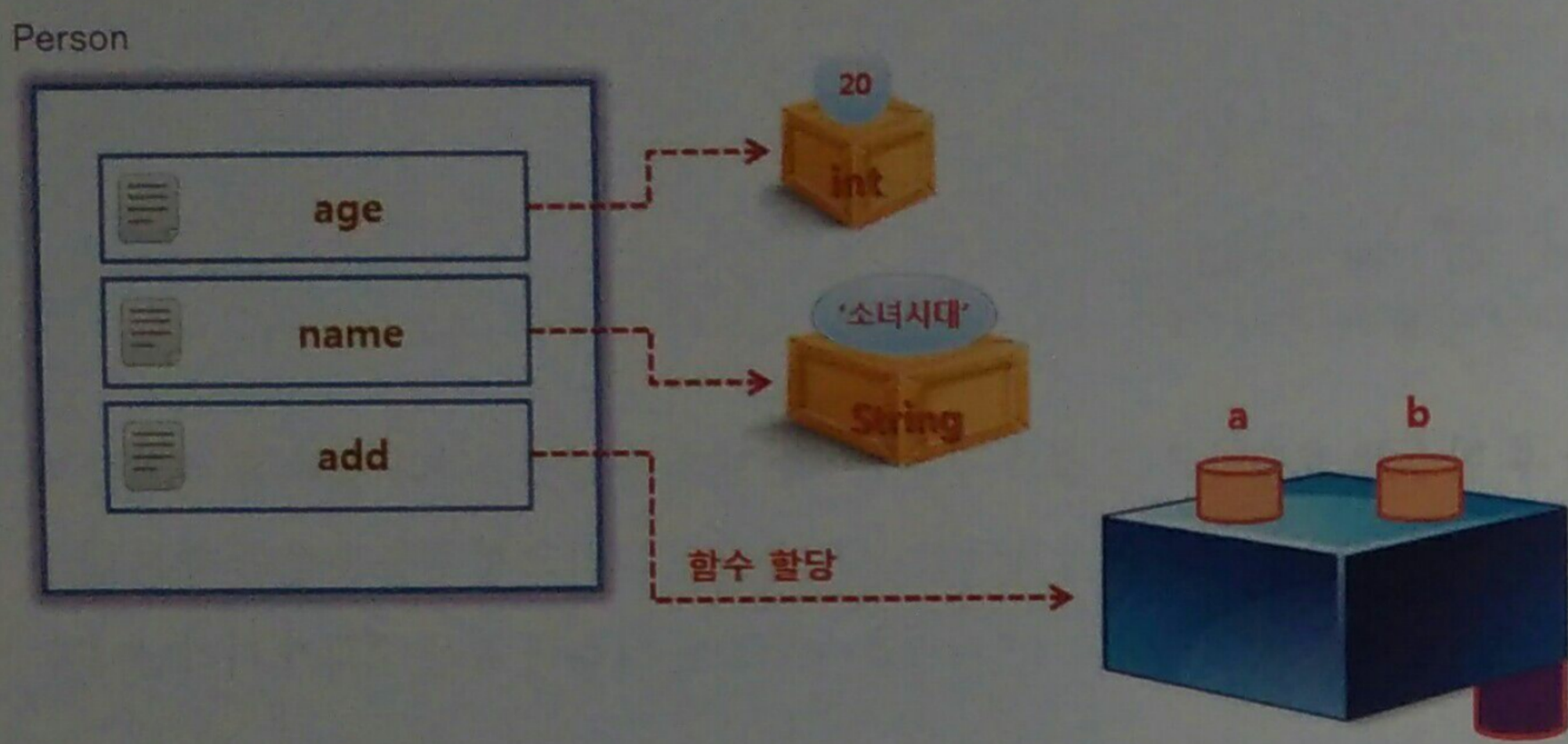
콘솔 창에 출력되는 결과는 누구나 쉽게 예상할 수 있습니다. 이제 ch03_test4.js 파일을 만들고 코드를 약간 바꿔 보겠습니다.

참조 파일 | NodeExample1>ch03_test4.js

```
var add = function (a, b) {
  return a + b;
};

var result = add(10, 10);

console.log('더하기 (10, 10) : %d', result);
```

이 파일을 실행하면 ch3_test3.js 파일을 실행했을 때와 같은 결과가 출력됩니다. 하지만 더하기 함수에 함수 이름이 없고 함수는 만들어지자마자 add 변수에 할당되었다는 점이 다릅니다. 자바스크립트의 함수는 변수에 할당되는 경우가 많으므로 함수를 이런 코드 패턴으로 만들 수 있다는 점을 잘 알아두어야 합니다.

그런데 함수가 변수에 할당될 수 있다면 객체 안에 속성으로도 들어갈 수 있습니다. 즉, 객체의 속성에는 변수를 만들어 할당할 수 있으므로 객체 안의 속성에 함수를 할당할 수 있습니다. 다음은 객체의 속성으로 더하기 함수를 할당했을 때를 표현한 그림입니다.



▲ 객체의 속성으로 더하기 함수를 할당하는 경우

다른 속성들처럼 add라는 이름으로 속성을 추가할 수 있는데 그 속성의 값이 함수가 됩니다. 이런 모양을 실제 코드로 만들면 다음과 같습니다. ch03_test5.js 파일을 만든 후 다음 코드를 입력합니다.

참조 파일 | NodeExample1>ch03_test5.js

```
var Person = {};

Person["age"] = 20;
Person["name"] = '소녀시대';
Person.add = function(a, b) {
  return a + b;
};

console.log('더하기 : %d', Person.add(10, 10));
```

이렇게 입력한 ch03_test5.js 파일을 실행하면 앞에서와 같이 더하기가 정상적으로 실행된 후 그 결과가 출력됩니다. 더하기 함수를 호출할 때는 객체의 속성으로 할당된 함수를 호출해야 하므로 Person.add( )와 같은 형태로 호출합니다. 이런 형태의 코드는 다음과 같이 일차적으로 변수에 함수를 할당한 후 다시 그 변수를 객체의 속성으로 추가되도록 수정할 수도 있습니다. ch3_test5.js 파일을 복사하여 ch03_test6.js 파일을 만든 후 다음과 같이 수정합니다.

참조 파일 | NodeExample1>ch03_test6.js

```
var Person = {};

Person["age"] = 20;
Person["name"] = '소녀시대';

var add = function(a, b) {
  return a + b;
};

Person["add"] = add;

console.log('더하기 : %d', Person.add(10, 10));
```

앞의 코드와 같지만 함수를 변수에 먼저 할당했다는 점이 다릅니다. 실행 결과도 앞의 코드와 같습니다. 이제 마지막으로 객체를 만들면서 동시에 속성을 초기화하는 코드를 만들어 보겠습니다. ch03_test7.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch03_test7.js

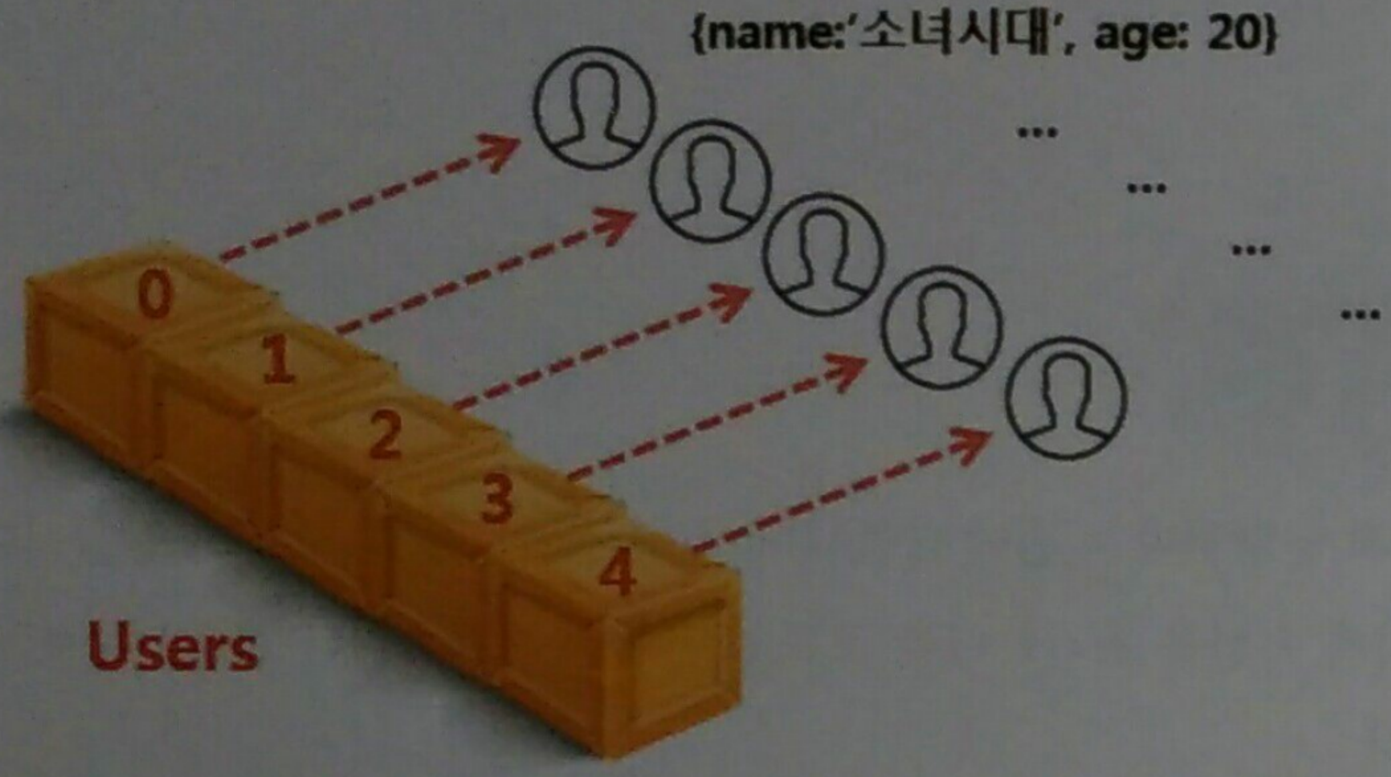
```
var Person = {
  age: 20,
  name: '소녀시대',
  add: function(a, b) {
    return a + b;
  }
};

console.log('더하기 : %d', Person.add(10, 10));
```

자바스크립트 객체는 중괄호를 이용해 만드는데, 객체를 만들 때 중괄호 안에 속성을 바로 넣어 줄 수도 있습니다. 이 속성에 포함된 속성 이름과 속성 값은 콜론(:)으로 구분하며 각각의 속성은 콤마(,)로 구분합니다. 속성에는 숫자나 문자열과 같은 일반적인 데이터와 함께 함수도 할당될 수 있으므로 더하기 함수를 add라는 이름의 속성으로 추가했습니다. 이 파일을 실행하면 이전과 마찬가지로 더하기 결과가 출력됩니다.

지금까지 자바스크립트에서 변수를 어떻게 만들고 객체는 어떻게 만드는지 그리고 함수는 어떻게 만드는지 알아보았습니다. 가장 중요한 점은 함수를 변수에 할당할 수 있다는 것인데, 이렇게 함수를 변수에 할당할 수 있기 때문에 다양한 작업이 가능해집니다. 함수를 변수에 할당하여 사용하는 코드는 앞으로 자주 보게 될 것입니다.

03-2 | 배열 이해하기

변수와 함수에 대해 알아보았으니 이번에는 배열에 대해 간단하게 정리해 보겠습니다. **배열**은 여러 개의 데이터를 하나의 변수에 담아 둘 수 있어서 자주 사용되며, 배열 안에 들어 있는 요소(또는 원소, Item)들은 대괄호 [ ]를 이용해서 접근할 수 있습니다. 배열이 만들어지는 모양은 다음과 같습니다.



▲ 배열이 만들어지는 모양

만약 다섯 명의 사용자 정보를 관리하기 위해 다섯 개의 객체를 만들고 각각 다른 이름의 변수에 할당했다면 매번 다섯 개의 변수에 접근하고 관리해야 합니다. 그런데 Users 변수를 하나만 만들고 그 안에 다섯 개의 사용자 객체를 넣어 한꺼번에 관리할 수 있다면 각각을 따로 관리하는 것보다 훨씬 편리해집니다.

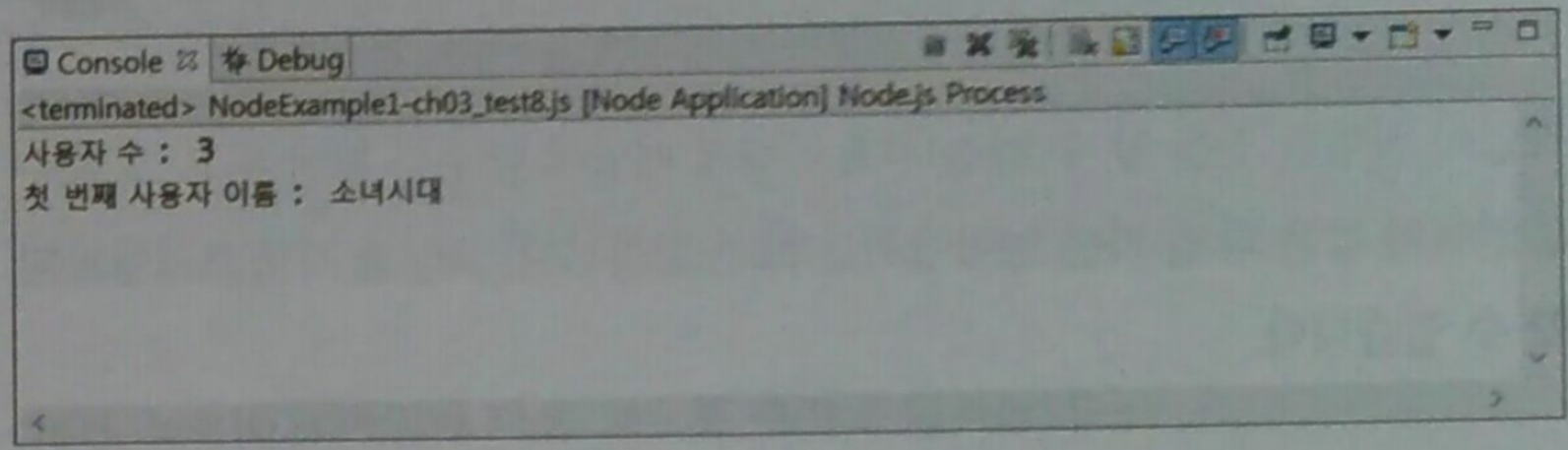
이렇게 여러 개의 데이터를 편리하게 다룰 수 있기 때문에 배열을 자주 사용합니다. 배열에 추가되는 요소들은 0부터 시작하는 **인덱스(Index)** 번호로 접근할 수 있습니다. 배열을 만들 때는 대괄호 [ ]를 이용해 만들고 그 안에 요소를 추가할 때는 push( ) 메소드를 사용합니다.

프로젝트에 ch03_test8.js 파일을 만들고 다음 코드를 입력합니다.

참조 파일 | NodeExample1>ch03_test8.js

```
var Users = [{name:'소녀시대', age:20}, {name:'걸스데이', age:22}];

Users.push({name:'티아라', age:23});

console.log('사용자 수 : %d', Users.length);
console.log('첫 번째 사용자 이름 : %s', Users[0].name);
```

배열 객체를 하나 만든 후 Users 변수에 할당했습니다. 배열 안에는 name과 age 속성을 가진 객체가 두 개 들어 있습니다. 이 배열에 새로운 객체를 추가하고 싶다면 push( ) 메소드를 호출하면 되고 배열에 들어 있는 요소들의 개수를 알아내려면 length 속성을 사용하면 됩니다. 각각의 배열 요소에 접근하기 위해서는 대괄호와 함께 그 요소의 인덱스 정보를 사용합니다. 이 파일을 실행하면 다음과 같이 사용자 객체의 수가 3으로 출력됩니다.



▲ 배열에 추가한 사용자 객체의 수와 배열 요소에 접근한 결과 출력

그런데 이미 앞에서 알아본 것처럼 변수에는 함수를 할당할 수 있다고 했습니다. 따라서 배열의 각 요소에도 함수를 직접 할당할 수 있습니다. push( ) 함수를 호출하여 함수를 배열 객체에 추가해 보겠습니다. ch03_test9.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch03_test9.js

```
var Users = [{name:'소녀시대', age:20}, {name:'걸스데이', age:22}];

var add = function(a, b) {
  return a + b;
};

Users.push(add)

console.log('배열 요소의 수 : %d', Users.length);
console.log('세 번째 요소로 추가된 함수 실행 : %d', Users[2](10, 10));
```

add 변수에 할당되었던 더하기 함수를 Users 배열에 그대로 추가했습니다. 이렇게 추가한 배열 요소는 세 번째 요소가 되므로 Users[2] 코드로 접근할 수 있습니다. 이렇게 접근한 요소가 함수이므로 함수를 실행하기 위해서는 Users[2](10, 10) 코드를 사용할 수 있습니다. 이렇게 하면 대괄호 뒤에 바로 소괄호가 오는 코드 형태가 만들어져 약간 복잡하게 보입니다. 다만 어떻게 이런 코드가 만들어질 수 있는지를 생각하면서 보면 이해가 될 것입니다.

```
Console  Debug
<terminated> NodeExample1-ch03_test9.js [Node Application] Node.js Process
배열 요소의 수 : 3
세번째 요소로 추가된 함수 실행 : 20
```

▲ 배열에 함수를 추가하고 배열 안에 들어 있는 함수 요소를 그대로 실행한 결과

실행 결과를 보면 함수가 문제없이 실행된 것을 알 수 있습니다. 이렇게 배열에 함수를 넣어 두고 그 함수를 실행하는 방식은 자바나 C 언어와 같은 타입 기반 언어에서는 보기 힘듭니다. 그만큼 자바스크립트의 함수는 아주 유연하게 사용할 수 있습니다.

배열의 모든 요소 하나씩 확인하기

이번에는 배열 안에 있는 모든 요소를 하나씩 확인하는 방법을 알아보겠습니다. 배열 안의 모든 요소를 확인하는 가장 간단한 방법은 for문을 사용하는 것입니다. 새 파일 ch03_test10.js를 만들고 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch03_test10.js

```
var Users = [{name:'소녀시대', age:20}, {name:'걸스데이', age:22}, {name:'티아라', age:23}];

console.log('배열 요소의 수 : %d', Users.length);
for (var i = 0; i < Users.length; i++) {
  console.log('배열 요소 #' + i + ' : %s', Users[i].name);
}
```

Users 배열 안에 들어 있는 각 객체들은 for문 안에서 하나씩 접근할 수 있으며, Users[i] 코드로 접근한 객체의 name 속성을 순서대로 콘솔 창에 출력할 수 있습니다. 파일을 실행하면 다음과 같은 결과가 출력됩니다.

```
Console  Debug
<terminated> NodeExample1-ch03_test10.js [Node Application] Node.js Process
배열 요소의 수 : 3
배열 요소 #0 : 소녀시대
배열 요소 #1 : 걸스데이
배열 요소 #2 : 티아라
```

▲ 배열 요소를 for문으로 하나씩 접근한 결과

for문처럼 배열의 각 요소에 하나씩 접근할 수 있는 또 다른 방법은 forEach( ) 메소드를 사용하는 것입니다. forEach( ) 메소드를 호출할 때는 함수를 파라미터로 전달하게 되는데 이 함수는 배열에 들어 있는 각 요소를 확인할 때마다 반복해서 호출됩니다.

정박사의 한마디 하나의 함수가 다른 함수의 파라미터로 전달될 수 있다고요?

함수를 만들 때 변수를 파라미터로 전달할 수 있습니다. 그런데 자바스크립트에서는 함수가 변수에 할당될 수 있다고 했으므로 하나의 함수가 다른 함수의 파라미터로 전달될 수 있습니다. 이에 대해서는 다음 장에서 자세히 설명합니다.

ch03_test10.js 파일에 다음 코드를 추가로 입력합니다.

참조 파일 | NodeExample1>ch03_test10.js

```
.....

console.log('\nforEach 구문 사용하기');
Users.forEach(function(item, index) {
  console.log('배열 요소 #' + index + ' : %s', item.name);
});
```

forEach( ) 메소드를 호출할 때 전달하는 함수에는 두 개의 파라미터가 들어가는데, 첫 번째는 배열의 각 요소이며 두 번째는 각 요소의 인덱스 값입니다. 이 파일을 실행하면 for문을 실행했을 때와 같은 결과가 표시됩니다.

```
Console  Debug
<terminated> NodeExample1-ch03_test10.js [Node Application] Node.js Process
배열 요소의 수 : 3
배열 요소 #0 : 소녀시대
배열 요소 #1 : 걸스데이
배열 요소 #2 : 티아라

forEach 구문 사용하기
배열 요소 #0 : 소녀시대
배열 요소 #1 : 걸스데이
배열 요소 #2 : 티아라
```

▲ 배열 요소를 forEach문으로 하나씩 접근한 결과

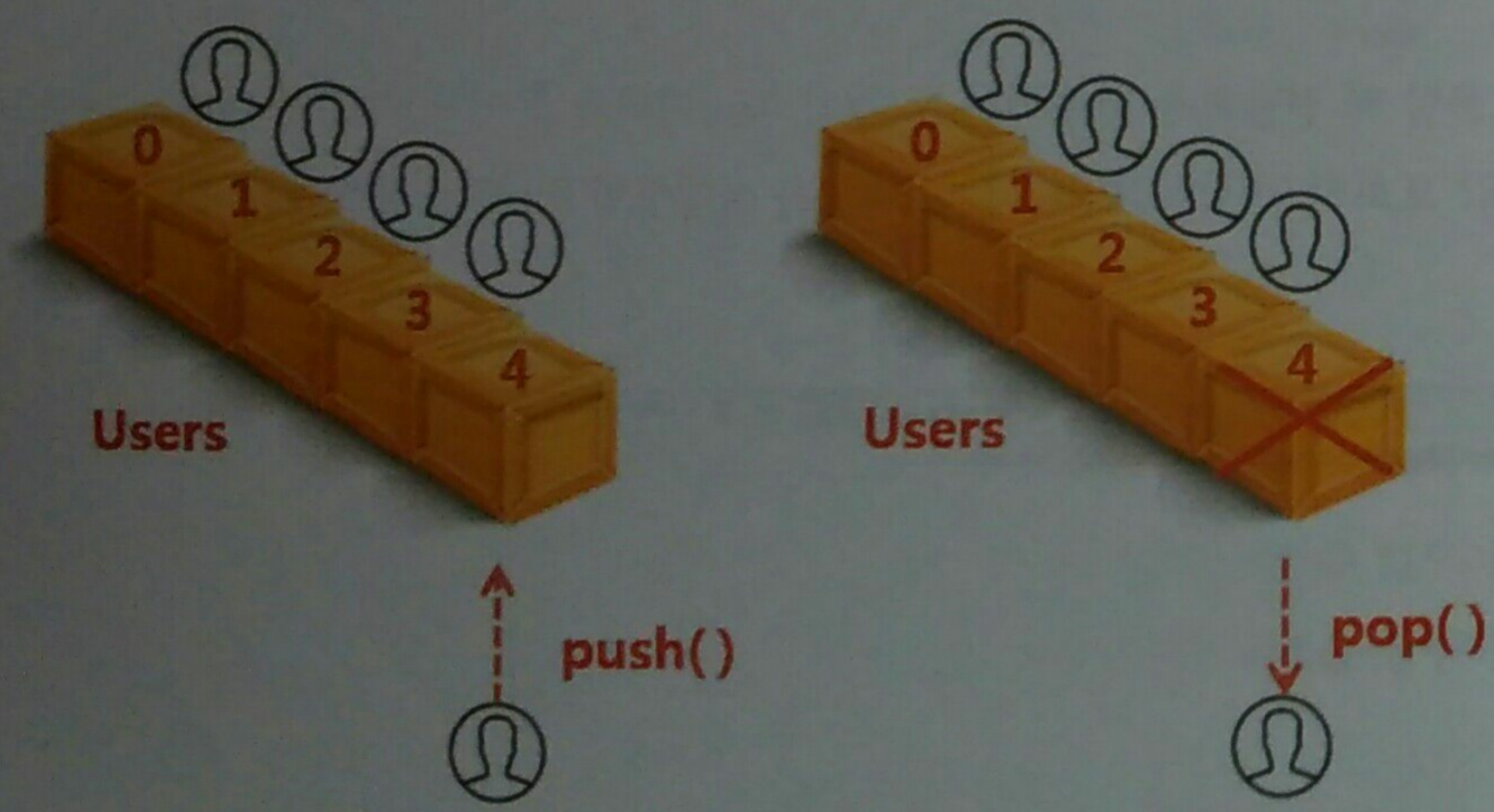
배열에 값 추가 및 삭제하기

배열의 모든 요소를 확인하는 방법에 대해서도 알아보았습니다. 마지막으로 배열에 값을 추가하거나 삭제하는 방법을 알아봅니다. 배열에는 다음과 같은 메소드가 포함되어 있어 그 요소 값을 확인하거나 추가 또는 삭제할 수 있습니다.

속성 / 메소드 이름	설명
push(object)	배열의 끝에 요소를 추가합니다.
pop( )	배열의 끝에 있는 요소를 삭제합니다.
unshift( )	배열의 앞에 요소를 추가합니다.
shift( )	배열의 앞에 있는 요소를 삭제합니다.
splice(index, removeCount [,object])	여러 개의 객체를 요소로 추가하거나 삭제합니다.
slice(index, copyCount)	여러 개의 요소를 잘라내어 새로운 배열 객체로 만듭니다.

[표] 배열 객체의 대표적인 속성과 메소드

먼저 배열의 끝에 요소를 추가하거나 삭제할 때는 push( )와 pop( ) 메소드를 사용합니다. 배열에 객체를 추가하는 경우가 많기 때문에 push( ) 메소드는 자주 사용됩니다. 다음 그림은 push( )와 pop( ) 메소드가 객체를 어떻게 배열에 넣고 빼는지 알려 줍니다.



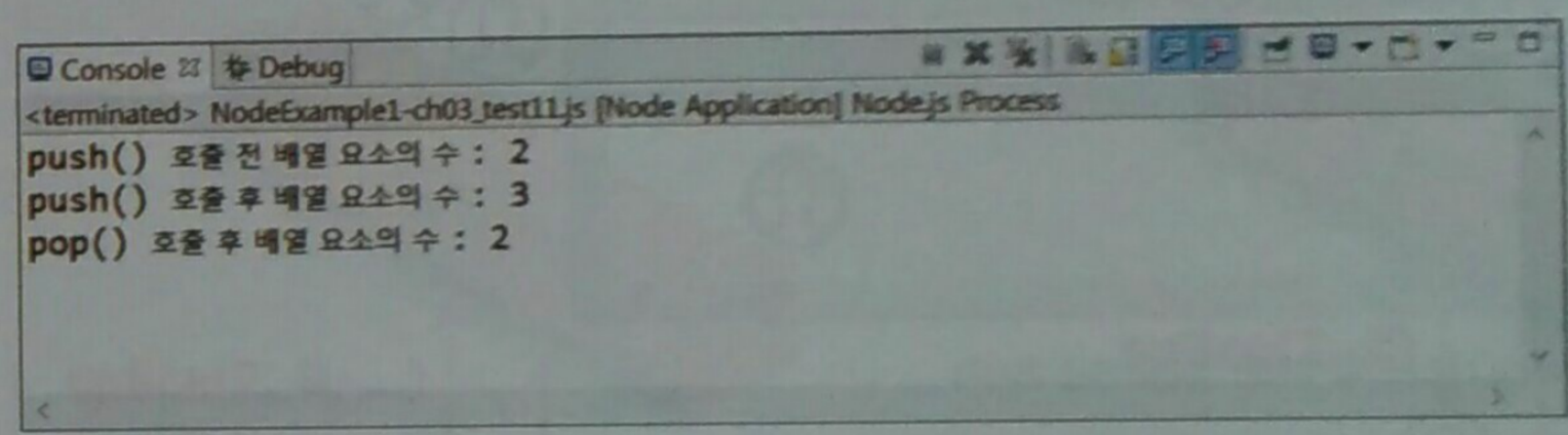
▲ push( )와 pop( ) 메소드로 배열의 끝에 요소를 추가하거나 삭제하기

사용자 객체를 배열에 넣었다가 삭제하는 기능을 만들기 위해 ch03_test11.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch03_test11.js

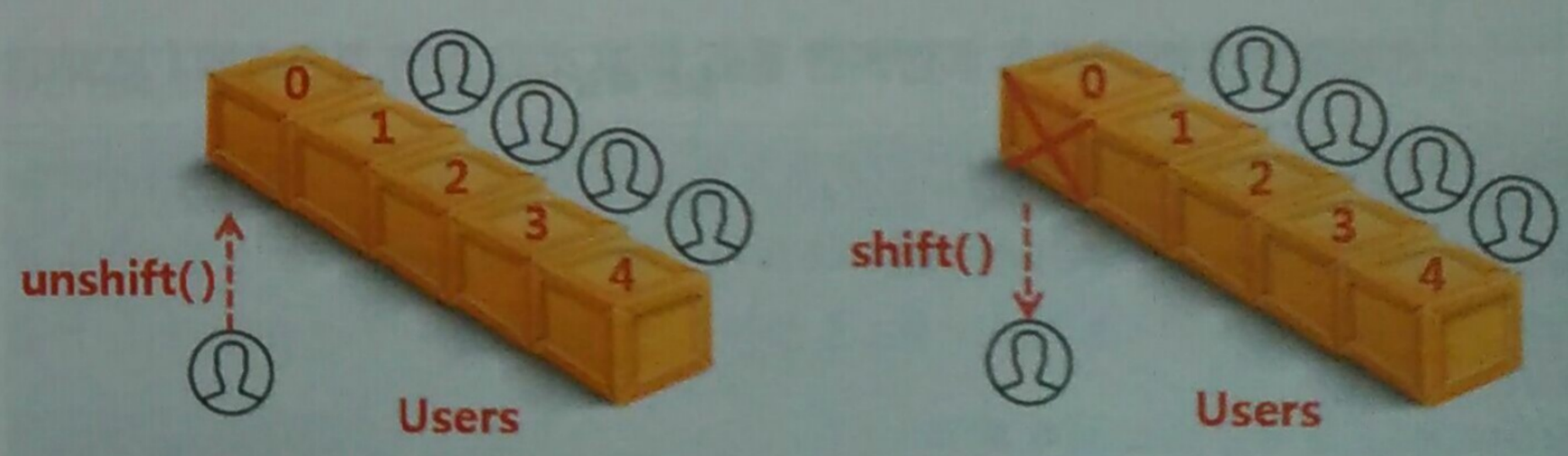
```
var Users = [{name:'소녀시대', age:20}, {name:'걸스데이', age:22}];
console.log('push() 호출 전 배열 요소의 수 : %d', Users.length);

Users.push({name:'티아라', age:23});
console.log('push() 호출 후 배열 요소의 수 : %d', Users.length);

Users.pop();
console.log('pop() 호출 후 배열 요소의 수 : %d', Users.length);
```

배열 객체가 할당된 Users 변수에는 사용자 정보가 들어 있는 객체가 두 개 있습니다. 그 배열 객체에 또 다른 객체를 넣기 위해 push( ) 메소드를 호출한 후 배열 요소의 수가 늘어났는지 확인합니다. 그다음 pop( ) 메소드를 호출하면 배열의 가장 끝에 있는 요소가 삭제됩니다. 이 파일을 실행하면 콘솔 창에 다음과 같은 메시지가 출력됩니다.



▲ push( ) 메소드로 객체를 추가했다가 pop( ) 메소드로 객체를 삭제한 경우의 콘솔 메시지

push( )와 pop( ) 메소드는 배열의 가장 끝부분에 요소를 추가하거나 삭제하지만 unshift( )와 shift( ) 메소드는 가장 앞에 요소를 추가하거나 삭제합니다.



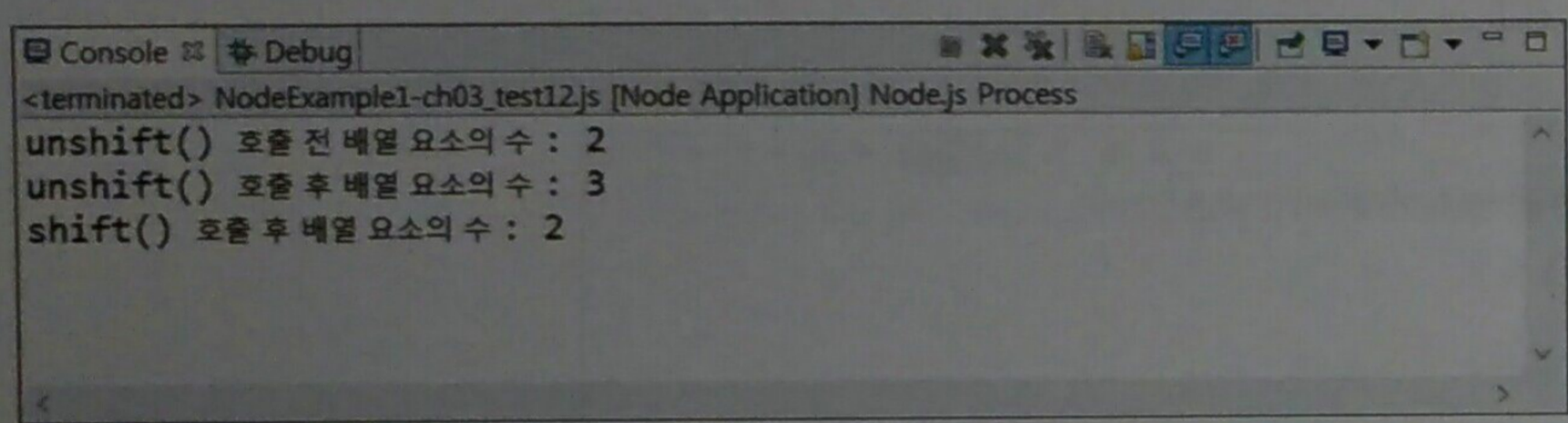
▲ unshift( )로 배열의 앞부분에 요소를 추가하고 shift( ) 메소드로 배열의 앞부분에 있는 요소를 삭제하기

배열의 앞쪽에 요소를 추가했다가 삭제하는 기능을 만들기 위해 ch03_test12.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch03_test12.js

```
var Users = [{name:'소녀시대', age:20}, {name:'걸스데이', age:22}];
console.log('unshift() 호출 전 배열 요소의 수 : %d', Users.length);

Users.unshift({name:'티아라', age:23});
console.log('unshift() 호출 후 배열 요소의 수 : %d', Users.length);

Users.shift();
console.log('shift() 호출 후 배열 요소의 수 : %d', Users.length);
```

Users 변수에 할당된 배열 객체에는 두 개의 요소가 들어 있습니다. unshift( ) 메소드를 호출하여 객체를 하나 더 추가했다가 shift( ) 메소드를 호출하여 삭제합니다. 이 파일을 실행하면 콘솔 창에 다음과 같은 메시지가 출력됩니다.



▲ unshift( ) 메소드로 객체를 추가했다가 shift( ) 메소드로 삭제한 경우의 콘솔 메시지

그런데 배열 중간에 있는 요소를 삭제하고 싶다면 어떻게 해야 할까요? delete 키워드를 사용하면 인덱스를 이용해 배열 요소를 삭제할 수 있습니다. ch03_test13.js 파일을 만들고 다음과 같이 delete 키워드를 이용해 배열 요소를 삭제하는 코드를 입력합니다.

참조 파일 | NodeExample1>ch03_test13.js

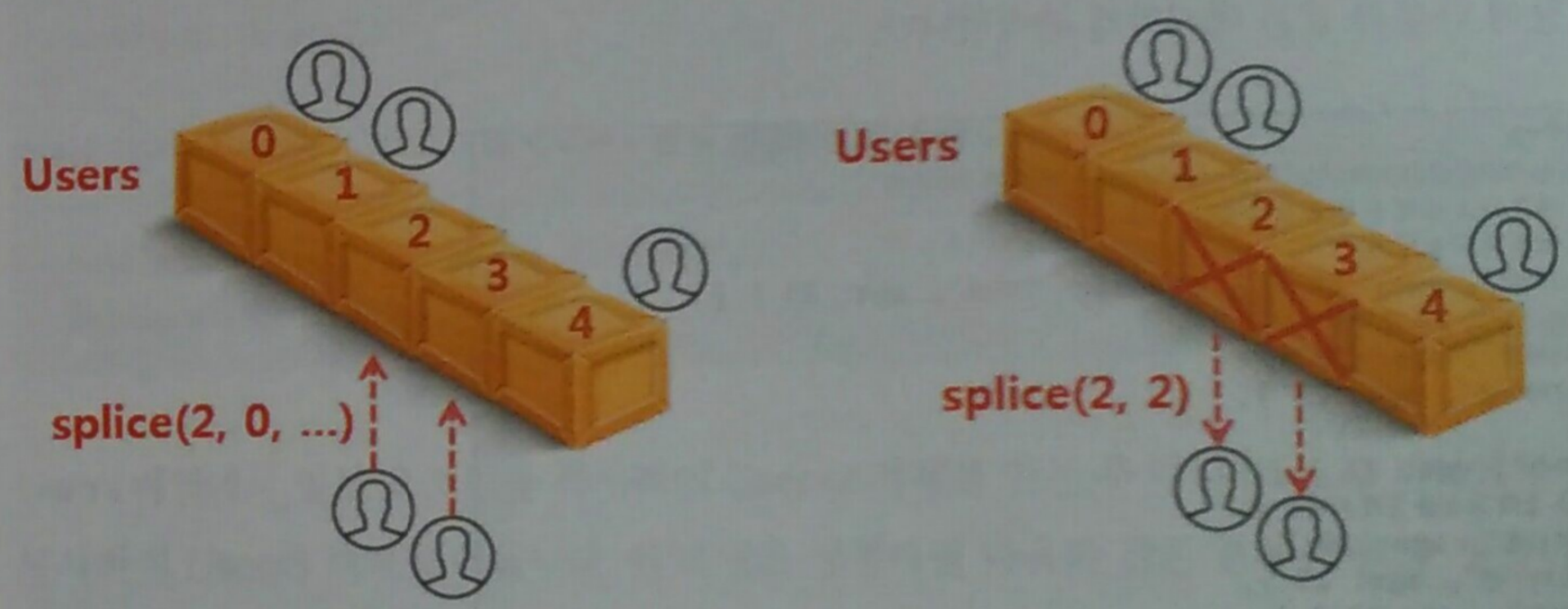
```
var Users = [{name:'소녀시대', age:20}, {name:'걸스데이', age:22}, {name:'티아라', age:23}];
console.log('delete 키워드로 배열 요소 삭제 전 배열 요소의 수 : %d', Users.length);

delete Users[1];
console.log('delete 키워드로 배열 요소 삭제 후');
console.dir(Users);
```

배열 객체에는 세 개의 객체가 들어 있는데 delete 키워드를 이용해서 두 번째 요소를 삭제합니다. 인덱스는 0부터 시작하므로 delete Users[1] 코드를 사용하면 두 번째 요소를 삭제할 수 있습니다. 이 파일을 실행하면 배열 요소가 삭제된 것을 볼 수 있습니다.

```
Console  Debug
<terminated> NodeExample1-ch03_test13.js [Node Application] Node.js Process
delete 키워드로 배열 요소 삭제 전 배열 요소의 수 : 3
delete 키워드로 배열 요소 삭제 후
[ { name: '소녀시대', age: 20 }, , { name: '티아라', age: 23 } ]
```

▲ delete 키워드로 배열 요소 한 개를 삭제한 경우 출력되는 메시지

그런데 이상한 점이 보입니다. 분명 두 번째 요소는 삭제되었는데 배열의 개수는 그대로 세 개입니다. 이것은 배열 안에 요소를 담아 둘 수 있는 공간이 그대로 남겨진 채 객체만 삭제되었기 때문입니다. 이런 경우에 빈 공간이 undefined로 남아 배열 객체를 다룰 때 혼동을 일으킬 수도 있습니다. 따라서 배열 요소를 담는 공간까지 없애 주는 splice( ) 메소드를 사용하는 것이 좋습니다. splice( ) 메소드는 배열 요소 여러 개를 한꺼번에 추가하거나 삭제할 수 있는 방법입니다.



▲ splice( ) 메소드를 이용해 배열 요소를 추가하거나 삭제하기

splice( ) 메소드로 배열 요소 여러 개를 한꺼번에 추가하거나 삭제하기

splice( ) 메소드를 호출할 때 전달하는 첫 번째 파라미터는 인덱스 값으로 배열의 몇 번째 요소부터 처리할 것인지를 지정합니다. 두 번째 파라미터는 삭제할 요소의 개수를 지정합니다. 만약 다섯 개의 객체가 들어 있는 Users 배열에서 세 번째와 네 번째 요소를 삭제하고 싶다면 다음과 같이 호출합니다.

```
Users.splice(2, 2);
```

세 번째 요소의 인덱스 값이 2이므로 첫 번째 파라미터의 값은 2가 되고 두 개의 요소를 삭제할 것이므로 두 번째 파라미터의 값도 2가 됩니다. 만약 값을 추가하고 싶다면 두 번째 파라미터의 값을 0으로 입력한 후 추가하려는 객체들을 파라미터로 전달합니다.

splice( ) 메소드를 이용해 배열에 요소들을 추가했다가 삭제하는 기능을 만들기 위해 ch03_test13.js 파일을 다시 만들고 다음 코드를 추가합니다.

참조 파일 | NodeExample1>ch03_test13.js

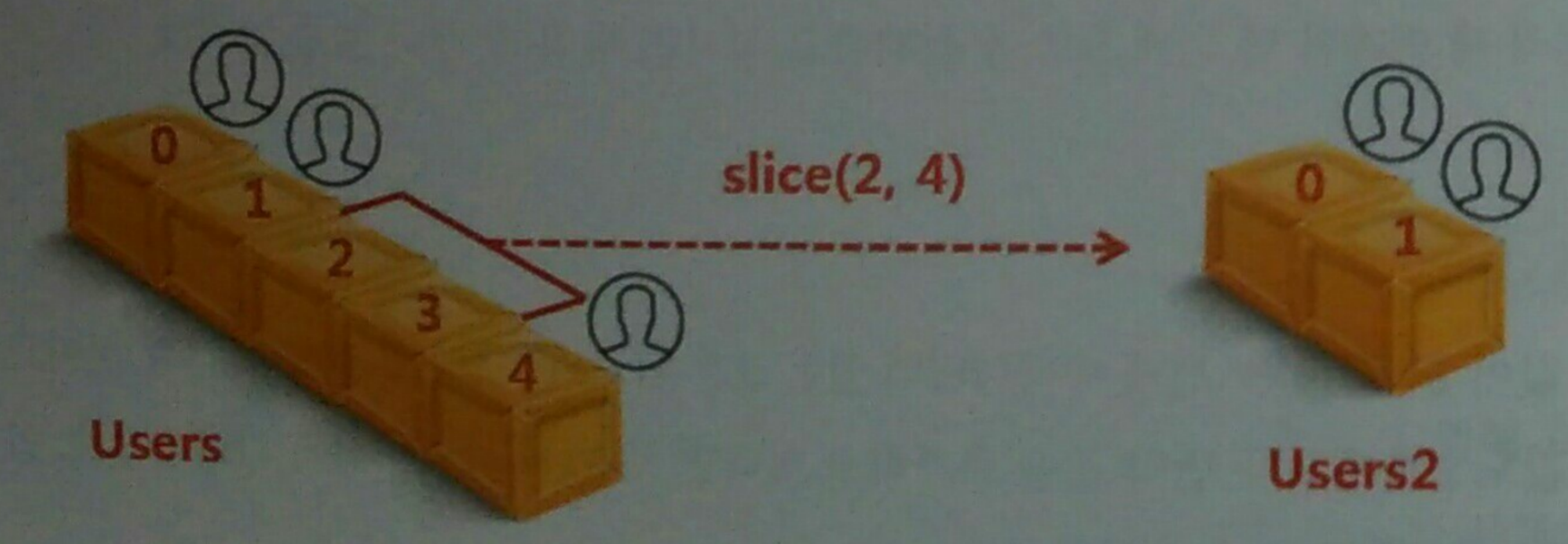
```
……

Users.splice(1, 0, {name:'애프터스쿨', age:25});
console.log('splice()로 요소를 인덱스 1에 추가한 후');
console.dir(Users);

Users.splice(2, 1);
console.log('splice()로 인덱스 1의 요소를 1개 삭제한 후');
console.dir(Users);
```

배열 객체를 할당한 Users 변수에 대해 splice( ) 메소드를 호출하면서 첫 번째 파라미터로 1을 전달하여 두 번째 위치에 객체를 추가합니다. 그다음 세 번째 위치에 있는 요소 한 개를 삭제합니다. 파일을 실행하면 콘솔 창에 다음과 같은 메시지가 출력됩니다.

```
Console  Debug
<terminated> NodeExample1-ch03_test13.js [Node Application] Node.js Process
delete 키워드로 배열 요소 삭제 전 배열 요소의 수 : 3
delete 키워드로 배열 요소 삭제 후
[ { name: '소녀시대', age: 20 }, , { name: '티아라', age: 23 } ]
splice()로 요소를 인덱스 1에 추가한 후
[ { name: '소녀시대', age: 20 },
  { name: '애프터스쿨', age: 25 },
  ,
  { name: '티아라', age: 23 } ]
splice()로 인덱스 1의 요소를 1개 삭제한 후
[ { name: '소녀시대', age: 20 },
  { name: '애프터스쿨', age: 25 },
  { name: '티아라', age: 23 } ]
```

▲ splice( ) 메소드를 이용해 배열 요소를 추가하거나 삭제한 경우의 메시지

마지막으로 slice( ) 메소드에 대해 알아봅니다. slice( ) 메소드는 배열의 일부 요소들을 복사하여 새로운 배열을 만들어 줍니다.



▲ slice( ) 메소드를 이용해 배열의 일부 요소를 복사하여 새로운 배열을 만들기

slice( ) 메소드에 전달하는 파라미터는 두 개입니다. 첫 번째는 복사할 요소의 시작 위치이며 두 번째는 끝 위치입니다. ch03_test14.js 파일을 새로 만들고 다음 코드를 입력합니다.

참조 파일 | NodeExample1>ch03_test14.js

```
var Users = [{name:'소녀시대', age:20}, {name:'걸스데이', age:22}, {name:'티아라', age:23}
            ,{name:'애프터스쿨', age:25}];

console.log('배열 요소의 수 : %d', Users.length);
console.log('원본 Users');
console.dir(Users);

var Users2 = Users.slice(1, 3);

console.log('slice()로 잘라낸 후 Users2');
console.dir(Users2);

var Users3 = Users2.slice(1);

console.log('slice()로 잘라낸 후 Users3');
console.dir(Users3);
```

Users 배열에서 일부 요소들을 복사하여 Users2 객체를 만든 후 다시 Users2 객체에서 일부 요소들을 복사하여 Users3 객체를 만듭니다. 이 파일을 실행하면 다음과 같은 결과를 볼 수 있습니다.

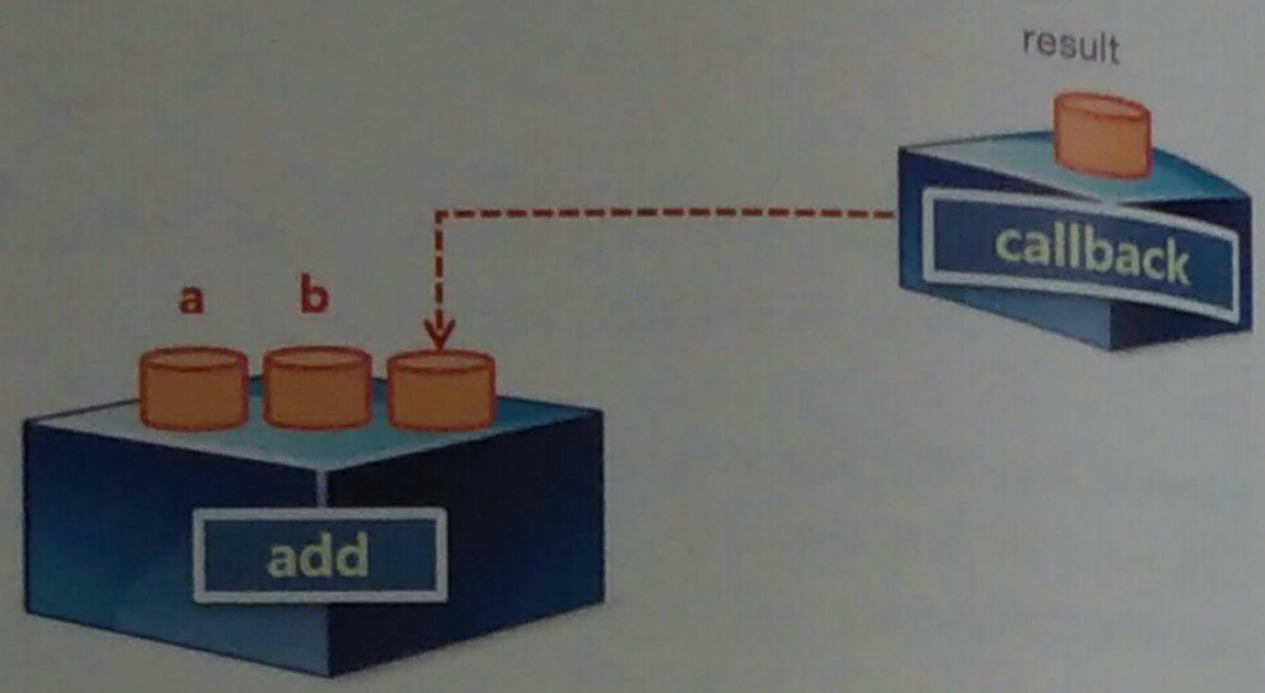
```
Console  Debug
<terminated> NodeExample1-ch03_test14.js [Node Application] Node.js Process
배열 요소의 수 : 4
원본 Users
[ { name: '소녀시대', age: 20 },
  { name: '걸스데이', age: 22 },
  { name: '티아라', age: 23 },
  { name: '애프터스쿨', age: 25 } ]
slice()로 잘라낸 후 Users2
[ { name: '걸스데이', age: 22 }, { name: '티아라', age: 23 } ]
slice()로 잘라낸 후 Users3
[ { name: '티아라', age: 23 } ]
```

▲ slice( ) 메소드를 이용해 배열의 일부 요소를 복사하여 새로운 배열을 만든 결과

데이터를 배열 객체에 넣어 두고 사용할 때가 많으므로 배열 객체의 사용법은 잘 알아두세요. 또한 이와 같은 코드는 노드를 사용할 때 자주 볼 수 있으므로 이미 알고 있던 내용이 아니라면 익숙해질 수 있도록 반복하여 연습하길 바랍니다.

03-3 | 콜백 함수 이해하기

자바스크립트의 변수에는 숫자나 문자열 같은 데이터, 그리고 중괄호를 이용해 만든 객체뿐 아니라 함수도 할당할 수 있다고 했습니다. 이렇게 변수에 함수를 할당할 수 있다는 특징은 함수를 호출할 때 다른 함수를 파라미터로 전달하거나 함수 안에서 또 다른 함수를 만들어 반환할 수 있습니다.



콜백 함수를 파라미터로 전달

```
function add(a, b, callback) {
    var result = a + b ;
    callback(result);
}
```

함수를 파라미터로 전달하기 ▶

함수를 호출했을 때 또 다른 함수를 파라미터로 전달하는 방법

함수를 파라미터로 전달하는 경우는 대부분 **비동기 프로그래밍**(Non-Blocking Programming) 방식으로 코드를 만들 때입니다. 예를 들어, 더하기 함수를 실행한 후 결과 값이 반환될 때까지 기다리지 않고 그다음 코드를 실행하려면 비동기 방식으로 코드를 만들어야 합니다. 즉, 더하기 함수를 실행하는 데 시간이 걸릴 수 있기 때문에 그다음 코드를 바로 실행합니다. 그러고 나서 연산이 끝났을 때 파라미터로 전달한 함수가 실행될 수 있다면 그 시점에 결과를 처리할 수 있으므로 효율적인 프로그램을 만들 수 있습니다. 이때 파라미터로 전달되는 함수를 **콜백 함수**(Callback function)라고 합니다. 콜백 함수는 함수가 실행되는 중간에 호출되어 상태 정보를 전달하거나 결과 값을 처리하는 데 사용됩니다.

ch03_test15.js 파일을 새로 만들고 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch03_test15.js

```
function add(a, b, callback) {
  var result = a + b;
  callback(result);
}

add(10, 10, function(result) {
  console.log('파라미터로 전달된 콜백 함수 호출됨.');
  console.log('더하기 (10, 10)의 결과 : %d', result);
});
```

더하기 함수를 정의할 때는 더 이상 값을 반환하지 않도록 return 키워드를 사용하는 코드 부분을 삭제합니다. 그 대신 더하기 연산을 한 결과 값은 파라미터로 전달된 콜백 함수를 호출하면서 그 콜백 함수로 전달합니다. 콜백 함수는 미리 변수에 할당해 두었다가 add( ) 함수를 호출할 때 파라미터로 전달할 수도 있지만, add( ) 함수를 호출할 때 익명 함수로 만들어서 파라미터로 바로 전달할 수도 있습니다. 콜백 함수는 더하기 연산을 하는 코드 아래에서 호출되는데 이때 콘솔 창에 메시지를 출력합니다. 파일을 실행하면 다음과 같은 메시지를 볼 수 있습니다.

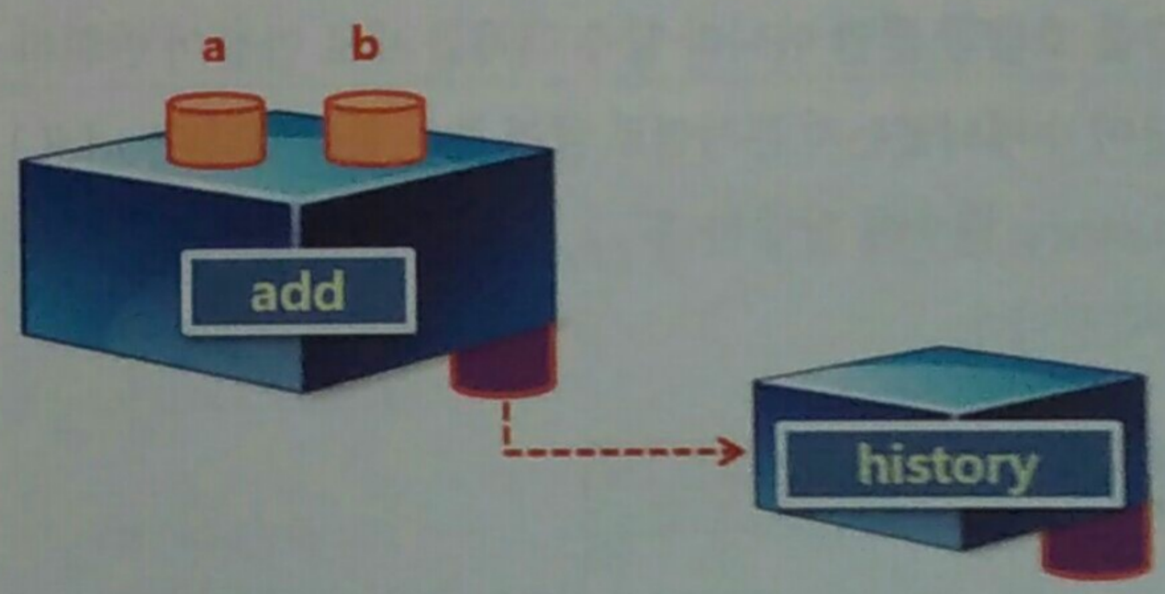
```
Console  Debug
<terminated> NodeExample1-ch03_test15.js [Node Application] Node.js Process
파라미터로 전달된 콜백 함수 호출됨.
더하기 (10, 10)의 결과 : 20
```

▲ 콜백 함수로 더하기 연산을 받아 출력하기

주의! 노드에서는 대부분의 코드를 비동기 방식으로 만들기 때문에 위와 같이 콜백 함수를 만들어 파라미터로 전달하고 콜백 함수가 호출될 때 결과 값을 콘솔 창에 뿌려 줄 수 있습니다.

함수 안에서 값을 반환할 때 새로운 함수를 만들어 반환하는 방법

지금까지 함수를 호출했을 때 또 다른 함수를 파라미터로 전달하는 방법에 대해 살펴보았습니다. 그러면 함수 안에서 값을 반환할 때 새로운 함수를 만들어 반환하는 방법도 살펴보겠습니다.




```
function add(a, b, callback) {
var result = a + b;
callback(result);

var history = function() {
  return a + ' + ' + b + ' = ' + result;
};
return history;
}
```

함수를 반환

▲ 함수에서 반환하는 값이 함수 객체인 경우

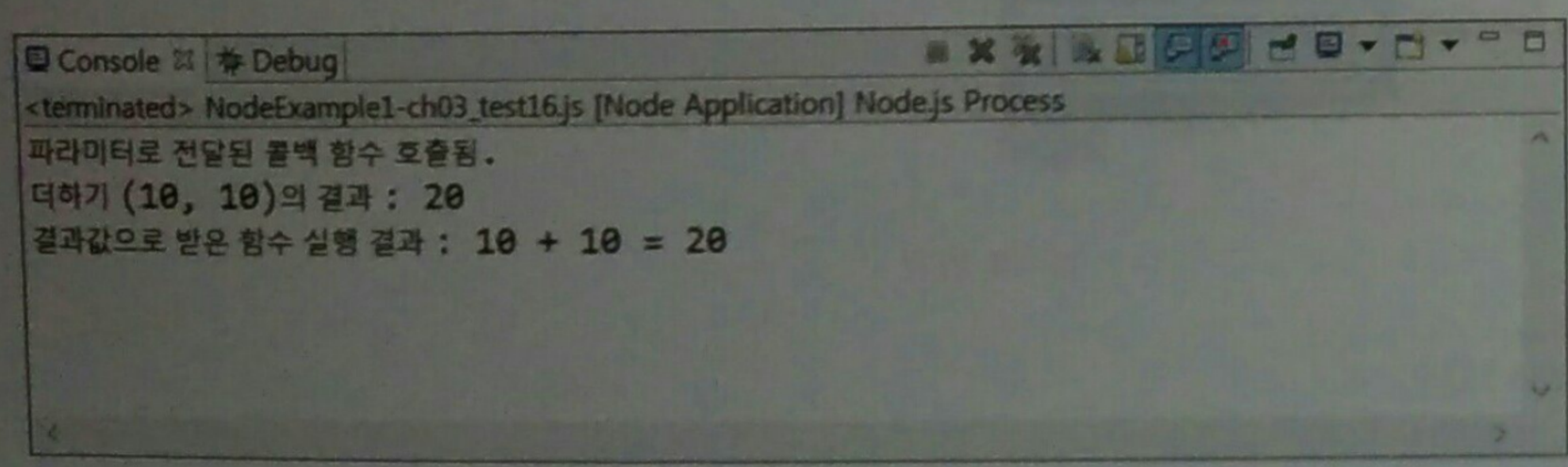
어떤 함수를 실행했을 때 또 다른 함수를 반환받으면 반환받은 함수를 그대로 실행할 수 있습니다. 이렇게 만들면 하나의 함수를 실행했을 때 추가적인 결과를 얻거나 또는 추가 작업을 할 수 있습니다. 그러

면 더하기 함수를 실행했을 때 그 기록을 남겨 두었다가 출력하는 기능을 가상으로 만들어 보겠습니다. ch03_test16.js 파일을 만든 후 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch03_test16.js

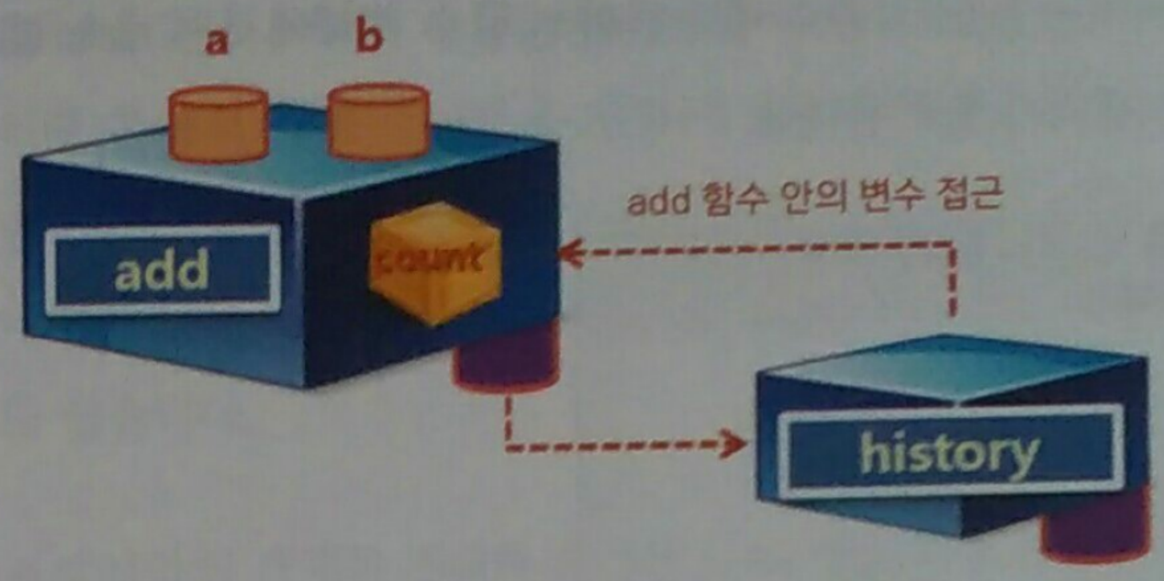
```
function add(a, b, callback) {
  var result = a + b;
  callback(result);

  var history = function() {
    return a + ' + ' + b + ' = ' + result;
  };
  return history;
}

var add_history = add(10, 10, function(result) {
  console.log('파라미터로 전달된 콜백 함수 호출됨.');
  console.log('더하기 (10, 10)의 결과 : %d', result);
});

console.log('결과 값으로 받은 함수 실행 결과 : ' + add_history())
```

여기에서는 add( ) 함수를 호출했을 때 콜백 함수를 호출할 뿐만 아니라 함수 객체를 새로 만들어 반환하도록 만들었습니다. 반환된 함수 객체는 어떤 연산이 수행되었는지 문자열로 알려 주는 함수입니다. add() 함수를 실행한 후 반환받은 함수 객체를 add_history 변수에 할당한 후 그 함수를 실행하였습니다. 파일을 실행하면 다음과 같은 결과가 콘솔 창에 출력됩니다.



▲ 반환받은 함수 실행하기

반환받은 history 객체가 함수이고 이 함수를 실행했을 때 어떤 값으로 더하기 연산을 했는지 알 수 있도록 만들었습니다. 이렇게 함수의 실행 결과로 함수 객체를 받을 수 있다는 것을 잘 이해했다면 반환된 함수에서 접근할 수 있는 변수에 대해서도 알아볼 필요가 있습니다. 반환된 함수는 보통 처음 실행한 함수 안에서 만들어지는데 그 안에 있는 변수들은 함수가 반환된 후에도 계속 접근할 수 있습니다.




```
function add(a, b, callback) {
    var result = 0;

    var history = function() {
        count++;
        return count + ':' + a + ' + ' + b + ' = ' + result;
    };
    return history;
}
```

▲ 반환된 함수 객체에서 함수를 만들어 준 add 함수 안의 변수에 접근하기

add 함수 안에 history 함수를 몇 번이나 실행했는지 알 수 있도록 count 변수를 만들었다면 이 변수는 add 함수에서 반환된 함수 안에서 계속 접근할 수 있습니다. 다음과 같이 ch03_test17.js 파일을 만들고 코드를 입력합니다.

참조 파일 | NodeExample1>ch03_test17.js

```
function add(a, b, callback) {
  var result = a + b;
  callback(result);

  var count = 0;
  var history = function() {
    count++;
    return count + ' : ' + a + ' + ' + b + ' = ' + result;
  };
  return history;
}

var add_history = add(10, 10, function(result) {
  console.log('파라미터로 전달된 콜백 함수 호출됨.');
  console.log('더하기 (10, 10)의 결과 : %d', result);
});

console.log('결과 값으로 받은 함수 실행 결과 : ' + add_history());
console.log('결과 값으로 받은 함수 실행 결과 : ' + add_history());
console.log('결과 값으로 받은 함수 실행 결과 : ' + add_history());
```

count 변수는 add( ) 함수 안에서 만들어진 것이지만 add( ) 함수에서 반환된 함수 객체에서는 계속 접근할 수 있습니다. 파일을 실행하면 다음과 같은 결과가 출력됩니다.

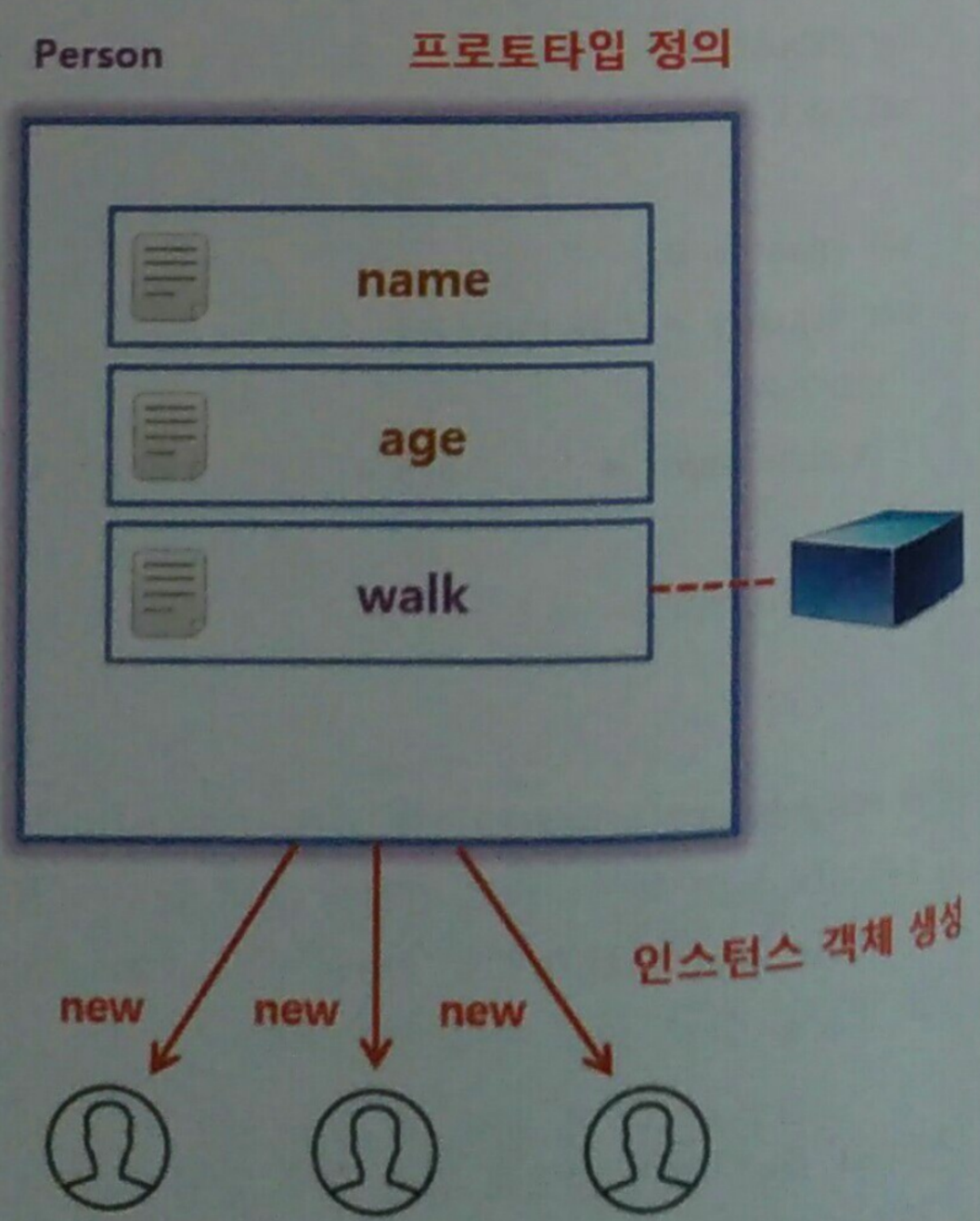
```
Console  Debug
<terminated> NodeExample1-ch03_test17.js [Node Application] Node.js Process
파라미터로 전달된 콜백 함수 호출됨.
더하기 (10, 10)의 결과 : 20
결과값으로 받은 함수 실행 결과 : 1 : 10 + 10 = 20
결과값으로 받은 함수 실행 결과 : 2 : 10 + 10 = 20
결과값으로 받은 함수 실행 결과 : 3 : 10 + 10 = 20
```

▲ 반환된 함수 객체에서 더하기 함수의 변수를 계속 접근한 결과

반환된 함수를 반복해서 실행하면 count 변수의 값이 증가하는 것을 볼 수 있습니다. 이 내용을 주의해서 보세요. 반환된 history 함수가 실행될 때는 이미 add 함수가 메모리에서 접근할 수 없는 상태가 된 다음입니다. 따라서 history 함수가 실행될 때는 add 함수 안에 만들어진 count 변수도 같이 접근할 수 없어야 합니다. 하지만 이와 같이 함수 안에서 새로운 함수를 만들어 반환하는 경우에는 예외적으로 변수 접근을 허용합니다. 이것을 **클로저(Closure)**라고 부릅니다. 여러분은 이 용어가 어떤 의미인지 지금 꼭 알 필요는 없습니다. 위에서 만들어 본 코드가 어떻게 동작하는지만 이해해도 충분합니다.

03-4 | 프로토타입 객체 만들기

자바스크립트의 객체를 만들 때는 중괄호를 이용한다고 했습니다. 그런데 자바스크립트 객체는 함수를 이용해서 만들 수도 있습니다. 그 이유는 함수도 객체이기 때문입니다. 함수에 여러 가지 기능과 속성이 추가되면서 **객체 지향(Object oriented)** 언어에서 **객체의 원형(Prototype)**인 클래스를 만들고, 그 클래스에서 새로운 인스턴스 객체를 여러 개 만들어 내듯이 자바스크립트에서도 객체의 원형을 정의한 후 그 원형에서 새로운 인스턴스 객체를 만들어 낼 수 있습니다.



프로토타입 객체를 정의한 후 프로토타입에서 인스턴스 객체 만들기 ▶

Person 프로토타입 객체는 데이터를 넣어 두려는 목적보다는 하나의 틀로 사용하기 위해 만듭니다. 붕어빵 틀에서 실제 붕어빵을 만들어 내듯이 실제 사람을 여러 명 찍어 내기 위한 틀을 프로토타입으로 만든다고 생각하면 이해하기 쉽습니다. 이 Person 프로토타입에는 사람의 이름을 담는 name 속성과 나이를 담는 age 속성이 들어갈 수 있습니다. 그리고 walk 함수를 넣어서 걸어가는 동작을 수행할 수도 있습니다.

프로토타입 객체를 정의하고 나면 new 연산자를 사용하여 인스턴스 객체들을 만들 수 있습니다. ch03_test18.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch03_test18.js

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

Person.prototype.walk = function(speed) {
  console.log(speed + 'km 속도로 걸어갑니다.');
}

var person01 = new Person('소녀시대', 20);
var person02 = new Person('걸스데이', 22);

console.log(person01.name + '객체의 walk(10)을 호출합니다.');
person01.walk(10);
```

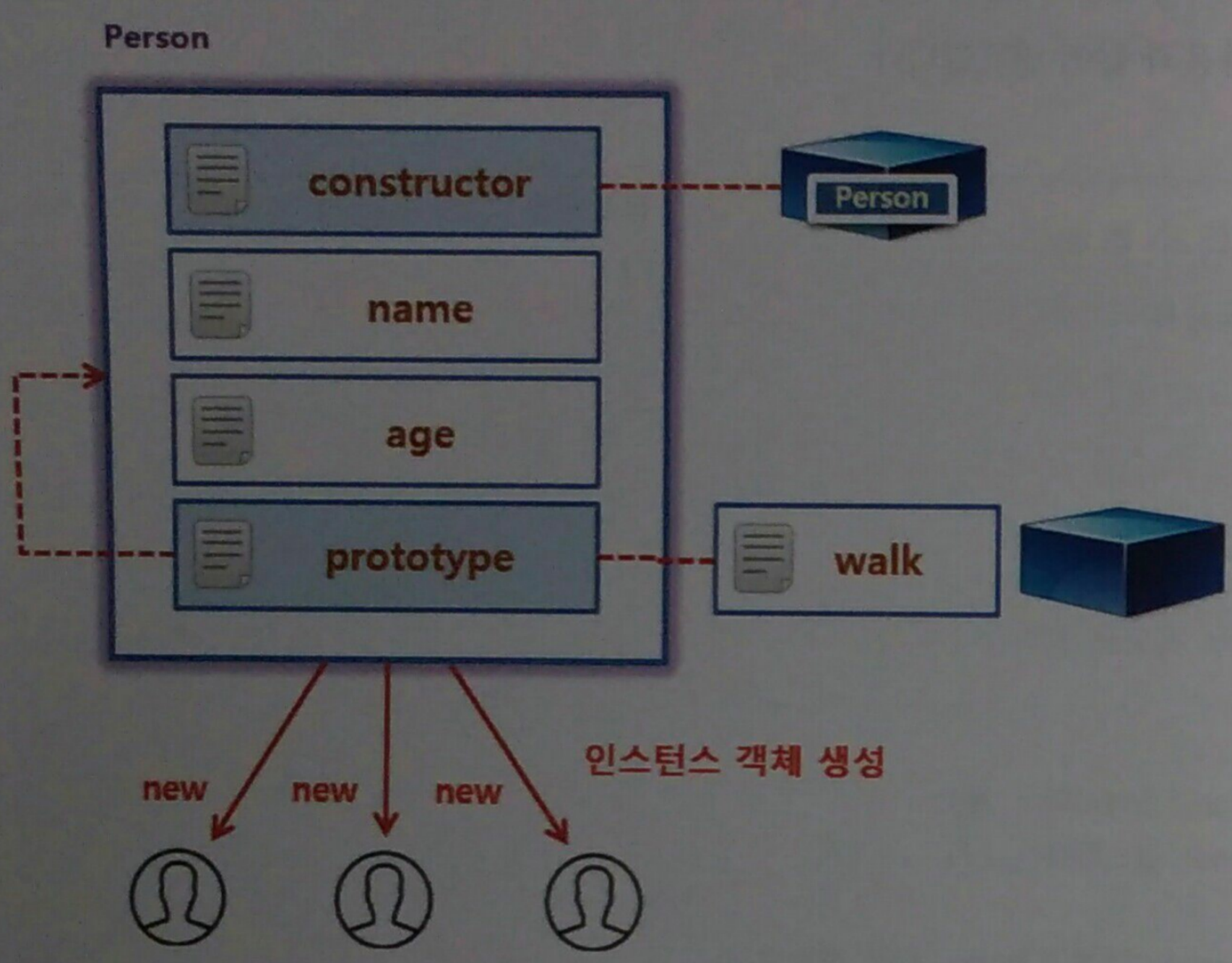
Person 프로토타입 객체를 만들려면 먼저 Person 함수를 정의합니다. 자바스크립트에서는 함수도 객체이기 때문에 이 함수는 객체의 역할을 할 수 있습니다. 즉, 함수 중에서 new 연산자로 호출되는 함수는 객체를 만들기 위한 함수로 분류되며, 이러한 함수를 **생성자(Constructor)**라고 합니다. 따라서 그 객체 안에 name 속성을 만들거나 name 속성에 접근하고 싶다면 this를 사용합니다. 함수의 파라미터로 전달받은 name과 age 속성을 이용해서 객체인 this에 name과 age 속성을 추가합니다.

주의! 여기에서 만드는 객체는 실제 객체로 만들어 사용하기 전 단계인 프로토타입이라는 것에 주의하세요.

정박사의 한마디 this는 함수를 호출한 객체입니다

자바스크립트에서 객체 지향 방식으로 코드를 구성하기 위해 생성자 함수로 객체를 만들면, 그 객체 안에서 사용하는 this 키워드는 그 함수를 호출하는 객체를 가리킵니다. 이에 대해 좀 더 자세히 알고 싶다면 자바스크립트 입문서를 참조하기 바랍니다.

Person 객체는 name과 age 속성을 갖게 되었습니다. 이제 이 객체에 walk 함수를 속성으로 추가하고 싶다면 Person.walk = function( ) {...}과 같은 형태가 아니라 Person.prototype.walk = function( ) {...}과 같은 형태로 만듭니다. 이것은 Person 객체가 실제 데이터를 담기 위해 만들어진 게 아니라 다른 인스턴스 객체를 만들기 위한 원형 틀로 만들어졌기 때문입니다. Person 객체 안에 있는 prototype 속성에 데이터나 함수를 속성으로 추가하면 실제 인스턴스 객체를 만들 때 메모리를 효율적으로 관리할 수 있습니다. 다음 그림은 Person 객체에서 prototype 속성을 사용하는 방식을 설명합니다.



▲ Person 객체의 prototype 속성 사용하기

Person 객체 안에 있는 prototype 속성도 객체로 만든 것인데, Person 함수가 만들어질 때 자동으로 만들어집니다. 이 prototype 객체는 Person 객체 자신을 가리키도록 되어 있어 다음의 두 코드는 같은 결과를 보여줍니다.

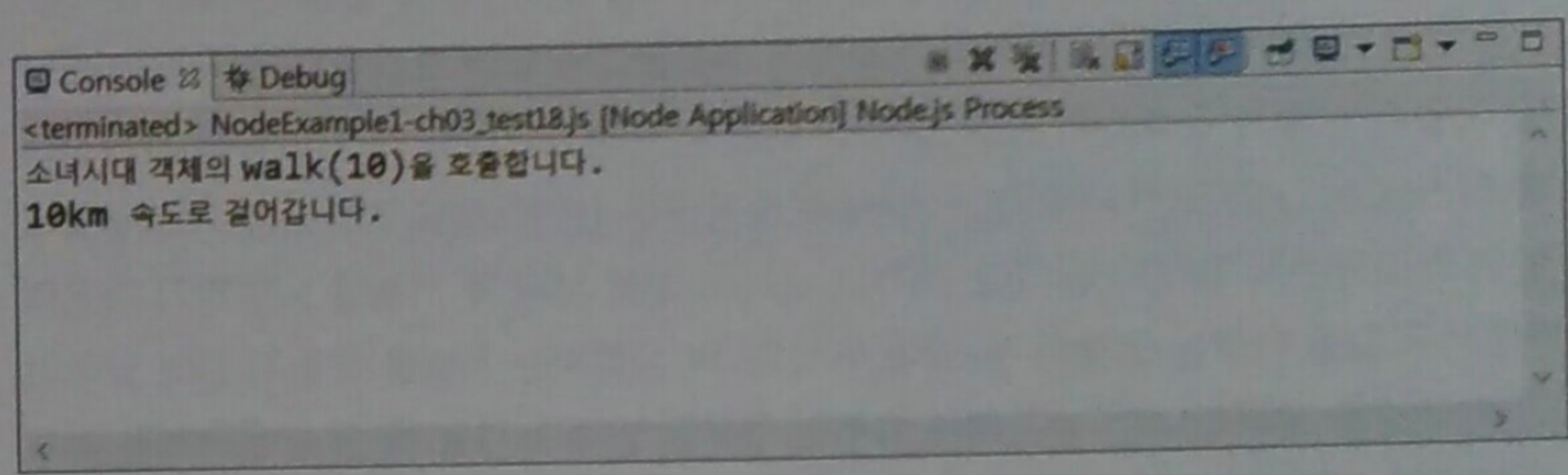
```
Person.walk = function() { ... }
Person.prototype.walk = function() { ... }
```

다시 말해, 이 두 가지 코드 모두 Person이라는 틀로 만들어 낸 인스턴스 객체에서 walk 함수를 호출할 수 있습니다. 다만 prototype 속성으로 추가하면 인스턴스 객체를 만들 때 메모리를 효율적으로 관리할 수 있다는 점이 서로 다릅니다.

그럼 prototype 객체는 언제 만들어진 것일까요? 여러분이 prototype 객체를 추가하지 않더라도 함수 객체를 만들 때 함수 안에 prototype이나 constructor와 같은 객체들이 자동으로 만들어집니다.

constructor 객체는 생성자라고 하는데 new 연산자로 인스턴스 객체를 만들 때 사용할 수 있으며, 필요한 파라미터를 전달받아 객체의 속성에 할당하거나 필요한 초기화 함수를 실행할 수 있습니다.

ch03_test18.js 파일을 실행하면 다음과 같이 두 개의 Person 인스턴스 객체를 만든 후 person01 변수에 할당된 첫 번째 인스턴스 객체의 walk( ) 함수를 호출한 결과를 볼 수 있습니다.



▲ 프로토타입 객체에서 만들어 낸 인스턴스 객체의 함수를 호출한 결과

새로운 인스턴스 객체들을 만들도록 정의한 Person 객체를 **프로토타입 객체**라고도 부르는데, 일반적인 함수와 구별해서 이해할 수 있어야 합니다.

지금까지 노드에서 자주 사용하는 자바스크립트의 개념 및 코드를 알아보았습니다. 자바스크립트와 친해지기 위한 내용을 대략적으로 설명한 것인데 조금 정리가 되었나요? 웹 브라우저에서 웹 문서를 만들 때 사용하는 자바스크립트와 노드에서 사용하는 자바스크립트의 코드 패턴이 약간 다를 수 있다는 점을 잊지 마세요. 다음 장에서는 가장 기본이 되는 노드의 기능을 알아보겠습니다.

04 노드의 기본 기능 알아보기

노드로 프로그램을 만들려면 몇 가지 기본적인 내용을 알아두면 좋습니다. 이벤트가 무엇인지, 파일은 어떻게 다루는지 등을 알아야 노드로 프로그래밍할 때 두려움이 사라집니다. 특히 노드는 프로그램이 이벤트 기반으로 동작하는 언어이므로 이벤트를 어떻게 보내고 받아 처리하는지 알아야 합니다. 그리고 노드는 주로 서버 쪽 기능을 담당하기 때문에 서버에서 파일을 다루는 방법도 잘 알아 두세요. 이 장에서는 서버를 만들기 전에 알아야 할 기본 내용을 살펴봅니다. 누군가가 요청한 정보를 받아 처리할 때 주소 문자열을 다루는 방법부터 이벤트와 파일을 다루는 방법까지 하나씩 알아보겠습니다.

질문으로 시작하기

질문1. 웹 서버로 정보를 요청할 때 주소 문자열을 어떻게 다루나요?

➡ 04-1. 주소 문자열과 요청 파라미터 다루기

질문2. 노드에서 이벤트를 어떻게 처리하나요?

➡ 04-2. 이벤트 이해하기

질문3. 서버에서 처리할 일이 많은 파일은 어떻게 읽고 쓰나요?

➡ 04-3. 파일 다루기

질문4. 로그를 콘솔이 아닌 파일에 남기려면 어떻게 하나요?

➡ 04-4. 로그 파일 남기기

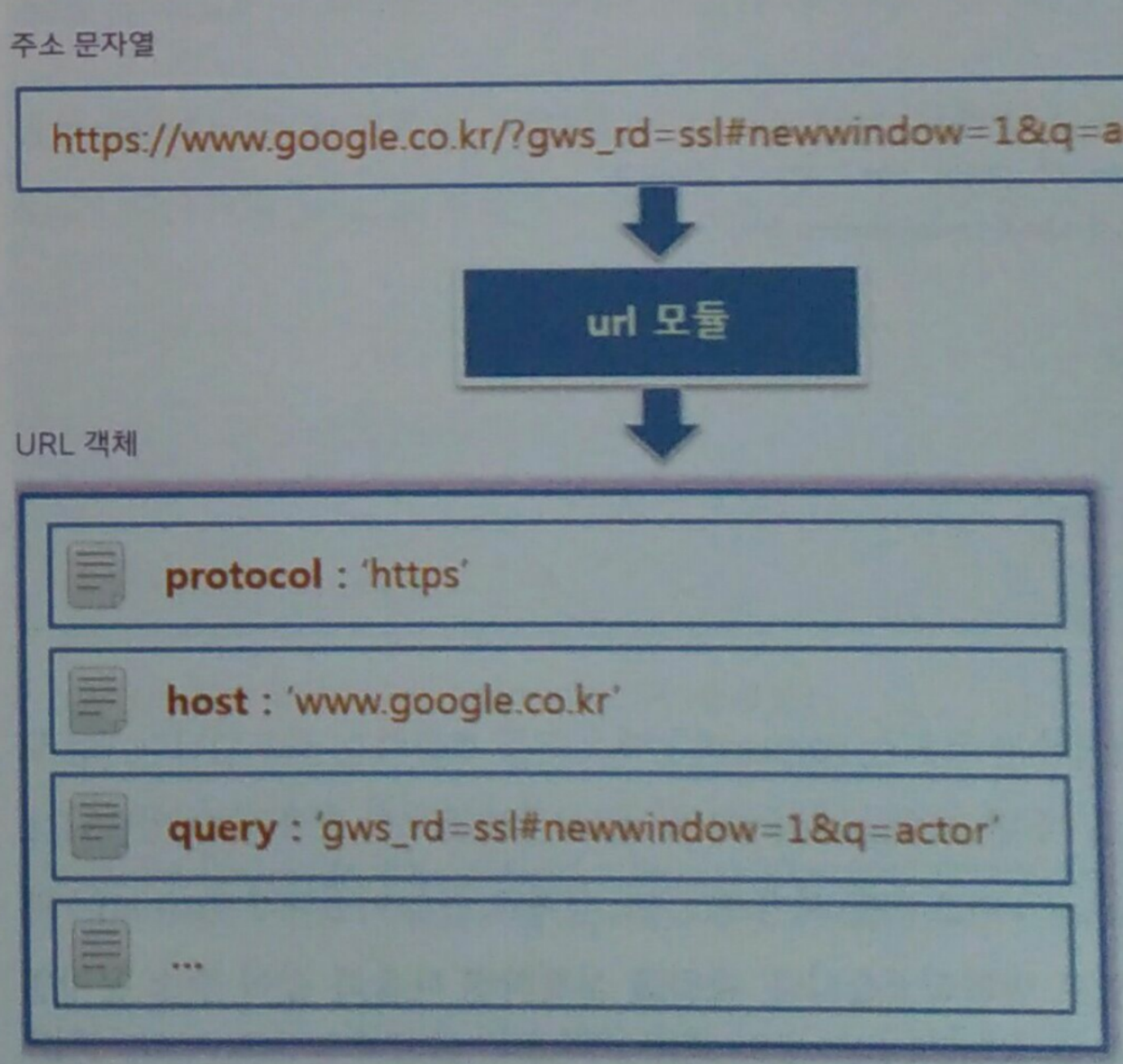
04-1 | 주소 문자열과 요청 파라미터 다루기

웹 사이트에 접속하기 위한 사이트 주소 정보는 노드에서 URL 객체로 만들 수 있습니다. 예를 들어, 구글 사이트에서 영화배우 정보를 찾기 위해 키워드로 actor를 넣고 [검색] 버튼을 누르면, 다음과 같은 주소 문자열을 만들어 검색 요청을 하게 됩니다.

```
▶ https://www.google.co.kr/?gws_rd=ssl#newwindow=1&q=actor
```

그런데 이렇게 만들어진 주소 문자열은 단순 문자열이므로 서버에서 이 정보를 받아 처리할 때는 어디까지가 사이트 주소인지, 그리고 어떤 내용이 요청 파라미터인지 구별해야 합니다. 이 구별을 위해서 ? 기호를 기준으로 앞에 있는 문자열과 뒤에 있는 문자열을 분리하는 경우가 많습니다. 이 작업을 쉽게 할 수 있도록 노드에 미리 만들어 둔 모듈이 **url 모듈**입니다.

url 모듈을 사용하면 일반 주소 문자열을 URL 객체로 만들거나 또는 URL 객체에서 일반 문자열로 변환하는 일이 쉬워집니다.



▲ url 모듈을 이용한 주소 문자열과 URL 객체 간 변환

url 모듈을 이용해 주소 문자열을 객체로 만들면 문자열 안에 있던 각각의 정보를 나누어 그 객체의 속성으로 보관합니다. 따라서 요청 프로토콜이 http인지 아니면 https인지를 구별하거나 요청 파라미터를 확인하고 싶다면 url 객체가 갖고 있는 속성 값을 확인하면 됩니다.

주소 문자열을 URL 객체로 변환하기

url 모듈에서 문자열을 객체로 만들거나 객체를 문자열로 만들기 위해 사용하는 주요 메소드는 다음과 같습니다.

메소드 이름	설명
parse( )	주소 문자열을 파싱하여 URL 객체를 만들어 줍니다.
format( )	URL 객체를 주소 문자열로 변환합니다.

[표] url 모듈의 주요 메소드

그러면 네이버 모바일 사이트에서 steve jobs를 검색할 때 사용되는 주소 문자열을 URL 객체로 만들거나 URL 객체에서 다시 원래의 주소 문자열로 바꾸는 방법을 간단하게 확인해 보겠습니다.

ch04_test1.js 파일을 새로 만들고 다음과 같이 url 모듈을 사용하는 코드를 입력합니다.

참조 파일 | NodeExample1>ch04_test1.js

```
var url = require('url');

// 주소 문자열을 URL 객체로 만들기
var curURL = url.parse('https://m.search.naver.com/search.naver?
                        query=steve+jobs&where=m&sm=mtp_hty');

// URL 객체를 주소 문자열로 만들기
var curStr = url.format(curURL);

console.log('주소 문자열 : %s', curStr);
console.dir(curURL);
```

url 모듈을 사용하기 위해 가장 처음 입력하는 코드는 require( ) 메소드를 호출하는 코드입니다. url 모듈을 로딩한 후 url 변수에 할당했으므로 이 변수의 parse( )와 format( ) 메소드를 호출할 수 있습니다. https://로 시작하는 주소 문자열은 parse( ) 메소드를 사용해 URL 객체로 만들어졌다가 format( ) 메소드를 사용해 다시 원래의 주소 문자열로 변환되었습니다. 파일을 실행하면 다음과 같이 주소 문자열 정보와 함께 URL 객체에 들어 있는 속성이 출력됩니다.

```
Console  Debug
<terminated> NodeExample1-ch04_test1.js [Node Application] Node.js Process
주소 문자열 : https://m.search.naver.com/search.naver?query=steve+jobs&wher
{ protocol: 'https:',
  slashes: true,
  auth: null,
  host: 'm.search.naver.com',
  port: null,
  hostname: 'm.search.naver.com',
  hash: null,
  search: '?query=steve+jobs&where=m&sm=mtp_hty',
  query: 'query=steve+jobs&where=m&sm=mtp_hty',
  pathname: '/search.naver',
  path: '/search.naver?query=steve+jobs&where=m&sm=mtp_hty',
  href: 'https://m.search.naver.com/search.naver?query=steve+jobs&where
```

▲ 주소 문자열을 URL 객체로 변환한 결과

요청 파라미터 확인하기

URL 객체의 속성을 보면 주소 문자열의 여러 가지 정보가 포함되어 있습니다. 그중에서도 query 속성은 요청 파라미터 정보를 가지고 있는데 이 정보를 잘 살펴보면 여러 개의 요청 파라미터가 모두 들어 있습니다. 웹 서버에서는 클라이언트에서 요청한 요청 파라미터를 받아 처리할 때가 많으므로 이 query 속성에 들어 있는 문자열을 다시 각각의 요청 파라미터로 분리해야 합니다.

요청 파라미터는 & 기호로 구분되는데 **querystring 모듈**을 사용하면 요청 파라미터를 쉽게 분리할 수 있습니다. ch04_test1.js 파일에 다음 코드를 추가로 입력합니다.

참조 파일 | NodeExample1>ch04_test1.js

```
.....

// 요청 파라미터 구분하기
var querystring = require('querystring');
var param = querystring.parse(curURL.query);

console.log('요청 파라미터 중 query의 값 : %s', param.query);
console.log('원본 요청 파라미터 : %s', querystring.stringify(param));
```

먼저 querystring 모듈을 사용하기 위해 require( ) 메소드를 호출합니다. 불러온 모듈은 querystring 변수에 할당하고 그 객체의 parse( ) 메소드를 호출합니다. parse( ) 메소드는 요청 파라미터 문자열을 파싱하여 객체로 만들어 줍니다. 이 객체에는 각각의 요청 파라미터들이 속성으로 추가되어 있어 속성 값을 참조하면 각각의 요청 파라미터를 사용할 수 있습니다. stringify( ) 메소드는 객체 안에 들어 있는 요청 파라미터를 다시 하나의 문자열로 바꿀 때 사용합니다.

메소드 이름	설명
parse( )	요청 파라미터 문자열을 파싱하여 요청 파라미터 객체로 만들어 줍니다.
stringify( )	요청 파라미터 객체를 문자열로 변환합니다.

[표] querystring 모듈의 주요 메소드

파일을 다시 실행하면 steve jobs 문자열이 query 속성에 들어 있는 것을 확인할 수 있습니다.

```
Console  Debug
<terminated> NodeExample1-ch04_test1.js [Node Application] Node.js Process
  port: null,
  hostname: 'm.search.naver.com',
  hash: null,
  search: '?query=steve+jobs&where=m&sm=mtp_hty',
  query: 'query=steve+jobs&where=m&sm=mtp_hty',
  pathname: '/search.naver',
  path: '/search.naver?query=steve+jobs&where=m&sm=mtp_hty',
  href: 'https://m.search.naver.com/search.naver?query=steve+jobs&where
요청 파라미터 중 query의 값 : steve jobs
원본 요청 파라미터 : query=steve%20jobs&where=m&sm=mtp_hty
```

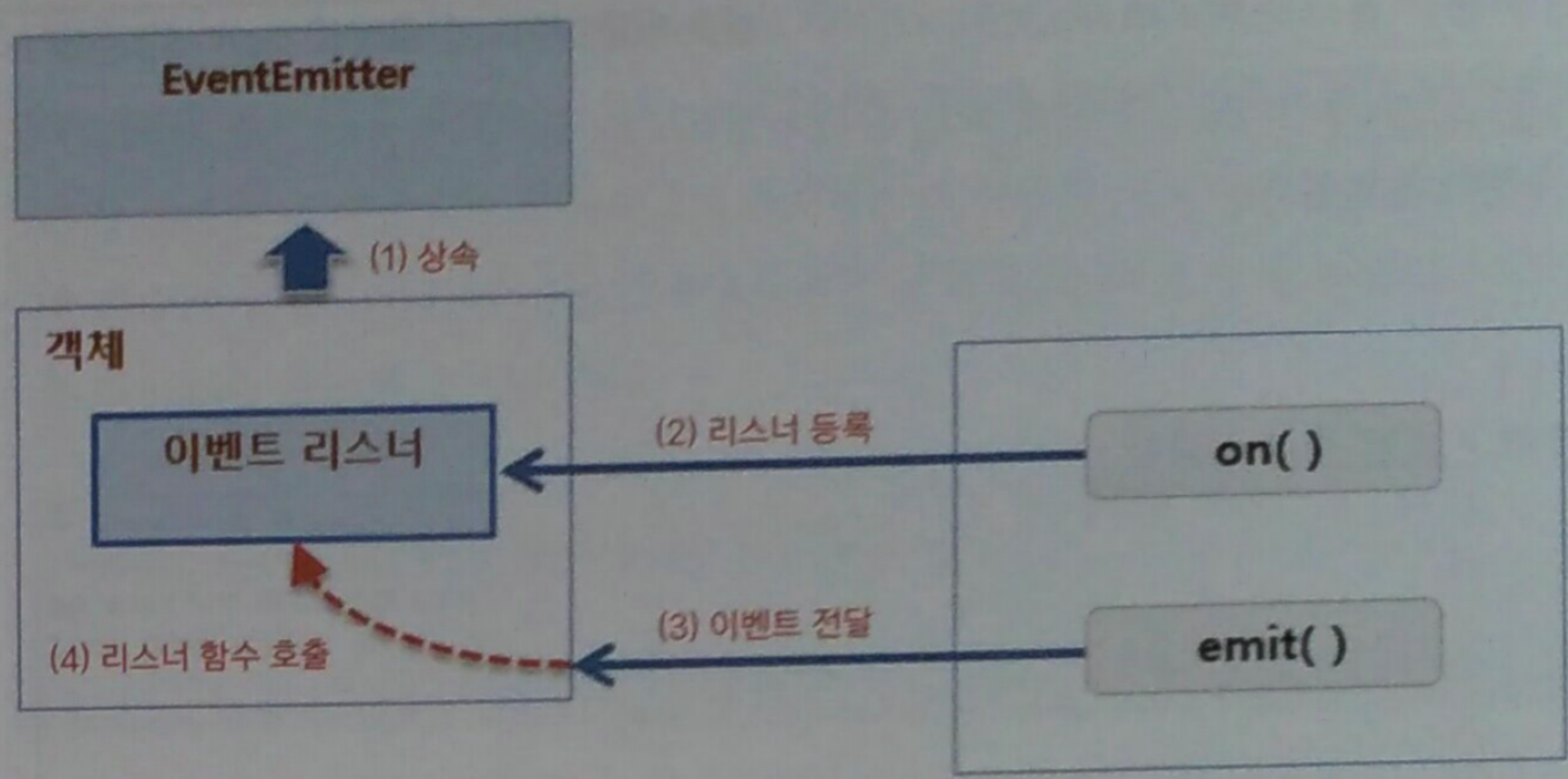
▲ querystring 모듈을 사용한 요청 파라미터 파싱

지금까지 입력한 몇 줄의 코드만 보더라도 노드에서 제공하는 기본 모듈을 사용하면 주소 문자열에 들어 있는 여러 가지 정보를 쉽게 분리할 수 있음을 알 수 있습니다.

04-2 | 이벤트 이해하기

노드는 대부분 이벤트를 기반으로 하는 비동기 방식으로 처리합니다. 그리고 비동기 방식으로 처리하기 위해 서로 이벤트를 전달합니다. 예를 들어, 어떤 함수를 실행한 결과물도 이벤트로 전달합니다. 이벤트는 한쪽에서 다른 쪽으로 알림 메시지를 보내는 것과 비슷합니다. 즉, '지금 이쪽의 상태는 이렇다'는 정보를 다른 쪽으로 보내는 것입니다. 노드에는 이런 이벤트를 보내고 받을 수 있도록 EventEmitter라는 것이 만들어져 있습니다. 다음은 EventEmitter를 사용해 이벤트를 주고받는 방법을 보여 줍니다.

용어! **이벤트는** 한쪽에서 다른 쪽으로 어떤 일이 발생했음을 알려주는 것입니다. 이때 다른 쪽에서 이 이벤트를 받고 싶다면 **이벤트 리스너(Event Listener)**를 등록할 수 있습니다. 이벤트 리스너는 특정 이벤트가 전달되었을 때 그 이벤트를 처리할 수 있도록 만들어 둔 것을 말합니다.



▲ 이벤트를 주고받는 방법

이벤트 보내고 받기

노드의 객체는 EventEmitter를 상속받을 수 있으며, 상속받은 후에는 EventEmitter 객체의 on( )과 emit( ) 메소드를 사용할 수 있습니다. on( ) 메소드는 이벤트가 전달될 객체에 이벤트 리스너를 설정하는 역할을 하는데 이 리스너 함수는 객체로 전달된 이벤트를 받아 처리할 수 있습니다. 보통은 노드 내부에서 미리 만들어 제공하는 이벤트를 받아 처리하지만, 필요할 때는 직접 이벤트를 만들어 전달할 수도 있습니다.

on( ) 메소드 이외에 once( ) 메소드를 사용할 수도 있습니다. once( ) 메소드를 사용할 때는 이벤트 리스너 함수가 한 번이라도 실행하고 나면 자동으로 제거되므로 이벤트를 딱 한 번만 받아서 처리할 수 있습니다.

이벤트를 다른 쪽으로 전달하고 싶다면 emit( ) 메소드를 사용합니다.

메소드 이름	설명
on(event, listener)	지정한 이벤트의 리스너를 추가합니다.
once(event, listener)	지정한 이벤트의 리스너를 추가하지만 한 번 실행한 후에는 자동으로 리스너가 제거됩니다.
removeListener(event, listener)	지정한 이벤트에 대한 리스너를 제거합니다.

[표] 이벤트를 처리하는 EventEmitter의 주요 메소드

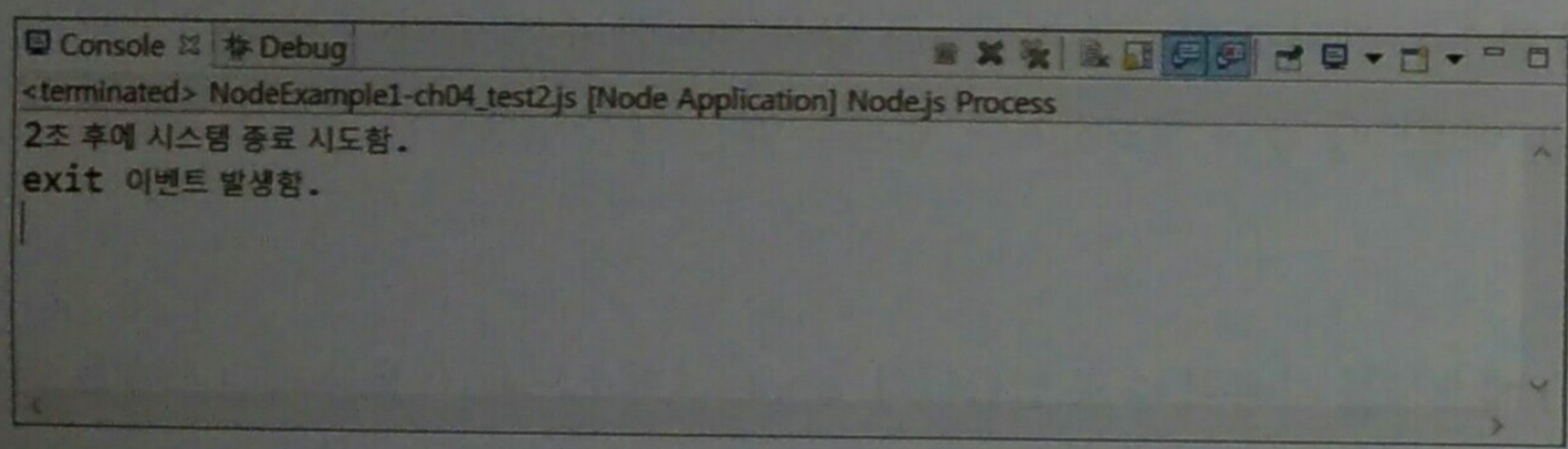
ch04_test2.js 파일을 새로 추가한 후 다음 코드를 입력합니다.

참조 파일 | NodeExample1>ch04_test2.js

```
process.on('exit', function() {
  console.log('exit 이벤트 발생함.');
});

setTimeout(function() {
  console.log('2초 후에 시스템 종료 시도함.');

  process.exit();
}, 2000);
```

process 객체는 노드에서 언제든지 사용할 수 있는 객체인데 이미 내부적으로 EventEmitter를 상속받도록 만들어져 있어서 on( )과 emit( ) 메소드를 바로 사용할 수 있습니다.

process 객체의 on( ) 메소드를 호출하면서 이벤트 이름을 exit로 지정하면 프로세스가 끝날 때를 알 수 있습니다. 그 아래 코드에서는 setTimeout( ) 메소드를 호출하여 2초 후 프로그램을 끝냅니다. 이 프로그램을 실행하면 콘솔 창에 다음과 같이 출력됩니다.



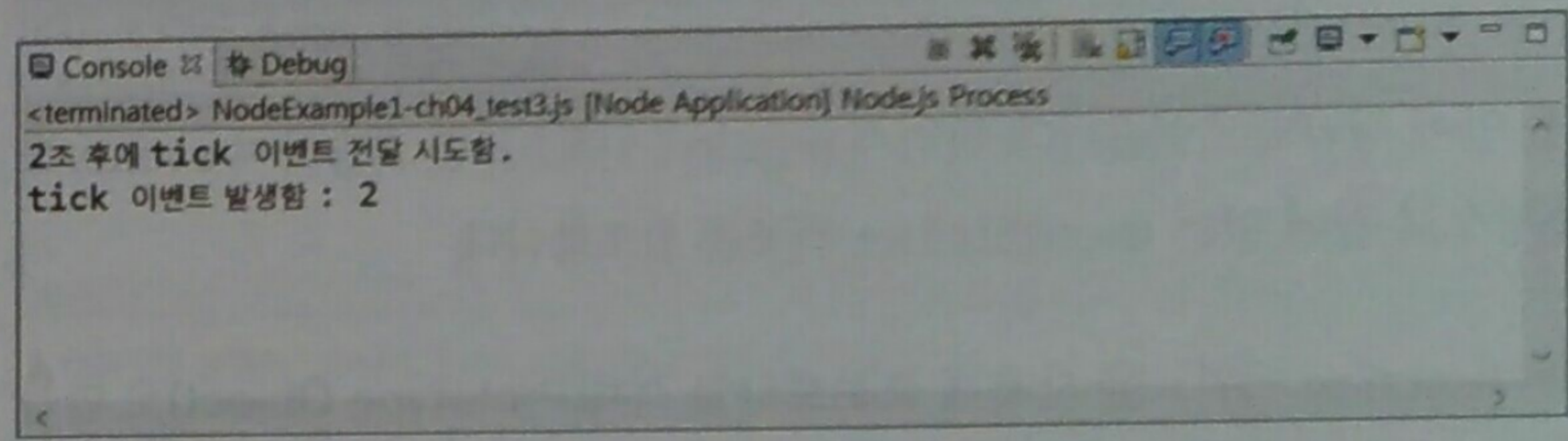
▲ 프로그램 종료 이벤트를 전달받아 처리한 결과

그렇다면 미리 정의되어 있는 이벤트가 아니라 우리가 직접 만든 이벤트는 어떻게 처리할 수 있을까요? 앞에서 입력한 것과 비슷한 방식으로 다음 코드를 입력합니다.

참조 파일 | NodeExample1>ch04_test3.js

```
process.on('tick', function(count) {
  console.log('tick 이벤트 발생함 : %s', count);
});

setTimeout(function() {
  console.log('2초 후에 tick 이벤트 전달 시도함.');

  process.emit('tick', '2');
}, 2000);
```

이번에는 tick 이벤트를 직접 만들고 2초 후에 setTimeout( ) 메소드를 사용해 process.emit( ) 메소드를 호출하면서 tick 이벤트를 process 객체로 전달했습니다. process.on( ) 메소드를 호출하여 이벤트를 등록하면 이 메소드를 호출하면서 파라미터로 전달한 tick 이벤트가 발생했을 때 그다음에 나오는 콜백 함수가 실행됩니다. 이 파일을 실행하면 다음과 같이 콘솔 창에 출력되는 문자열을 통해 tick 이벤트를 받았는지 확인할 수 있습니다.



▲ 직접 지정한 이벤트를 보낸 후 전달받아 처리한 결과

계산기 객체를 모듈로 만들어 보기

지금까지 실행해 본 코드만으로도 한쪽에서 이벤트를 만들어 전달할 수 있고, 다른 쪽에서 그것을 받아 처리할 수 있다는 것을 알 수 있었습니다. 그런데 항상 process 객체를 사용해 이벤트를 전달한다면, 같은 이름의 이벤트를 사용하는 경우에 충돌이 생길 수도 있습니다. 이 때문에 별도의 모듈 파일을 만들고 그 안에서 이벤트를 처리하도록 만드는 것이 좋습니다. 그러면 앞에서 다른 모듈 구성 방법을 잘 생각하면서 새로운 모듈을 만들어 보겠습니다.

먼저 계산기 객체를 모듈로 만들기 위해 calc3.js 파일에 다음 코드를 입력합니다.

참조 파일 | NodeExample1>calc3js

```
var util = require('util');
var EventEmitter = require('events').EventEmitter;

var Calc = function() {
  var self = this;

  this.on('stop', function() {
    console.log('Calc에 stop event 전달됨.');
  });
};

util.inherits(Calc, EventEmitter);

Calc.prototype.add = function(a, b) {
```

```
  return a + b;
}

module.exports = Calc;
module.exports.title = 'calculator';
```

EventEmitter는 events 모듈 안에 정의되어 있습니다. 따라서 require( ) 메소드를 호출하여 events 모듈을 불러들인 후 그 안에 속성으로 들어 있는 EventEmitter 객체를 참조합니다.

Calc 객체는 계산기 객체로서 function 키워드를 사용해 **프로토타입 객체(Prototype Object)**로 만듭니다. 프로토타입 객체 안에서는 this 키워드를 이용해 자기 자신을 가리킬 수 있으며, 그 객체 안에 정의된 속성에 접근할 수 있습니다. 그런 다음 Calc 객체가 이벤트 처리를 할 수 있도록 EventEmitter를 상속하도록 만듭니다. 상속은 util 모듈에 있는 inherits( ) 메소드를 사용하면 쉽게 정의할 수 있습니다. 코드의 가장 위쪽에서 require( ) 메소드를 호출하여 util 모듈을 불러왔기 때문에 util.inherits( ) 메소드를 호출하는 코드를 넣어도 오류는 발생하지 않습니다.

Calc 객체 안에 들어 있는 prototype 객체의 속성으로 add 함수를 추가하면 new 연산자를 이용해 Calc 객체의 인스턴스 객체를 만들었을 때 add( ) 함수를 사용할 수 있습니다. 코드의 마지막 부분에서는 calc3.js 파일에 정의한 모듈을 불러들이는 쪽에서 Calc 객체를 참조할 수 있도록 module.exports에 Calc 객체를 지정합니다. 추가적으로 title 속성의 값으로 calculator라는 이름을 설정합니다.

이 계산기 객체로 전달되는 stop 이벤트를 처리하기 위해 Calc 객체 안에서 on( ) 메소드를 호출하는 부분도 눈여겨볼 필요가 있습니다.

이렇게 만든 모듈을 사용하는 코드는 다음과 같이 별도의 파일에 입력합니다.

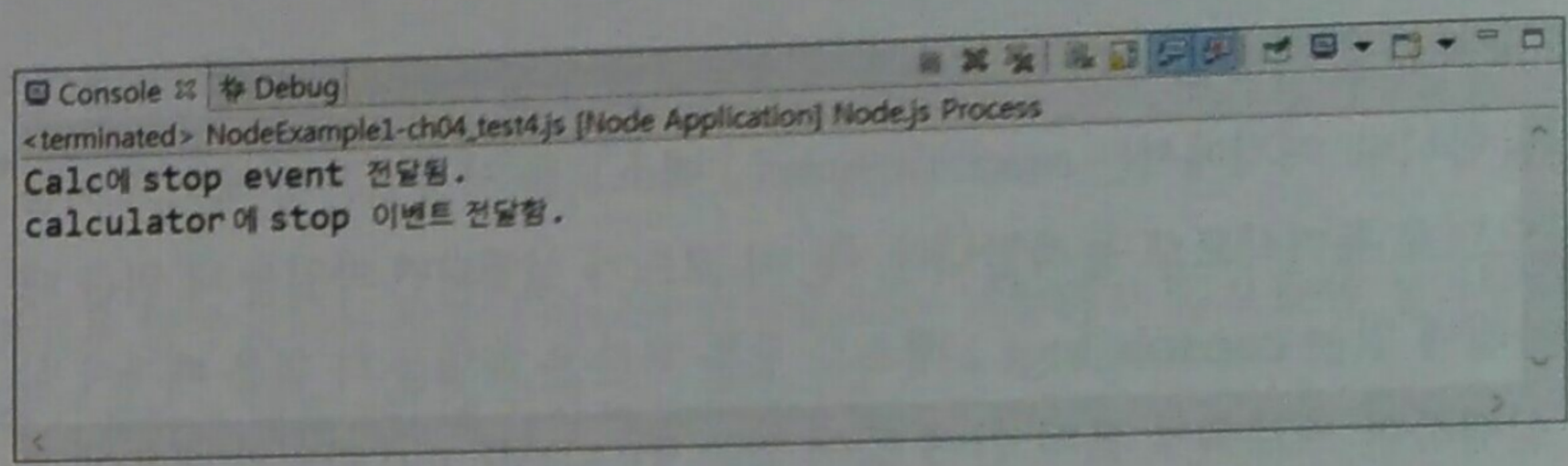
참조 파일 | NodeExample1>ch04_test4.js

```
var Calc = require('./calc3');

var calc = new Calc();
calc.emit('stop');

console.log(Calc.title + '에 stop 이벤트 전달함.');
```

require( ) 메소드를 호출하면서 ./calc3을 파라미터로 전달하면 별도의 모듈 파일 안에 정의해 둔 Calc 객체를 변수로 사용할 수 있습니다. Calc 객체는 프로토타입 객체로 계산기 기능을 정의만 한 것이므로

먼저 new 연산자를 이용해 인스턴스 객체를 만듭니다. Calc 객체가 EventEmitter를 상속하므로 인스턴스 객체의 emit( ) 메소드를 호출하여 stop 이벤트를 전달합니다.

이 파일을 실행하면 다음과 같이 직접 정의한 stop 이벤트가 정상적으로 전달된 것을 볼 수 있습니다.



▲ 직접 만든 모듈에 이벤트를 보내고 모듈 안에서 이벤트 처리하기

프로그램을 만들 때는 이벤트를 받아서 처리하는 경우가 많습니다. 그러므로 지금 만들어 본 코드에 들어 있는 on( )과 emit( ) 메소드에 대해 잘 알아두기 바랍니다.

04-3 | 파일 다루기

노드의 파일 시스템은 파일을 다루는 기능과 디렉터리를 다루는 기능으로 구성되어 있으며, 동기식 IO와 비동기식 IO 기능을 함께 제공합니다. 동기식 IO는 파일 작업이 끝날 때까지 대기하며, 비동기식 IO는 파일 작업을 요청만 하고 그다음 작업을 바로 수행합니다. 이후 파일 작업이 끝나면 그 상태는 이벤트로 받아서 처리합니다. **동기식 IO와 비동기식 IO**를 구별하기 위해 동기식 IO 메소드는 Sync라는 단어를 붙입니다.

주의! 동기식 방식(동기식 IO)을 사용할 때는 파일 처리가 끝날 때까지 대기하므로 처리 속도에 영향을 미칠 수 있습니다.

파일을 읽어 들이거나 파일에 쓰기

파일을 읽어 들이는 기능을 확인하기 위해 ch04_test5.js 파일을 만들고 다음 코드를 입력합니다.

참조 파일 | NodeExample1>ch04_test5.js

```
var fs = require('fs');

// 파일을 동기식 IO로 읽어 들입니다.
var data = fs.readFileSync('./README.md', 'utf8');
```

```
// 읽어 들인 데이터를 출력합니다.
console.log(data);
```

파일 시스템에 접근하기 위해 fs 모듈을 사용합니다. fs 모듈을 사용하려면 우선 require( ) 메소드를 호출하면서 fs를 파라미터로 전달합니다. 여기에서는 readFileSync( ) 메소드를 사용했는데 메소드 이름에 Sync라는 단어가 붙어 있으므로 동기식으로 동작합니다. 즉, 이 코드가 실행되면 파일을 다 읽을 때까지 대기합니다. 따라서 그 아래에 있는 console.log( ) 메소드 호출 부분은 파일을 다 읽을 때까지 실행되지 않습니다. 이 파일을 실행하면 프로젝트 폴더 안에 들어 있는 README.md 파일의 내용이 콘솔 창에 출력됩니다.

이번에는 README.md 파일을 읽어 들여 콘솔 창에 출력하는 기능은 같지만 비동기식 IO로 실행되는 코드를 ch04_test6.js 파일에 입력합니다.

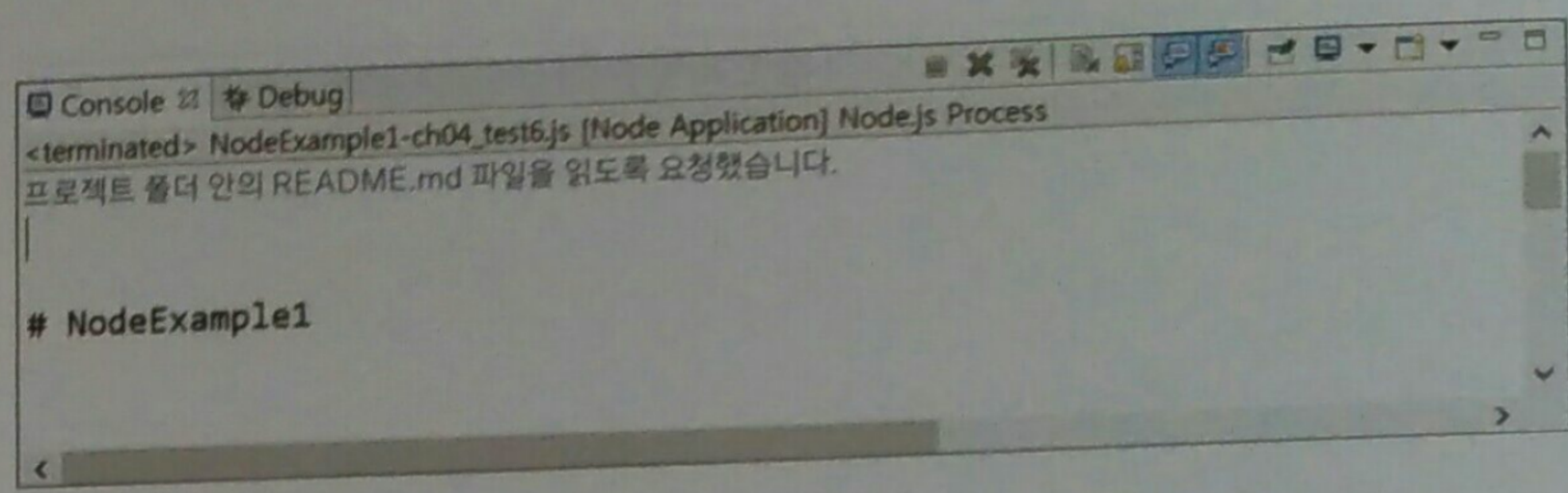
참조 파일 | NodeExample1>ch04_test6.js

```
var fs = require('fs');

// 파일을 비동기식 IO로 읽어 들입니다.
fs.readFile('./README.md', 'utf8', function(err, data) {
  // 읽어 들인 데이터를 출력합니다.
  console.log(data);
});

console.log('프로젝트 폴더 안의 README.md 파일을 읽도록 요청했습니다.');
```

여러 번 이야기했듯이 파일을 비동기식으로 읽어 들이는 방식이 노드에서 더 자주 사용하는 코드 패턴입니다. readFile( ) 메소드를 실행하면서 세 번째 파라미터로 전달된 함수(function)는 파일을 읽어 들이는 작업이 끝났을 때 호출됩니다. 이때 두 개의 파라미터 err와 data를 전달받아서 오류가 발생했는지 아니면 제대로 실행되었는지를 알 수 있습니다.

오류가 발생하면 err에 오류 데이터가 들어가고 그렇지 않으면 err 변수의 값이 null이 됩니다. 그러므로 일반적으로는 첫 번째 파라미터인 err가 null인지를 체크하는 코드를 사용한 후 문제가 없으면 파일 읽기에 성공한 것으로 처리합니다.

이 파일을 실행하면 readFile( ) 메소드를 먼저 호출했지만 그다음에 작성한 코드가 먼저 실행되는 것을 볼 수 있습니다.



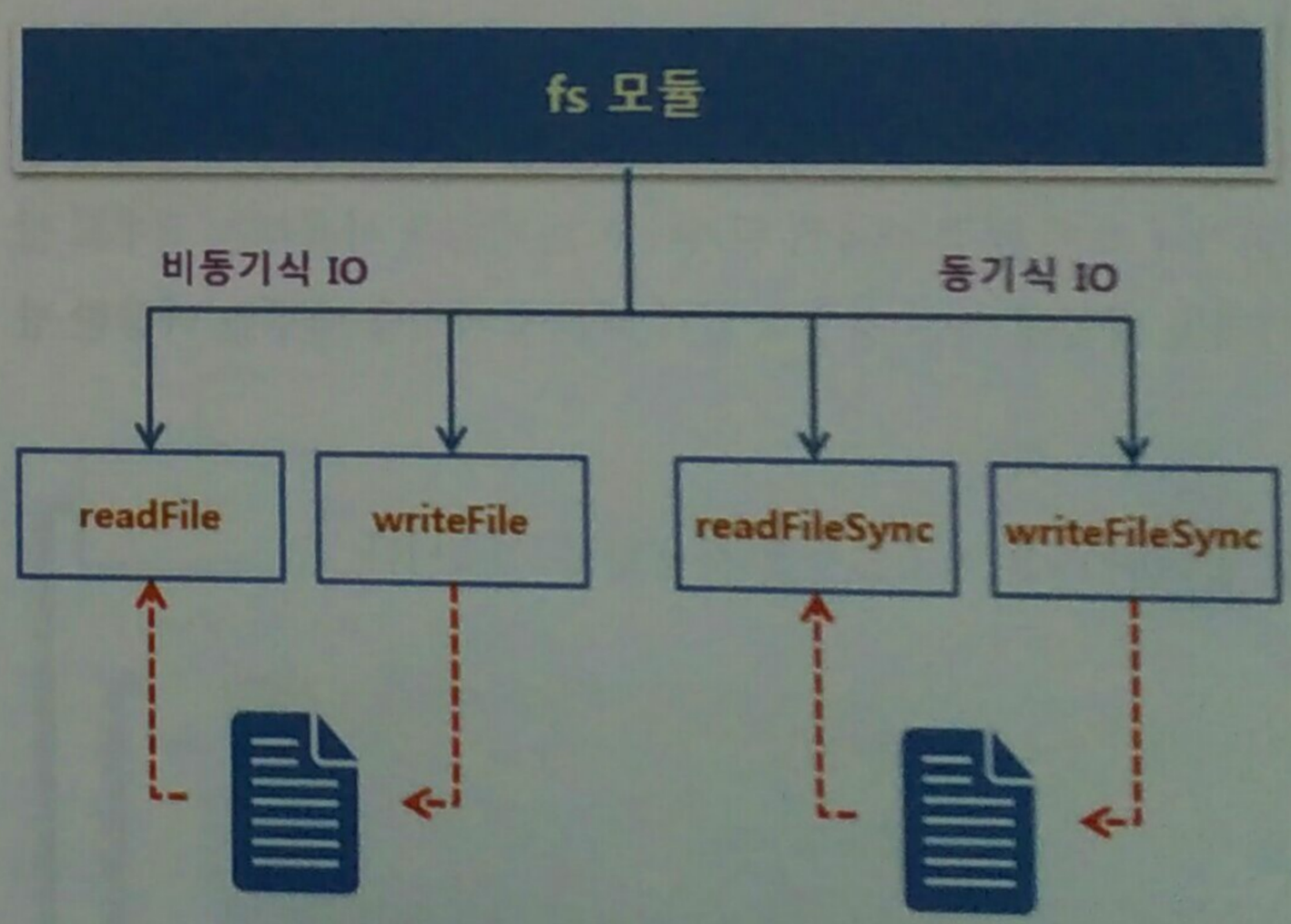
▲ 비동기식 IO로 파일을 읽어 내용을 출력한 경우

파일을 읽어 들이는 것뿐만 아니라 파일을 쓰는 기능까지 fs 모듈에 정의합니다. 다음은 파일을 읽고 쓸 때 사용하는 대표적인 네 가지 메소드입니다.

메소드 이름	설명
readFile(filename, [encoding], [callback])	비동기식 IO로 파일을 읽어 들입니다.
readFileSync(filename, [encoding])	동기식 IO로 파일을 읽어 들입니다.
writeFile(filename, data, encoding='utf8', [callback])	비동기식 IO로 파일을 씁니다.
writeFileSync(filename, data, encoding='utf8')	동기식 IO로 파일을 씁니다.

[표] 파일을 읽고 쓰는 대표적인 네 가지 메소드

대부분은 비동기 방식으로 파일을 읽고 쓰기 때문에 비동기 방식의 메소드를 주로 사용합니다.



▲ 파일을 읽고 쓰는 메소드의 구분

이번에는 파일에 데이터를 쓰는 기능을 만들기 위해 ch04_test7.js 파일을 만들고 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch04_test7.js

```
var fs = require('fs');

// 파일에 데이터를 씁니다.
fs.writeFile('./output.txt', 'Hello World!', function(err) {
  if(err) {
    console.log('Error : ' + err);
  }

  console.log('output.txt 파일에 데이터 쓰기 완료.');
});
```

파일에 데이터를 쓸 때 사용하는 writeFile( ) 메소드는 첫 번째 파라미터로서 파일 이름을 전달받습니다. 두 번째 파라미터는 파일에 쓸 내용이고 세 번째는 작업이 끝나면 호출될 콜백 함수입니다. 작업 중 오류가 발생하면 콜백 함수로 오류 객체가 전달됩니다. 오류 객체가 null 값으로 전달되면 데이터 쓰기가 완료된 것입니다.

파일을 실행하면 프로젝트 폴더 안에 output.txt 파일이 만들어지고 그 안에 'Hello World!'라는 글자가 쓰여 있는 것을 확인할 수 있습니다.

파일을 직접 열고 닫으면서 읽거나 쓰기

지금까지 살펴본 방식을 사용하면 어렵지 않게 파일을 읽거나 쓸 수 있습니다. 그런데 실제로 파일을 읽거나 쓸 때는 한꺼번에 모든 데이터를 읽거나 쓰지 않고 조금씩 읽거나 쓰는 방식을 사용하는 경우도 많습니다. 또한 다른 곳에서 받아 온 데이터를 파일에 쓰는 경우도 있기 때문에 파일을 다루는 다양한 방식이 따로 정의되어 있습니다.

파일을 직접 열고 닫으면서 읽거나 쓰고 싶다면 다음 메소드들을 사용할 수 있습니다.

메소드 이름	설명
open(path, flags [, mode] [, callback])	파일을 엽니다.
read(fd, buffer, offset, length, position [, callback])	지정한 부분의 파일 내용을 읽어 들입니다.
write(fd, buffer, offset, length, position [, callback])	파일의 지정한 부분에 데이터를 씁니다.
close(fd [, callback])	파일을 닫아 줍니다.

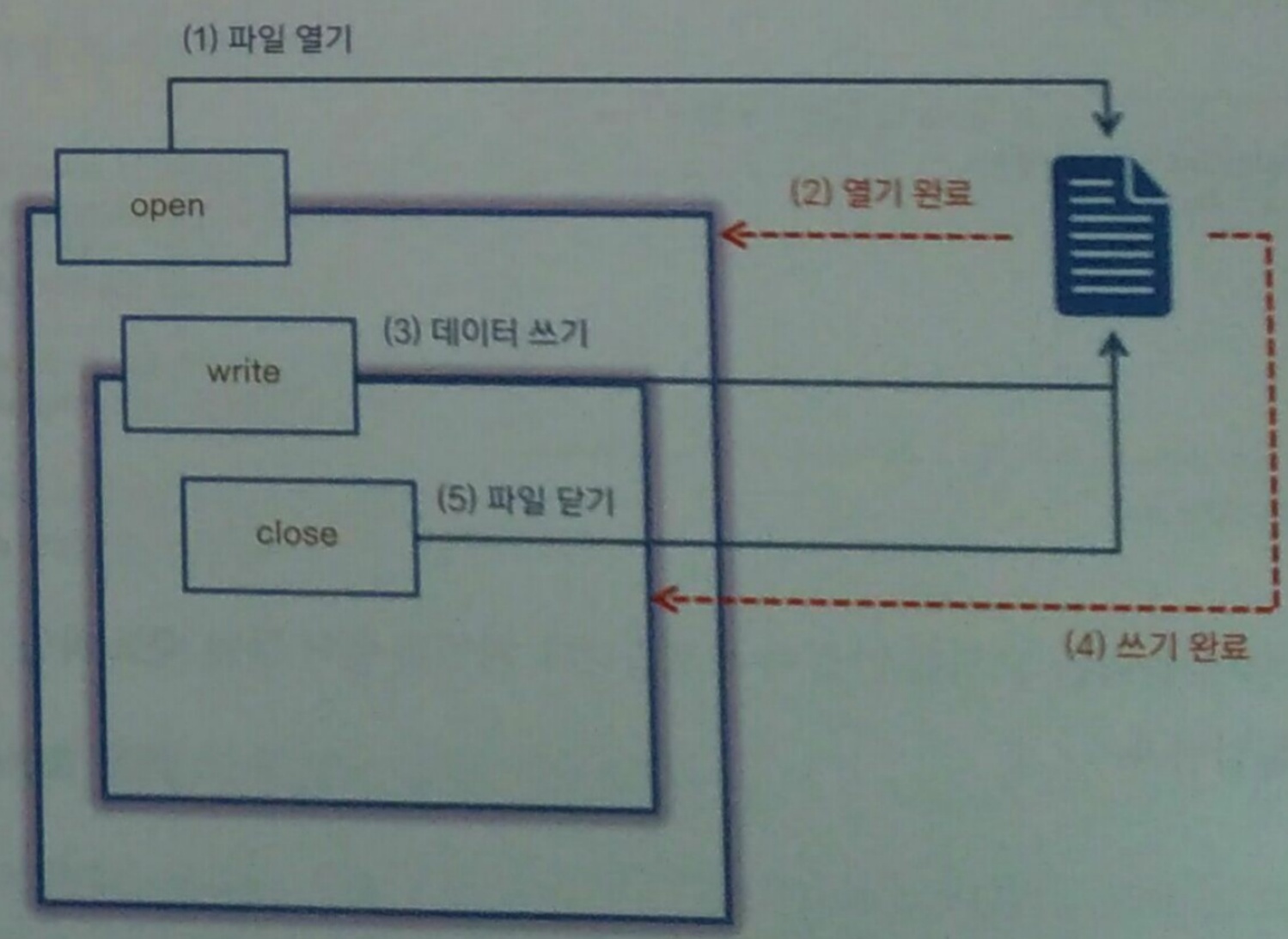
[표] 파일을 직접 열고 닫으면서 일부분만 읽거나 쓰기 위한 메소드

파일의 내용을 일부분만 읽어 들이려면 먼저 open( ) 메소드로 파일을 열어야 합니다. 파일을 열고 데이터 쓰기 기능을 만들려면 새로운 자바스크립트 파일 ch04_test8.js를 만들고 다음 코드를 입력합니다.

참조 파일 | NodeExample1>ch04_test8.js

```
var fs = require('fs');

// 파일에 데이터를 씁니다.
fs.open('./output.txt', 'w', function(err, fd) {
  if(err) throw err;

  var buf = new Buffer('안녕!\n');
  fs.write(fd, buf, 0, buf.length, null, function(err, written, buffer) {
    if(err) throw err;

      console.log(err, written, buffer);

     fs.close(fd, function() {
     console.log('파일 열고 데이터 쓰고 파일 닫기 완료.');
    });
  });
});
```

이 코드를 보면 함수를 호출할 때마다 파라미터로 함수를 전달하고 있습니다. 따라서 함수 안에 함수가 들어가는 형태가 되므로 함수 호출의 순서가 중요합니다. 함수를 호출하는 순서는 [open → write → close]입니다.



▲ 파일을 열어 데이터를 쓴 후 파일을 닫는 과정

먼저 open( ) 메소드를 호출하면 파일을 열 수 있습니다. 그리고 파일을 열면 write( ) 메소드를 사용해서 데이터를 쓸 수 있습니다. 데이터는 필요한 만큼 Buffer 객체 안에 씁니다. 파일에 데이터를 쓰고 나면 close( ) 메소드를 호출하여 파일을 닫습니다. 이렇게 각 메소드를 호출할 때마다 콜백 함수를 파라미터로 전달하므로 각각의 기능이 실행을 끝냈을 때 그다음 메소드를 실행합니다.

파일을 열기 위해 open( ) 메소드를 호출할 때 동시에 세 개의 파라미터가 전달되었습니다. 첫 번째는 파일의 이름, 두 번째는 파일을 읽거나 쓰기 위한 플래그입니다. 대표적인 플래그로는 'r', 'w', 'w+', 'a+'가 있습니다.

플래그	설명
'r'	읽기에 사용하는 플래그입니다. 파일이 없으면 예외가 발생합니다.
'w'	쓰기에 사용하는 플래그입니다. 파일이 없으면 만들어지고 파일이 있으면 이전 내용을 모두 삭제합니다.
'w+'	읽기와 쓰기에 모두 사용하는 플래그입니다. 파일이 없으면 만들어지고 파일이 있으면 이전 내용을 모두 삭제합니다.
'a+'	읽기와 추가에 모두 사용하는 플래그입니다. 파일이 없으면 만들어지고 있으면 이전 내용에 새로운 내용을 추가합니다.

[표] 파일을 열 때 사용하는 대표적인 플래그

코드에서는 'w' 플래그를 사용하였으므로 쓰기 작업만을 위해 파일을 엽니다. 콜백 함수 안에서는 버퍼 객체를 하나 만들고 '안녕!'이라는 글자를 넣은 후 write( ) 메소드를 호출하여 파일에 내용을 씁니다. 파일이 열리면 fd 객체를 전달받을 수 있으므로 이 fd 객체로 파일을 구별합니다. 파일에 데이터를 쓸 때 어느 위치에 쓸 것인지 정할 수 있습니다. 코드를 모두 입력한 후 실행하면 파일이 만들어지는데, 다음과 같이 그 파일에 쓴 내용이 콘솔 창에 표시됩니다.

```
Console  Debug
<terminated> NodeExample1-ch04_test8.js [Node Application] Node.js Process
null 8 <Buffer ec 95 88 eb 85 95 21 0a>
파일 열고 데이터 쓰고 파일 닫기 완료.
```

▲ 파일을 열고 데이터를 쓴 후 파일을 닫는 프로그램의 실행 결과

이번에는 앞에서 만들어진 파일을 읽어 들이는 코드를 만들어 보겠습니다. 파일을 읽어 들일 때도 파일을 먼저 열고 난 후 데이터를 읽어 들입니다.

참조 파일 | NodeExample1>ch04_test9.js

```
var fs = require('fs');

// 파일에서 데이터를 읽어 들입니다.
fs.open('./output.txt', 'r', function(err, fd) {
  if(err) throw err;

  var buf = new Buffer(10);
  console.log('버퍼 타입 : %s', Buffer.isBuffer(buf));

  fs.read(fd, buf, 0, buf.length, null, function(err, bytesRead, buffer) {
    if(err) throw err;

    var inStr = buffer.toString('utf8', 0, bytesRead);
    console.log('파일에서 읽은 데이터  : %s', inStr);

    console.log(err, bytesRead, buffer);

    fs.close(fd, function() {
      console.log('output.txt 파일을 열고 읽기 완료.');
    });
  });
});
```

Buffer 객체는 바이너리 데이터를 읽고 쓰는 데 사용합니다. 새로운 버퍼 객체를 만들기 위해서는 new 연산자를 사용하며, 그 안에 들어갈 데이터의 크기만 지정하면 됩니다. write( ) 메소드를 사용해 문자열을 버퍼에 쓰거나 처음부터 문자열을 사용해 버퍼 객체를 만들 수도 있습니다. 파일을 실행하면 파일에서 읽은 데이터가 콘솔 창에 출력됩니다.

```
Console  Debug
<terminated> NodeExample1-ch04_test9.js [Node Application] Node.js Process
버퍼 타입 :  true
파일에서 읽은 데이터  : 안녕!

null 8 <Buffer ec 95 88 eb 85 95 21 0a 00 f0>
output.txt 파일을 열고 읽기 완료.
```

▲ 파일을 열고 데이터를 읽은 후 콘솔 창에 보여 준 결과

버퍼 객체 사용하는 방법 알아보기

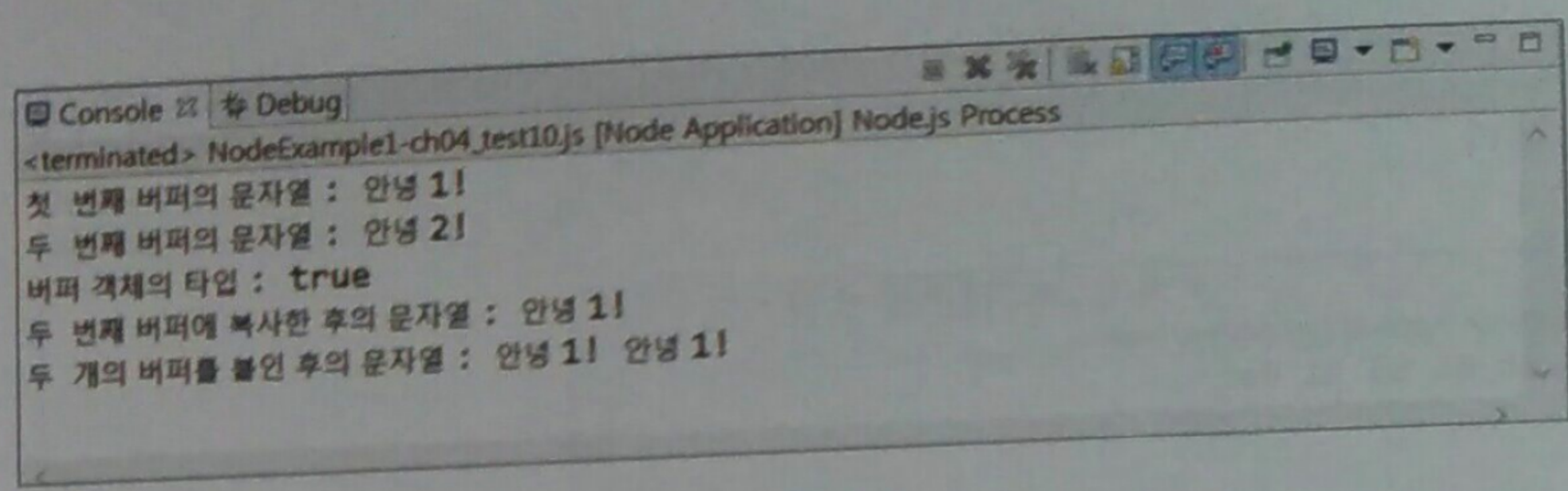
이번에는 Buffer 객체를 사용하는 방법에 대해 알아봅니다. 다음은 Buffer 객체의 기능을 확인하기 위해 만든 간단한 코드입니다.

참조 파일 | NodeExample1>ch04_test10.js

```
// 버퍼 객체를 크기만 지정하여 만든 후 문자열을 씁니다.
var output = '안녕 1!';
var buffer1 = new Buffer(10);
var len = buffer1.write(output, 'utf8');
console.log('첫 번째 버퍼의 문자열 : %s', buffer1.toString());

// 버퍼 객체를 문자열을 이용해 만듭니다.
var buffer2 = new Buffer('안녕 2!', 'utf8');
console.log('두 번째 버퍼의 문자열 : %s', buffer2.toString());

// 타입을 확인합니다.
console.log('버퍼 객체의 타입 : %s', Buffer.isBuffer(buffer1));

// 버퍼 객체에 들어 있는 문자열 데이터를 문자열 변수로 만듭니다.
var byteLen = Buffer.byteLength(output);
var str1 = buffer1.toString('utf8', 0, byteLen);
var str2 = buffer2.toString('utf8');

// 첫 번째 버퍼 객체의 문자열을 두 번째 버퍼 객체로 복사합니다.
buffer1.copy(buffer2, 0, 0, len);
console.log('두 번째 버퍼에 복사한 후의 문자열 : %s', buffer2.toString('utf8'));

// 두 개의 버퍼를 붙여 줍니다.
var buffer3 = Buffer.concat([buffer1, buffer2]);
console.log('두 개의 버퍼를 붙인 후의 문자열 : %s', buffer3.toString('utf8'));
```

두 개의 버퍼를 서로 다른 방식으로 만듭니다. 하나는 빈 버퍼를 먼저 만들고 그 안에 문자열을 넣었으며, 다른 하나는 버퍼를 만들면서 문자열을 파라미터로 전달하였습니다. 이 두 개의 버퍼에 대해 toString() 메소드를 호출하여 결과 문자열을 확인해 보면 문자열이 똑같이 들어 있는 것을 알 수 있습니다. 다만 크기를 먼저 지정하면 나머지 공간이 그대로 버퍼에 남아 있게 됩니다. 변수에 들어 있는 것이 버퍼 객체인지 아닌지 확인할 때는 isBuffer( ) 메소드를 사용합니다. 하나의 버퍼 객체를 다른 버퍼 객체로 복사할 때는 copy( ) 메소드를 사용하며, 두 개의 버퍼를 하나로 붙여서 새로운 버퍼 객체를 만들 때는 concat() 메소드를 사용합니다.

버퍼에 대해 이해했다면, 파일을 읽었을 때 콜백 함수로 전달된 버퍼를 이용해 문자열을 만드는 부분도 이해가 될 것입니다. 파일을 실행하면 다음과 같이 출력됩니다.



▲ 버퍼를 다루는 기능의 실행 결과

스트림 단위로 파일 읽고 쓰기

파일을 읽거나 쓸 때는 데이터 단위가 아닌 스트림 단위로 처리할 수도 있습니다. 스트림은 데이터가 전달되는 통로와 같은 개념입니다. 파일에서 읽을 때는 createReadStream( ), 파일에 쓸 때는 createWriteStream( ) 메소드로 스트림 객체를 만든 후 데이터를 읽고 쓰게 됩니다.

메소드 이름	설명
createReadStream(path[, options])	파일을 읽기 위한 스트림 객체를 만듭니다.
createWriteStream(path[, options])	파일을 쓰기 위한 스트림 객체를 만듭니다.

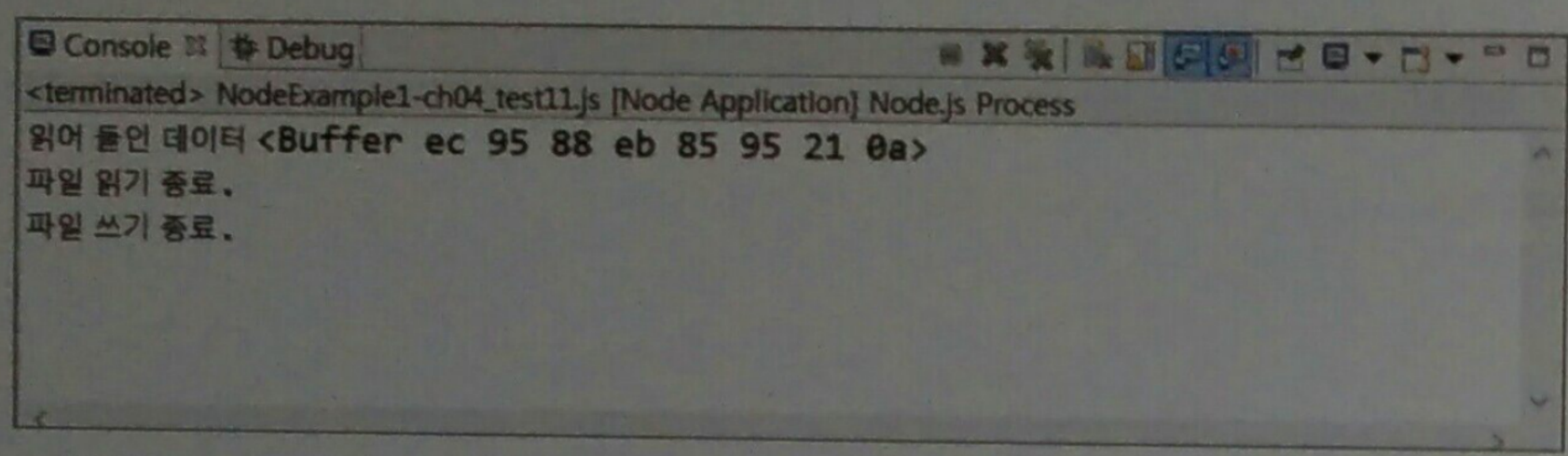
[표] 파일을 스트림 단위로 처리하는 대표적인 메소드

옵션으로는 flags, encoding, autoClose 속성이 들어 있는 자바스크립트 객체를 전달할 수 있습니다. 다음은 output.txt 파일의 내용을 읽어 들인 후 output2.txt 파일로 쓰는 코드입니다.

참조 파일 | NodeExample1>ch04_test11.js

```
var fs = require('fs');

var infile = fs.createReadStream('./output.txt', {flags: 'r'} );
var outfile = fs.createWriteStream('./output2.txt', {flags: 'w'});

infile.on('data', function(data) {
  console.log('읽어 들인 데이터', data);
  outfile.write(data);
});

infile.on('end', function() {
  console.log('파일 읽기 종료.');
  outfile.end(function() {
    console.log('파일 쓰기 종료.');
  });
});
```

파일을 실행하면 새로운 output2.txt 파일이 만들어지며, 그 안에 output.txt 파일의 내용이 똑같이 들어 있는 것을 확인할 수 있습니다.



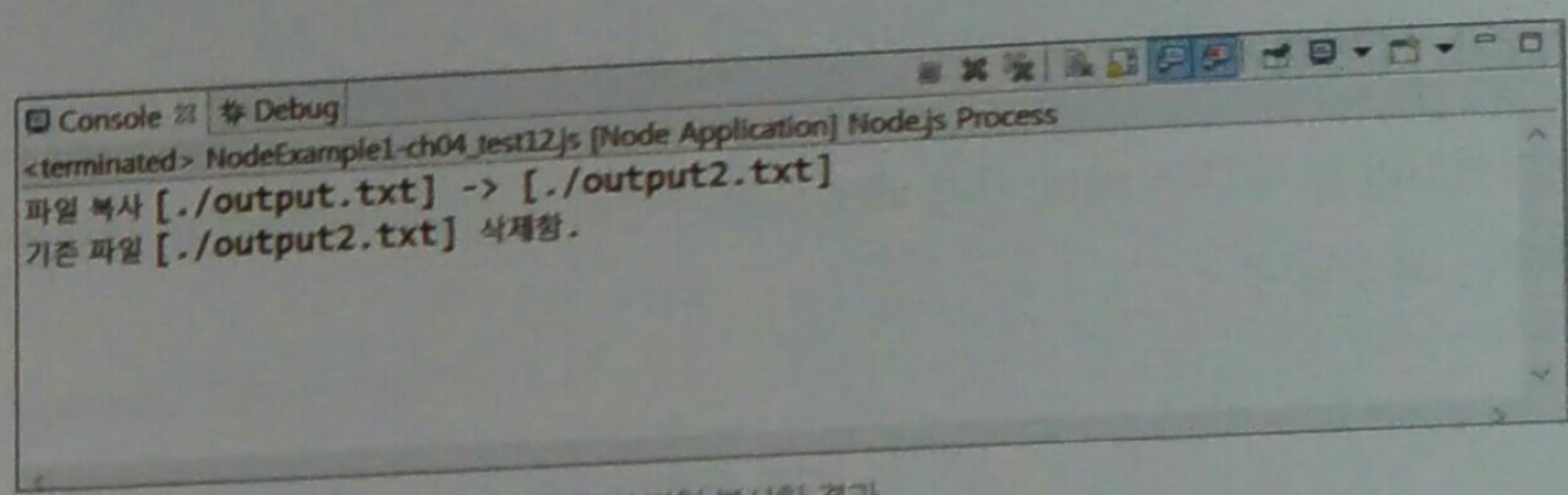
▲ 파일을 스트림으로 읽은 후 스트림으로 쓰는 기능의 실행 결과

앞서 살펴본 기능은 두 개의 스트림을 붙여 주면 더 간단하게 만들 수 있습니다. pipe( ) 메소드는 두 개의 스트림을 붙여 주는 역할을 합니다. ReadStream 타입의 객체와 WriteStream 타입의 객체를 붙여 주면 스트림 간에 데이터를 알아서 전달합니다. 다음은 앞에서 실행한 코드를 pipe( ) 메소드로 바꾼 것입니다.

참조 파일 | NodeExample1>ch04_test12.js

```
var fs = require('fs');

var inname = './output.txt';
var outname = './output2.txt';

fs.exists(outname, function(exists) {
  if(exists) {
    fs.unlink(outname, function(err) {
      if(err) throw err;
      console.log('기존 파일 [' + outname +'] 삭제함.');
    });
  }

  var infile = fs.createReadStream(inname, {flags: 'r'});
  var outfile = fs.createWriteStream(outname, {flags: 'w'});

  infile.pipe(outfile);
  console.log('파일 복사 [' + inname + '] -> [' + outname + ']');
});
```

기존에 만들어 놓은 output2.txt 파일이 있으면 파일이 중복될 수 있습니다. 그래서 같은 이름을 가진 파일을 다시 만들기 전에 먼저 이전 파일을 삭제하도록 unlink( ) 메소드를 사용하였습니다. 파일을 실행해 보면 pipe( ) 메소드로 두 개의 스트림 객체를 연결하기만 했는데도 파일 내용이 복사된 것을 알 수 있습니다.



▲ pipe( ) 메소드로 두 개의 스트림을 연결하여 파일 복사한 결과

주의! 기존의 output2.txt 파일을 삭제한 후 output.txt 파일을 output2.txt 파일로 복사하는 과정이 비동기 방식으로 처리되므로 화면에 출력되는 순서는 다를 수 있습니다.

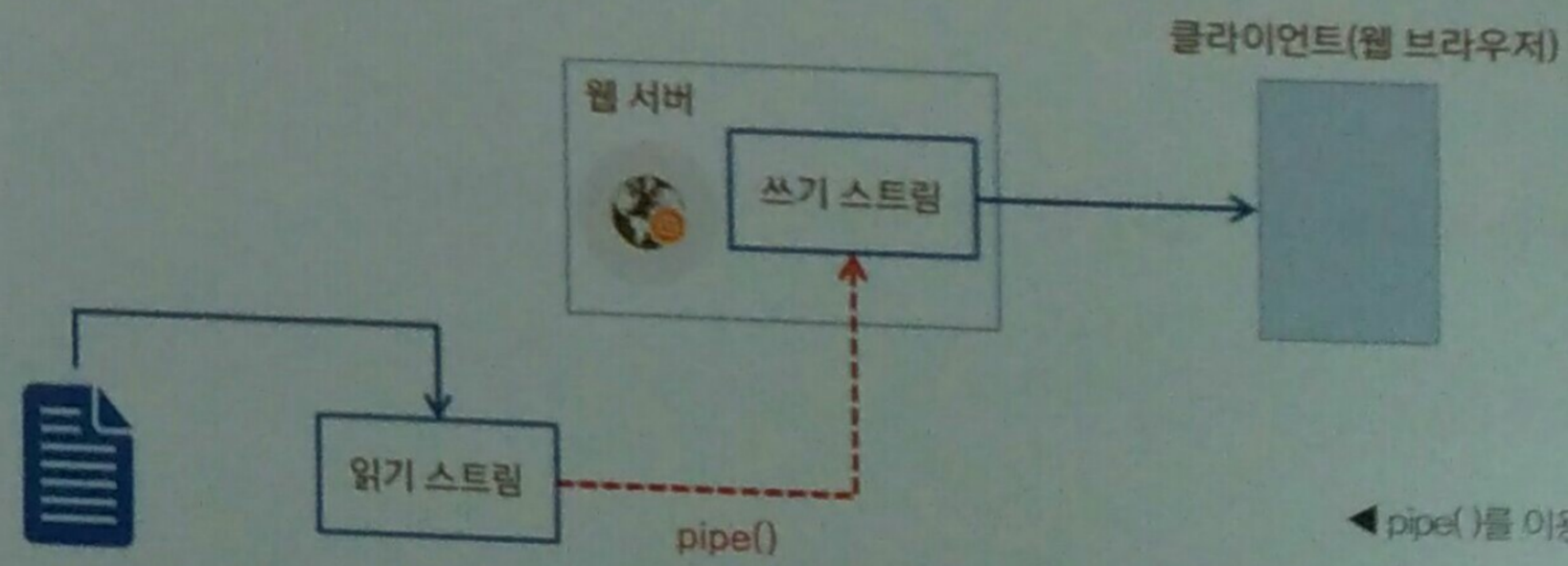
http 모듈로 요청받은 파일 내용을 읽고 응답하기

이렇게 스트림을 서로 연결하는 방법은 웹 서버를 만들고 사용자의 요청을 처리할 때 유용합니다. 다음은 http 모듈을 사용해 사용자로부터 요청을 받았을 때 파일의 내용을 읽어 응답으로 보내는 코드입니다.

참조 파일 | NodeExample1>ch04_test13.js

```
var fs = require('fs');
var http = require('http');

var server = http.createServer(function(req, res) {
  // 파일을 읽어 응답 스트림과 pipe()로 연결합니다.
  var instream = fs.createReadStream('./output.txt');
  instream.pipe(res);
});

server.listen(7001, '127.0.0.1');
```

웹 서버에서 클라이언트로부터 요청을 받으면 먼저 output.txt 파일에서 스트림을 만든 후 클라이언트로 데이터를 보낼 수 있는 스트림과 연결해 줍니다. http 모듈을 이용해 웹 서버를 만드는 방법은 다음 장에서 알아볼 것입니다. 지금은 파일에서 만든 스트림 객체와 웹 서버의 스트림 객체를 pipe( ) 메소드로 연결할 수 있다는 것만 이해하면 됩니다. 두 객체의 연결이 가능한 이유는 파일에서 데이터를 읽어오기 위해 만든 것도 스트림 객체이고, 데이터를 쓰기 위해 웹 서버에서 클라이언트 쪽에 만든 것도 스트림 객체이기 때문입니다. 따라서 읽기 스트림과 쓰기 스트림은 pipe( ) 메소드를 사용해 연결할 수 있습니다.



◀ pipe( )를 이용해 읽기 스트림과 쓰기 스트림 연결하기

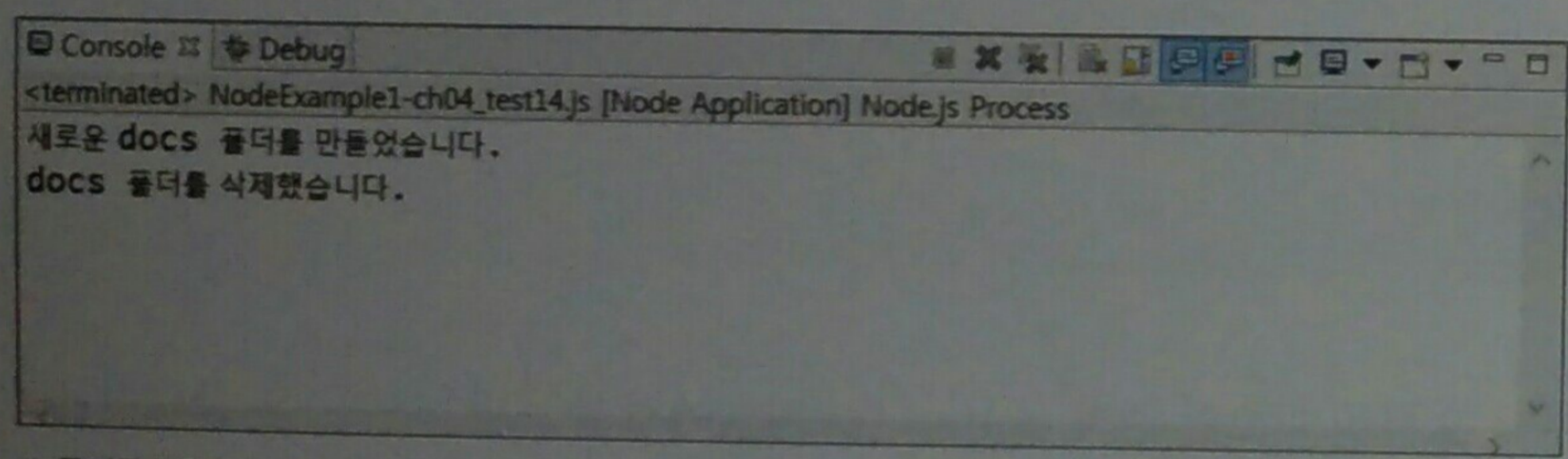
fs 모듈로 새 디렉터리 만들고 삭제하기

fs 모듈은 그 밖에도 파일과 디렉터리를 다루는 여러 가지 메소드를 포함하고 있습니다. 다음은 새로운 디렉터리를 만들었다가 삭제하는 코드입니다.

참조 파일 | NodeExample1>ch04_test14.js

```
var fs = require('fs');
fs.mkdir('./docs', 0666, function(err) {
  if(err) throw err;
  console.log('새로운 docs 폴더를 만들었습니다.');

  fs.rmdir('./docs', function(err) {
    if(err) throw err;
    console.log('docs 폴더를 삭제했습니다.');
  });
});
```

파일을 실행하면 새로운 폴더가 만들어졌다가 삭제됩니다. 만약 삭제하는 부분의 코드를 // 기호를 이용해서 설명글(Comment, 주석)로 막는다면 새로운 폴더는 만들어지지만 삭제되지는 않습니다. 그러므로 파일 탐색기를 열어 새로 만든 파일을 확인할 수 있습니다.



▲ 폴더를 만들었다가 삭제한 결과

이제 파일을 처리하는 방법에 대해 어느 정도 이해되었을 것입니다. 그런데 파일의 내용을 읽거나 쓰는 데 사용하는 자바스크립트 코드의 양이 많지 않음에도 불구하고 노드의 비동기 프로그래밍 방식이 콜백 함수를 사용하기 때문에 코드 구조가 조금 복잡해 보일 수 있습니다. 또 기능을 추가함에 따라 코드의 양도 점점 많아집니다. 이 때문에 자바스크립트를 사용해 본 웹 개발자들도 노드의 코드 패턴이 익숙하지 않다고 느낍니다.

일단 여기에서는 파일을 다루는 방식과 코드 입력 방식에 대해 잘 알아두길 바랍니다.

04-4 | 로그 파일 남기기

console 객체의 log( ) 또는 error( ) 메소드 등을 호출하면 로그를 출력할 수 있습니다. 그런데 프로그램의 크기가 커질수록 로그의 양도 많아지고 로그를 보관했다가 나중에 확인해야 하는 경우도 생깁니다. 따라서 어떻게 로그를 남기고 보관할 것인지가 중요해집니다. 로그를 보관하려면 화면에만 출력하는 것만으로는 부족합니다. 이 때문에 다양한 방식으로 로그를 남길 수 있도록 외부 모듈을 사용합니다. 여기에서는 여러 가지 로그 모듈 중에서 **winston 모듈**로 로그를 남기는 방법을 알아보겠습니다.

로그를 남기려면 설정이 필요합니다. 이렇게 설정한 코드는 한번 만들어 두면 그대로 복사한 후 일부 설정만 바꾸어 다른 프로그램에도 사용할 수 있습니다. 그렇기 때문에 처음 코드를 만들 때 그 코드의 사용 방법을 잘 이해하는 것이 중요합니다. 새로운 ch04_test15.js 파일을 만들고 다음 코드를 그대로 입력합니다.

참조 파일 | NodeExample1>ch04_test15.js

```
var winston = require('winston'); // 로그 처리 모듈
var winstonDaily = require('winston-daily-rotate-file'); // 로그 일별 처리 모듈
var moment = require('moment'); // 시간 처리 모듈

function timeStampFormat() {
   return moment().format('YYYY-MM-DD HH:mm:ss.SSS ZZ');
   // ex) '2016-05-01 20:14:28.500 +0900'
};

var logger = new (winston.Logger)({
      transports: [
          new (winstonDaily)({
                name: 'info-file',
                filename: './log/server',
                datePattern: '_yyyy-MM-dd.log',
                colorize: false,
                maxsize: 50000000,
                maxFiles: 1000,
                level: 'info',
                showLevel: true,
                json: false,
                timestamp: timeStampFormat
    }),
```

```
        new (winston.transports.Console)({
                name: 'debug-console',
                colorize: true,
                level: 'debug',
                showLevel: true,
                json: false,
                timestamp: timeStampFormat
        })
],
  exceptionHandlers: [
        new (winstonDaily)({
                name: 'exception-file',
                filename: './log/exception',
                datePattern: '_yyyy-MM-dd.log',
                colorize: false,
                maxsize: 50000000,
                maxFiles: 1000,
                level: 'error',
                showLevel: true,
                json: false,
                timestamp: timeStampFormat
        }),
        new (winston.transports.Console)({
                name: 'exception-console',
                colorize: true,
                level: 'debug',
                showLevel: true,
                json: false,
                timestamp: timeStampFormat
        })
    ]
});
```

코드의 양이 조금 많아 보이지만 실무에서 사용할 수 있는 로그 설정을 만들기 위해서는 이 정도의 코드가 필요합니다.

winston 모듈로 만드는 **로거(Logger, 로그를 출력하는 객체를 말할 때 사용하는 용어)**는 transports라는 속성 값으로 여러 개의 설정 정보를 전달할 수 있습니다. 이름이 info-file인 설정 정보는 매일 새로운 파일에 로그를 기록하도록 설정하고, info 수준의 로그만 기록하도록 설정하였습니다. 로그 파일의 크기가 50MB를 넘어가면 자동으로 새로운 파일로 생성되며, 이때 자동으로 분리되어 생성되는 파일의 개수는 최대 1000개까지 가능합니다. 이런 정보들은 name, level, maxsize, maxFiles 등의 속성으로 설정합니다. 콘솔 창에 출력되는 로그는 debug 수준까지 출력되도록 설정하고 컬러도 적용하였습니다.

정박사의 한마디 로그 수준(Log Level)에 대해 아시나요?

로그 수준이란 어떤 정보까지 출력할 것인지 결정하는 것을 말합니다. 예를 들어, 오류만 보여 줄 것인지 아니면 사소한 정보까지 모두 보여 줄 것인지를 로그 수준으로 결정합니다. winston 모듈에서 사용하는 로그 수준은 다음처럼 단계별로 구성되며, 하위 수준은 상위 수준을 모두 포함하여 출력합니다.

```
debug: 0 > info: 1 > notice: 2 > warning: 3 > error: 4 > crit: 5 > alert: 6 > emerg: 7
```

여러분이 만드는 프로그램별로 로그 파일의 이름을 다르게 기록해야 합니다. 그러니 앞에서 살펴 본 코드를 복사하여 다른 프로그램에서 사용하고 싶다면 파일 이름이나 설정 부분을 약간씩 수정하면서 사용하세요.

ch04_test15.js 파일은 winston 모듈 외에 winston-daily-rotate-file 모듈과 시간을 다룰 수 있는 moment 모듈도 함께 사용합니다. 따라서 파일을 실행하기 전에 먼저 다음과 같이 명령 프롬프트에서 모듈을 설치해야 합니다.

```
% npm install winston --save
% npm install winston-daily-rotate-file --save
% npm install moment --save
```

로그를 설정하는 코드 아래에는 output.txt 파일을 복사하면서 로그를 남기는 코드가 들어 있습니다. 따라서 파일 탐색기를 열고 로그 파일이 저장될 [log] 폴더를 만든 후 ch04-test15.js 파일을 실행하면 다음과 같이 로그 파일이 만들어지는 것을 알 수 있습니다. winston 모듈은 로그를 파일로 저장하면서 동시에 화면에 출력되도록 설정할 수 있습니다. 로그를 파일이나 화면에 출력하고 싶다면 logger 객체에 들어 있는 debug( ), info( ), error( )와 같은 메소드를 호출하면 됩니다. logger.debug( ) 메소드를 호출하면서 글자를 넣으면 다음과 같이 콘솔에 그 글자가 출력됩니다.

```
Console  Debug
<terminated> NodeExample1-ch04_test15.js [Node Application] Node.js Process
2015-06-28 17:23:31.213 +0900 - info: 파일 복사 [./output.txt] -> [./output2.txt]
2015-06-28 17:23:31.215 +0900 - info: 기존 파일 [./output2.txt] 삭제됨.
```

◀ 로그 파일을 만들어 낸 화면

지금까지 노드의 기본 모듈 몇 가지, 이벤트와 파일을 다루는 방법, 로그를 남기는 방법에 대해 알아보았습니다. 이 장에서는 노드가 다른 일반적인 언어에서 제공하는 기본 기능을 똑같이 사용할 수 있으면서 비동기 프로그래밍 방식으로 코드를 입력한다는 것을 살펴보았습니다. 그러나 아직까지는 노드로 서버를 만들 때 다른 언어에 비해 편리하다고 느낄 만한 내용은 아닙니다.

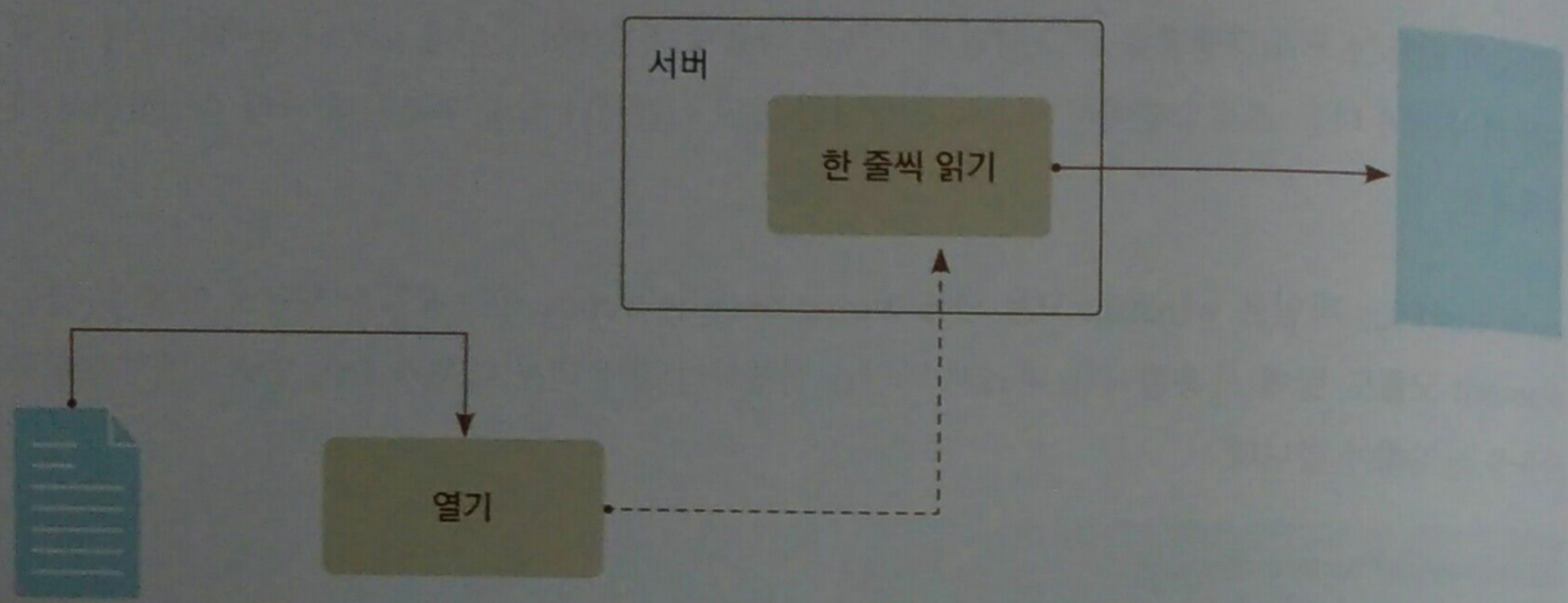
다음 장부터는 여러 가지 유형의 서버를 하나씩 만들어 갈 것입니다. 그중에서도 웹 서버는 다른 모든 서버들을 구성하기 위한 기초가 되므로 먼저 간단한 웹 서버를 만드는 방법에 대해 살펴봅니다. 웹 서버를 만드는 과정에서 서버를 만들 때 편리한 기능을 하나둘씩 보게 될 것입니다.

Do it!

도전 문제

01. 파일의 내용을 한 줄씩 읽어 들여 화면에 출력하는 기능을 만들어 보세요.

프로젝트 소스 Mission01 난이도 초급 ★☆☆☆



Hint!

1 하나의 파일을 만들고 각 줄에는 공백으로 구분된 이름, 나이, 전화번호가 들어가도록 구성합니다.

2 파일의 내용을 한 줄씩 읽어 들이면서 각 정보를 공백으로 구분합니다.

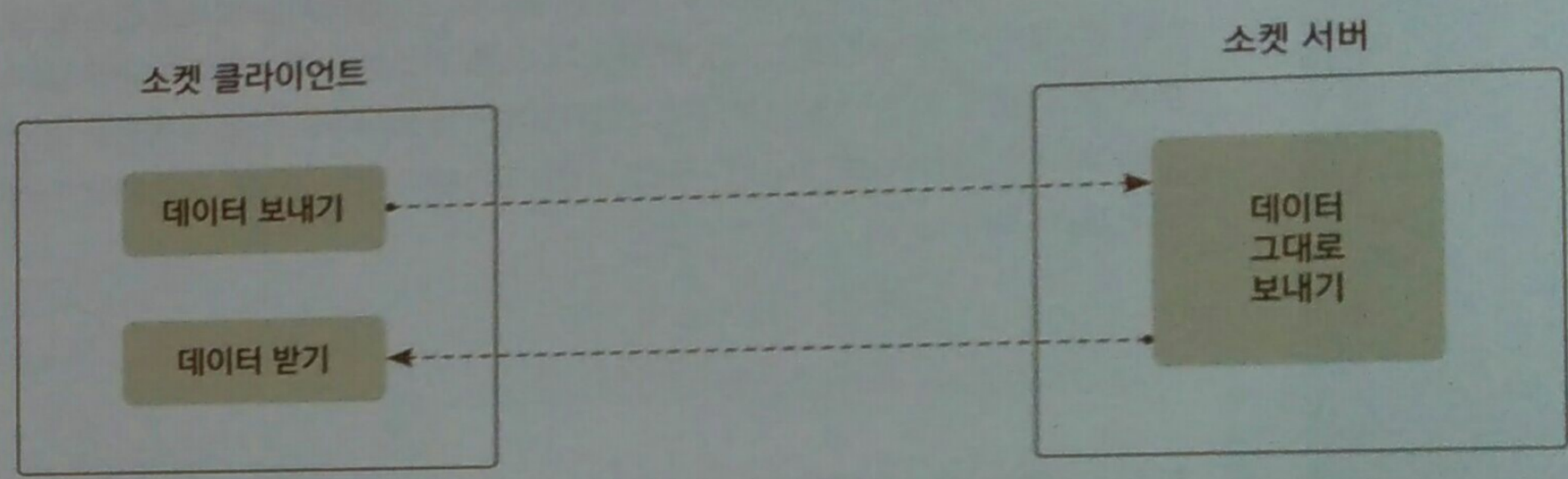
3 구분된 정보 중에서 이름만 화면에 출력합니다.

MEMO

Do it!
도전 문제

02. 소켓 서버와 소켓 클라이언트 기능을 노드로 구성하여 소켓 클라이언트에서 데이터를 보내면 소켓 서버에서 다시 돌려주는 기능을 만들어 보세요.

프로젝트 소스 Mission02 난이도 중급 ★★☆☆



Hint!

1. 노드의 소켓 기능을 이용해 소켓 서버와 소켓 클라이언트를 만듭니다.
2. 소켓 클라이언트에서는 소켓 서버로 연결합니다.
3. 소켓 클라이언트에서 소켓 서버로 '안녕!' 같은 글자를 보내면 소켓 서버에서 그 글자를 그대로 다시 소켓 클라이언트로 보냅니다.
4. 소켓 클라이언트와 소켓 서버에서는 보내고 받은 데이터를 화면에 출력합니다.

MEMO

05 웹 서버 만들기

노드의 수많은 모듈 중에서도 웹 서버는 다른 서버 기능을 추가할 수 있는 서버입니다. 따라서 채팅 서버나 모바일 서버보다 먼저 살펴보는 게 좋습니다. 일단 노드에는 웹 서버를 만들 때 필요한 http 모듈이 들어 있는데 이 모듈을 사용하면 HTTP 프로토콜로 요청하는 내용과 응답을 모두 처리할 수 있습니다. 그러나 좀 더 쉽고 빠르게 웹 서버를 구성하려면 익스프레스(Express)를 사용하는 것이 좋습니다. 익스프레스는 웹 서버 기능을 쉽게 만들 수 있게 코드를 자동으로 만들어 줍니다. 이 장에서는 익스프레스로 웹 서버를 만든 후 쿠키나 세션 관리는 어떻게 하는지, 그리고 요청한 패스에 따라 라우팅(Routing, 분기)은 어떻게 하는지 알아봅니다. 그다음에는 웹 서버에서 사용자 정보를 관리하는 기능을 하나씩 만들어 보면서 웹 서버의 기본 기능부터 파일 업로드 기능까지 차례대로 살펴보겠습니다.

질문으로 시작하기

질문1. 빨리 웹 서버를 만들어보고 싶나요? ➡ 05-1. 간단한 웹 서버 만들기

질문2. 익스프레스로 웹 서버를 만들면 편리한가요? ➡ 05-2. 익스프레스로 웹 서버 만들기

질문3. 미들웨어는 어떻게 사용하나요? ➡ 05-3. 미들웨어 사용하기

질문4. 클라이언트가 요청한 것을 분기하여 처리하고 싶나요? ➡ 05-4. 요청 라우팅하기

질문5. 쿠키와 세션은 어떻게 처리하나요? ➡ 05-5. 쿠키와 세션 관리하기

질문6. 파일 업로드는 어떻게 하나요? ➡ 05-6. 파일 업로드 기능 만들기

05-1 | 간단한 웹 서버 만들기

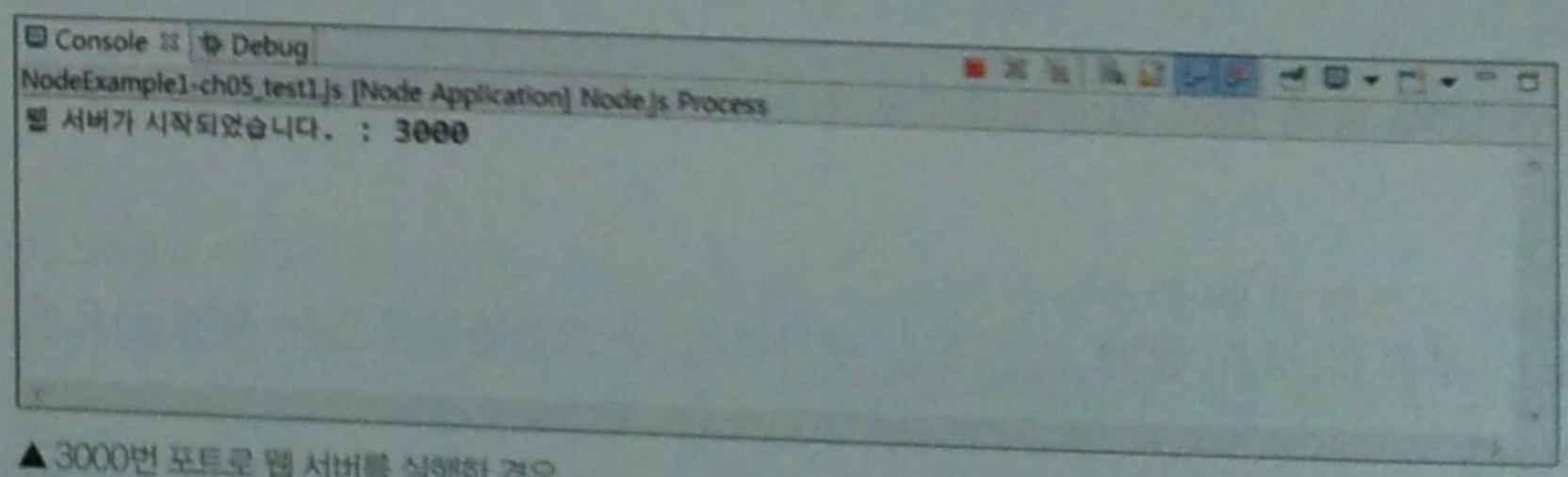
노드에 기본으로 들어 있는 http 모듈을 사용하면 웹 서버 기능을 담당하는 서버 객체를 만들 수 있습니다. http 모듈을 로딩했을 때 반환되는 객체에는 createServer( ) 메소드가 정의되어 있습니다. 따라서 이 메소드를 호출하면 서버 객체를 만들 수 있습니다. 새로운 자바스크립트 파일 ch05_test1.js를 만들고 다음 코드를 입력합니다.

참조 파일 | NodeExample1>ch05_test1.js

```
var http = require('http');

// 웹 서버 객체를 만듭니다.
var server = http.createServer();

// 웹 서버를 시작하여 3000번 포트에서 대기합니다.
var port = 3000;
server.listen(port, function() {
  console.log('웹 서버가 시작되었습니다. : %d', port);
});
```

http 모듈에 들어 있는 서버 기능을 사용하려면 먼저 require( ) 메소드로 http 모듈을 불러옵니다. 그런 다음 모듈 이름과 똑같은 http 변수를 만든 후 이 변수에 할당합니다. 이 http 객체의 createServer( ) 메소드를 호출하면 서버 객체가 반환됩니다. 이 서버 객체의 listen( ) 메소드를 호출하면 웹 서버가 시작됩니다. 서버를 시작할 때는 포트를 3000번으로 지정하여 해당 포트에서 클라이언트의 요청을 대기합니다. listen( ) 메소드를 호출할 때 전달하는 두 번째 파라미터는 콜백 함수로, 웹 서버가 시작되면 호출됩니다.

ch05_test1.js 파일을 실행해 보면 콘솔 창에 다음과 같은 로그가 출력되면서 웹 서버가 시작됩니다. 만약 서버를 중지하고 싶다면 콘솔 창의 위쪽 탭에 있는 빨간색 사각형 버튼을 누르면 됩니다.



▲ 3000번 포트로 웹 서버를 실행한 경우

와우! 벌써 웹 서버를 만들었습니다. 하지만 웹 브라우저에서 이 웹 서버로 무언가를 요청했을 때 아직까지는 웹 서버가 해 줄 수 있는 것이 아무것도 없습니다. 그래도 지금 실행한 것이 웹 서버이니 이 웹 서버가 어떤 과정으로 만들어졌는지 다시 한 번 자세히 살펴보겠습니다.

▲ http 모듈로 웹 서버를 만들어 실행하는 과정

웹 서버는 일반적으로 웹 브라우저라고 하는 클라이언트에서 HTTP 프로토콜로 요청한 정보를 처리한 후 응답을 보내 주는 역할을 합니다. 이때 PC나 서버 기계에 있는 6만 개가 넘는 포트들 중에서 하나를 사용하는데 여기에서는 3000번 포트를 사용합니다. 웹 서버를 만들어 실행하는 과정을 보면 먼저 createServer( ) 메소드를 호출하여 웹 서버 객체를 만들고, 그 객체의 listen( ) 메소드를 호출하여 특정 포트에서 대기하도록 설정합니다. createServer( ) 메소드로 만든 서버 객체에서 사용할 수 있는 대표적인 메소드는 다음과 같습니다.

메소드 이름	설명
listen(port[, hostname][, backlog][, callback]	서버를 실행하여 대기시킵니다.
close([callback])	서버를 종료합니다.

[표] 서버 객체에서 사용할 수 있는 대표적인 메소드

그런데 PC나 서버 기계에 이더넷(Ethernet) 카드가 여러 개 있는 경우에는 서버에서 사용할 수 있는 인터넷 주소, 즉 IP 주소가 여러 개 존재합니다. 이렇게 IP 주소가 여러 개 있는 경우에는 특정 IP를 지정하면서 서버를 실행해야 할 때도 있습니다. 이런 경우에는 listen( ) 메소드를 호출하면서 IP를 직접 지정합니다. 다음 코드를 사용하면 IP와 함께 백로그(실질적으로 동시 접속 연결 수를 결정하는 정보)를 지정할 수 있습니다.

앞에서 입력한 listen( ) 메소드 부분의 코드 앞에 // 기호를 추가하여 주석(실행되지 않는 설명글)으로 바꾼 후 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch05_test1.js

```
……

// 웹 서버를 시작하여 192.168.0.5 IP와 3000번 포트에서 대기하도록 설정합니다.
var host = '192.168.0.5';
var port = 3000;
server.listen(port, host, '50000', function() {
  console.log('웹 서버가 시작되었습니다. : %s, %d', host, port);
});
```

host 변수에 들어가는 IP 정보는 임시로 192.168.0.5를 넣었지만 여러분의 PC가 인터넷에 연결되었을 때의 IP로 바꾸어 지정해야 합니다.

정박사의 한마디 여러분 PC의 IP를 알고 싶으세요?

명령 프롬프트를 열고 다음과 같이 입력하면 사용 중인 PC의 IP를 확인할 수 있습니다.

```
% ipconfig /all
```

클라이언트가 웹 서버에 요청할 때 발생하는 이벤트 처리하기

만약 웹 브라우저가 이 웹 서버에 접속한 후 데이터를 요청하면 그때마다 적절한 이벤트가 발생하므로 다양한 상황에 맞추어 콜백 함수를 각각 등록하면 됩니다. 다음은 서버 객체에서 사용할 수 있는 주요 이벤트들입니다.

이벤트 이름	설명
connection	클라이언트가 접속하여 연결이 만들어질 때 발생하는 이벤트입니다.
request	클라이언트가 요청할 때 발생하는 이벤트입니다.
close	서버를 종료할 때 발생하는 이벤트입니다.

[표] 서버 객체의 주요 이벤트

이 외에도 여러 가지 이벤트가 있지만 클라이언트가 접속할 때 발생하는 connection 이벤트와 클라이언트가 특정 패스로 요청할 때 발생하는 request 이벤트가 가장 자주 사용됩니다.

웹 브라우저에서 요청할 때 어떤 이벤트가 발생하는지 보기 위해 새로운 파일 ch05_test2.js를 만든 후 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch05_test2.js

```
var http = require('http');

// 웹 서버 객체를 만듭니다.
var server = http.createServer();

// 웹 서버를 시작하여 3000번 포트에서 대기하도록 설정합니다.
var port = 3000;
server.listen(port, function() {
  console.log('웹 서버가 시작되었습니다. : %d', port);
});

// 클라이언트 연결 이벤트 처리
server.on('connection', function(socket) {
  var addr = socket.address();
  console.log('클라이언트가 접속했습니다. : %s, %d', addr.address, addr.port);
});

// 클라이언트 요청 이벤트 처리
server.on('request', function(req, res) {
  console.log('클라이언트 요청이 들어왔습니다.');
  console.dir(req);
});

// 서버 종료 이벤트 처리
server.on('close', function() {
  console.log('서버가 종료됩니다.');
});
```

웹 브라우저와 같은 클라이언트가 웹 서버에 연결되면 connection 이벤트가 발생합니다. 그러므로 on() 메소드를 호출할 때 첫 번째 파라미터로 connection 이벤트 이름을 전달하고, 두 번째 파라미터로 콜백 함수를 전달합니다. 연결이 만들어져 콜백 함수가 호출될 때는 Socket 객체가 파라미터로 전달됩니다. 이 객체는 클라이언트 연결 정보를 담고 있으므로 address( ) 메소드를 호출하여 클라이언트의 IP와 포트 정보를 확인할 수 있습니다.

클라이언트가 특정 패스로 요청을 하면 request 이벤트가 발생합니다. 이때도 앞에서 알아본 것처럼 on( ) 메소드를 사용해 이벤트를 처리합니다. 콜백 메소드로 전달되는 요청 객체를 console.dir( ) 메소드로 화면에 출력하면 어떤 정보가 들어 있는지 확인할 수 있습니다.

이 파일을 실행하고 웹 브라우저를 연 다음 주소 창에 http://localhost:3000/을 입력해서 사이트를 열어 봅니다. 웹 브라우저에는 아무런 반응이 없지만 콘솔 창에는 다음과 같은 결과가 출력됩니다.

```
Console  Debug
NodeExample1-ch05_test2.js [Node Application] Node.js Process
웹 서버가 시작되었습니다. : 3000
클라이언트가 접속했습니다. : ::1, 58732
클라이언트 요청이 들어왔습니다.
{ _readableState:
   { objectMode: false,
     highWaterMark: 16384,
     buffer: [],
     length: 0
```

▲ 웹 브라우저에서 클라이언트가 요청할 때 서버에 보이는 메시지

클라이언트가 접속한 IP와 포트를 알 수 있고, 클라이언트가 요청을 했을 때 콜백 함수에 전달되는 req 객체에 들어 있는 다양한 정보도 확인할 수 있습니다.

정박사의 한마디

파일을 실행했을 때 EADDRINUSE라는 오류가 발생하나요?

서버를 실행했을 때 다음과 같은 오류가 발생하면서 서버가 시작되지 않는 경우가 있습니다. 이것은 이미 같은 포트에서 서버가 시작되었기 때문에 발생하는 문제입니다.

```
Console  Debug
<terminated> NodeExample1-ch05_test2.js [Node Application] Node.js Process
events.js:85
      throw er; // Unhandled 'error' event
            ^
Error: listen EADDRINUSE
    at exports._errnoException (util.js:746:11)
    at Server._listen2 (net.js:1156:14)
    at listen (net.js:1182:10)
    at Server.listen (net.js:1267:5)
    at Object.<anonymous> (C:\Users\user\workspace-book5\NodeExample1\ch05_tes
    at Module._compile (module.js:460:26)
    at Object.Module._extensions..js (module.js:478:10)
```

▲ 서버를 실행했을 때 오류 메시지

이 경우 이전에 실행했던 서버를 먼저 종료해야 합니다. 콘솔 창의 오른쪽에 있는 콘솔 모양 아이콘 옆에는 아래쪽 화살표(▼)가 있습니다. 이 화살표를 눌러 보면 이미 실행되어 있는 여러 개의 콘솔 창이 중첩되어 있습니다.

```
Console  Debug
<terminated> NodeExam
events.js:85
      throw er; // U
            ^
Error: listen EADDR
    at exports._erru
    at Server._listen2 (net.js:1156:14)
    at listen (net.js:1182:10)
    at Server.listen (net.js:1267:5)
    at Object.<anonymous> (C:\Users\user\workspace-book5\NodeExample1\ch05_tes
    at Module._compile (module.js:460:26)
    at Object.Module._extensions..js (module.js:478:10)
```

```
1 Android
2 DDMS
3 Nodeclipse Console
4 NodeExample1-ch05_test2.js [Node Application] Node.js Process
5 <terminated> NodeExample1-ch05_test2.js [Node Application] Node.js Pr
```

▲ 콘솔 창이 여러 개 중첩되어 있는 경우

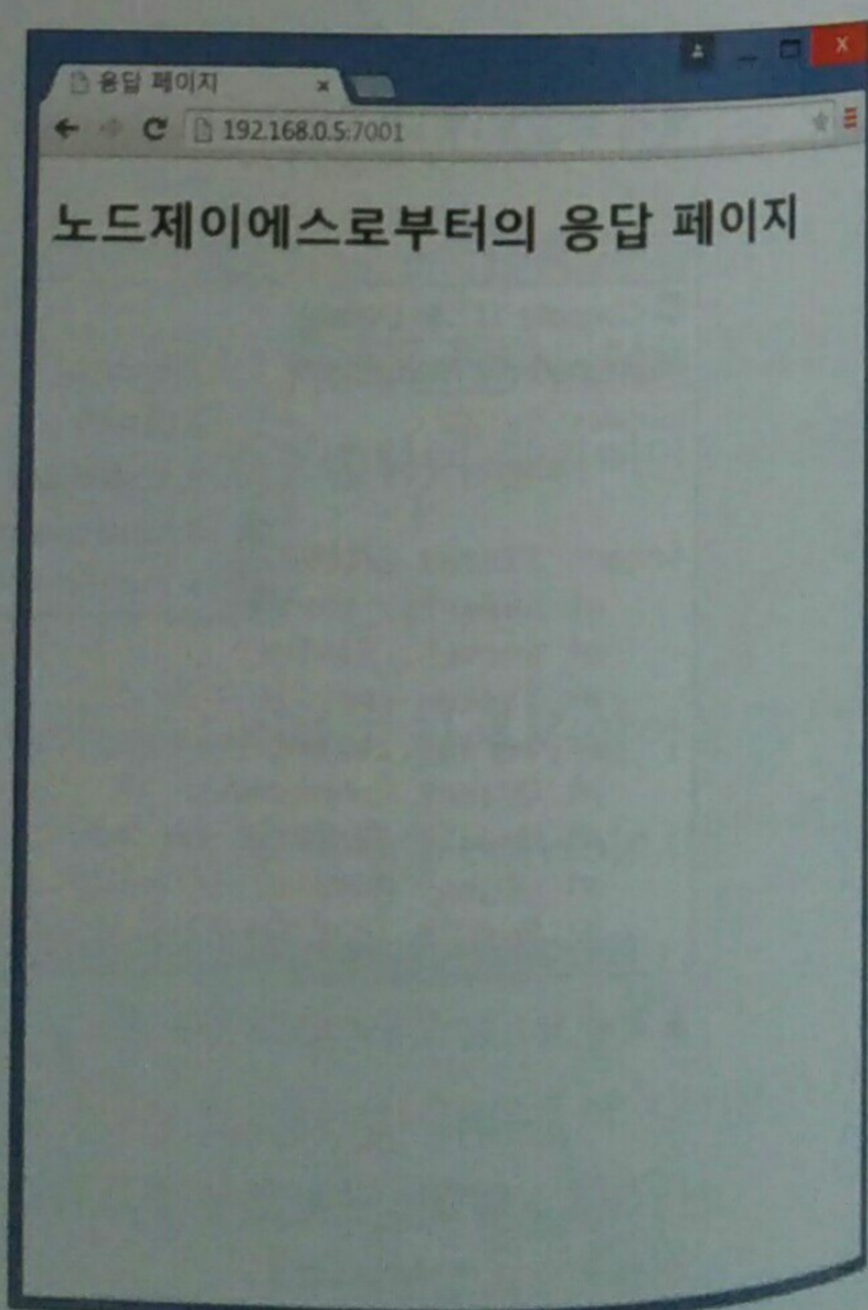
이전에 실행했던 콘솔 창을 열고 빨간색 사각형 버튼을 눌러 서버를 종료한 후 다시 파일을 실행하면 서버가 정상적으로 실행됩니다.

그런데 웹 브라우저에서 페이지를 열어도 서버에서 아무런 응답을 보내주지 않기 때문에 웹 브라우저에서는 결과 페이지를 볼 수 없습니다. 서버에서 응답을 보내도록 request 이벤트를 처리하는 콜백 함수 안에 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch05_test3.js

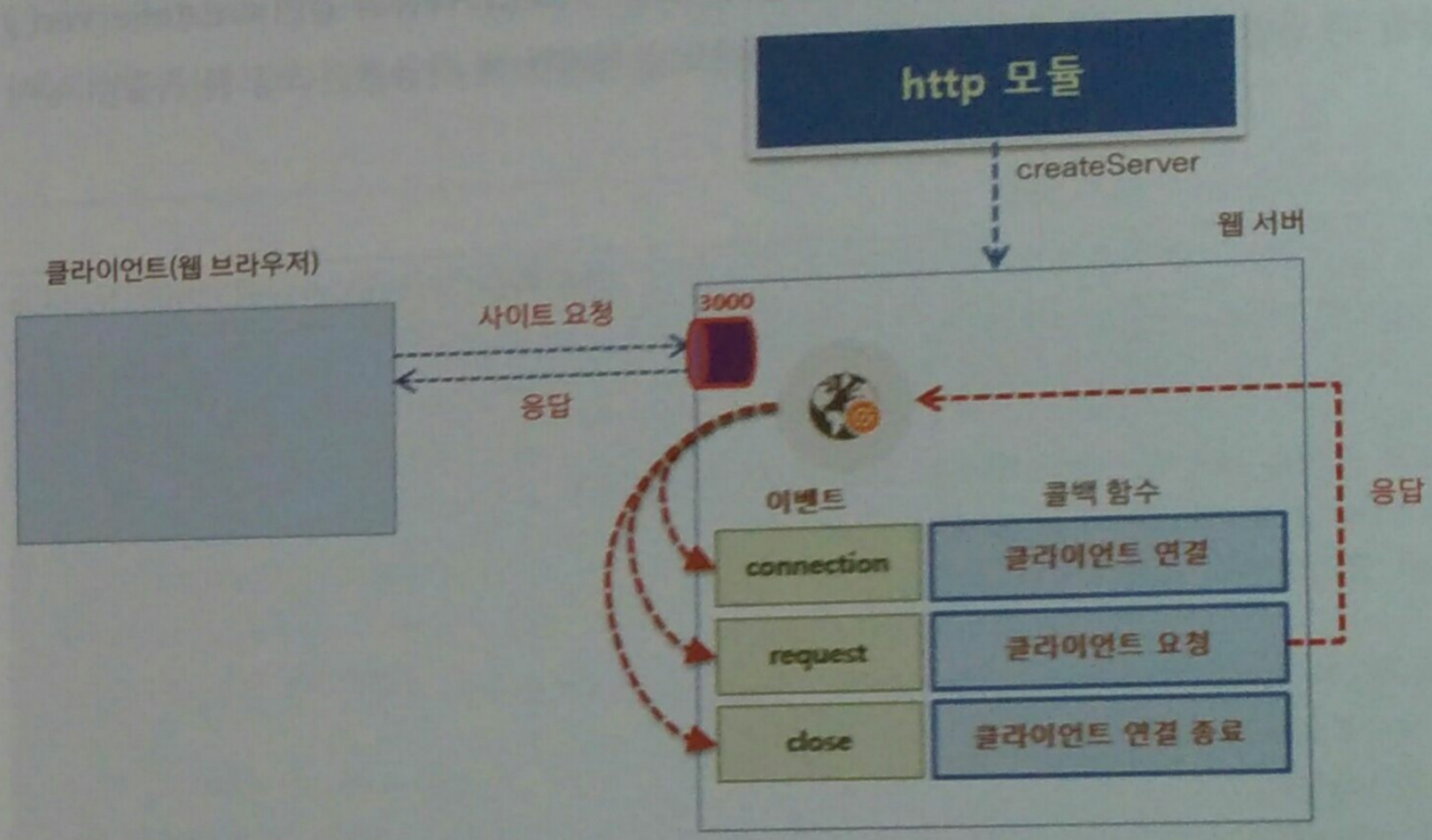
```
……

// 클라이언트 요청 이벤트 처리
server.on('request', function(req, res) {
  console.log('클라이언트 요청이 들어왔습니다.');

  res.writeHead(200, {"Content-Type": "text/html; charset=utf-8"});
  res.write("<!DOCTYPE html>");
  res.write("<html>");
  res.write("  <head>");
  res.write("    <title>응답 페이지</title>");
  res.write("  </head>");
  res.write("  <body>");
  res.write("    <h1>노드제이에스로부터의 응답 페이지</h1>");
  res.write("  </body>");
  res.write("</html>");
  res.end();
});

……
```

웹 브라우저를 열고 http://localhost:3000/ 또는 IP 주소를 확인한 경우에는 IP 주소를 지정하여 주소를 입력하고 서버에 접속하면 응답 페이지가 나타납니다.



웹 브라우저에 보이는 응답 페이지 ▶

res 객체의 writeHead( ), write( ), end( ) 메소드를 사용하면 클라이언트로 응답을 보낼 수 있습니다. 이 중에서 end( ) 메소드는 응답을 모두 보냈다는 것을 의미하며, 일반적으로는 end( ) 메소드가 호출될 때 클라이언트로 응답을 전송합니다. 이렇게 해서 클라이언트가 웹 서버에 요청을 보내고 응답을 받을 수 있는 기능을 만들어 보았습니다. 지금까지 노드로 만든 기능을 정리하면 다음과 같습니다.



▲ 클라이언트에서 요청했을 때 응답을 보내는 과정

on( ) 메소드는 이벤트를 처리할 때 사용하는 가장 기본적인 메소드입니다. 이 메소드로 connection, request, close 이벤트를 처리할 수 있는 콜백 함수를 각각 등록해 두면 상황에 맞게 자동 호출됩니다. 클라이언트가 요청을 해 왔을 때 발생하는 request 이벤트를 처리할 수 있게 등록해 둔 콜백 함수에서는 res 객체를 사용해서 클라이언트로 응답을 보냅니다. res 객체를 사용해서 응답을 보낼 때 사용하는 주요 메소드는 다음과 같습니다.

메소드 이름	설명
writeHead(statusCode [, statusMessage][, headers])	응답으로 보낼 헤더를 만듭니다.
write(chunk[, encoding][, callback])	응답 본문(body) 데이터를 만듭니다. 여러 번 호출될 수 있습니다.
end([data][, encoding][, callback])	클라이언트로 응답을 전송합니다. 파라미터에 데이터가 들어 있다면 이 데이터를 포함시켜 응답을 전송합니다. 클라이언트의 요청이 있을 때 한 번은 호출되어야 응답을 보내며, 콜백 함수가 지정되면 응답이 전송된 후 콜백 함수가 호출됩니다.

[표] 응답 객체의 주요 메소드

request 이벤트를 처리하는 콜백 함수에서 클라이언트로 응답을 보내는 방법에 대해 알아보았습니다. 그런데 request 이벤트를 사용해서 클라이언트로 응답을 보내지 않고 서버 객체를 만들 때 사용한 createServer( ) 메소드 호출 부분에 응답을 보내는 코드를 바로 입력할 수도 있습니다.

ch05_test3.js 파일을 복사하여 ch05_test4.js 파일을 만듭니다. 그리고 다음과 같이 createServer( ) 메소드를 호출할 때 전달하는 콜백 함수에 request 이벤트를 처리할 때 사용했던 코드를 똑같이 넣어 봅니다.

참조 파일 | NodeExample1>ch05_test4.js

```
……

// 클라이언트의 요청을 처리하는 콜백 함수를 파라미터로 전달합니다.
var server = http.createServer(function(req, res) {
  console.log('클라이언트 요청이 들어왔습니다.');

  res.writeHead(200, {"Content-Type": "text/html; charset=utf-8"});
  res.write("<!DOCTYPE html>");
  res.write("<html>");
  res.write("  <head>");
  res.write("    <title>응답 페이지</title>");
  res.write("  </head>");
  res.write("  <body>");
  res.write("    <h1>노드제이에스로부터의 응답 페이지</h1>");
  res.write("  </body>");
  res.write("</html>");
  res.end();
});

……
```

위의 코드를 입력했으면 request 이벤트를 처리하는 코드는 삭제합니다. 파일을 실행한 후 웹 브라우저에서 이 웹 서버에 접속하면 앞의 과정에서 실행한 것과 똑같은 결과가 나타납니다. 이 방법은 request 이벤트를 처리하는 콜백 함수에서 응답을 보내는 것과 다르지 않습니다. 하지만 때에 따라서 이렇게도 만들 수도 있다는 것을 알아 두세요.

클라이언트에서 요청이 있을 때 파일 읽어 응답하기

fs 모듈로 파일을 다루는 방법은 이미 첫째 장에서 알아보았습니다. 파일을 읽기 위해 readFile( ) 메소드를 사용하거나 파일을 스트림 객체로 읽을 수 있습니다. 이제 클라이언트의 요청이 들어왔을 때 파일

을 읽고 응답하는 방법을 알아보겠습니다. 먼저 적당한 이미지 파일을 인터넷에서 다운로드한 후 house.png라는 이름으로 저장합니다.

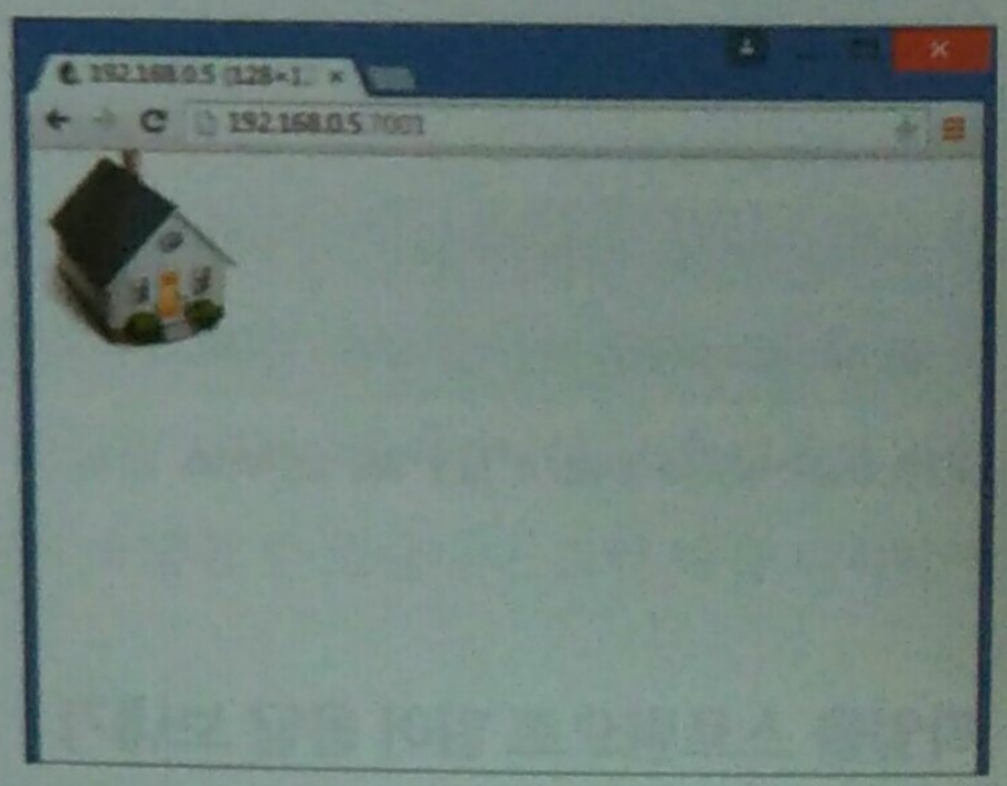
주의! 무료 이미지 파일 찾는 방법: 아이콘파인더(http://www.iconfinder.com) 사이트에 접속한 후 house라는 검색어로 집 이미지를 찾았습니다. 검색된 이미지 중 하나를 다운로드하여 프로젝트 폴더 안에 house.png라는 이름으로 저장하세요.

ch05_test3.js 파일을 복사하여 ch05_test5.js 파일을 만든 후 다음과 같이 입력합니다. 이 코드는 이미지 파일을 읽어 클라이언트 쪽으로 응답을 보냅니다.

참조 파일 | NodeExample1>ch05_test5.js

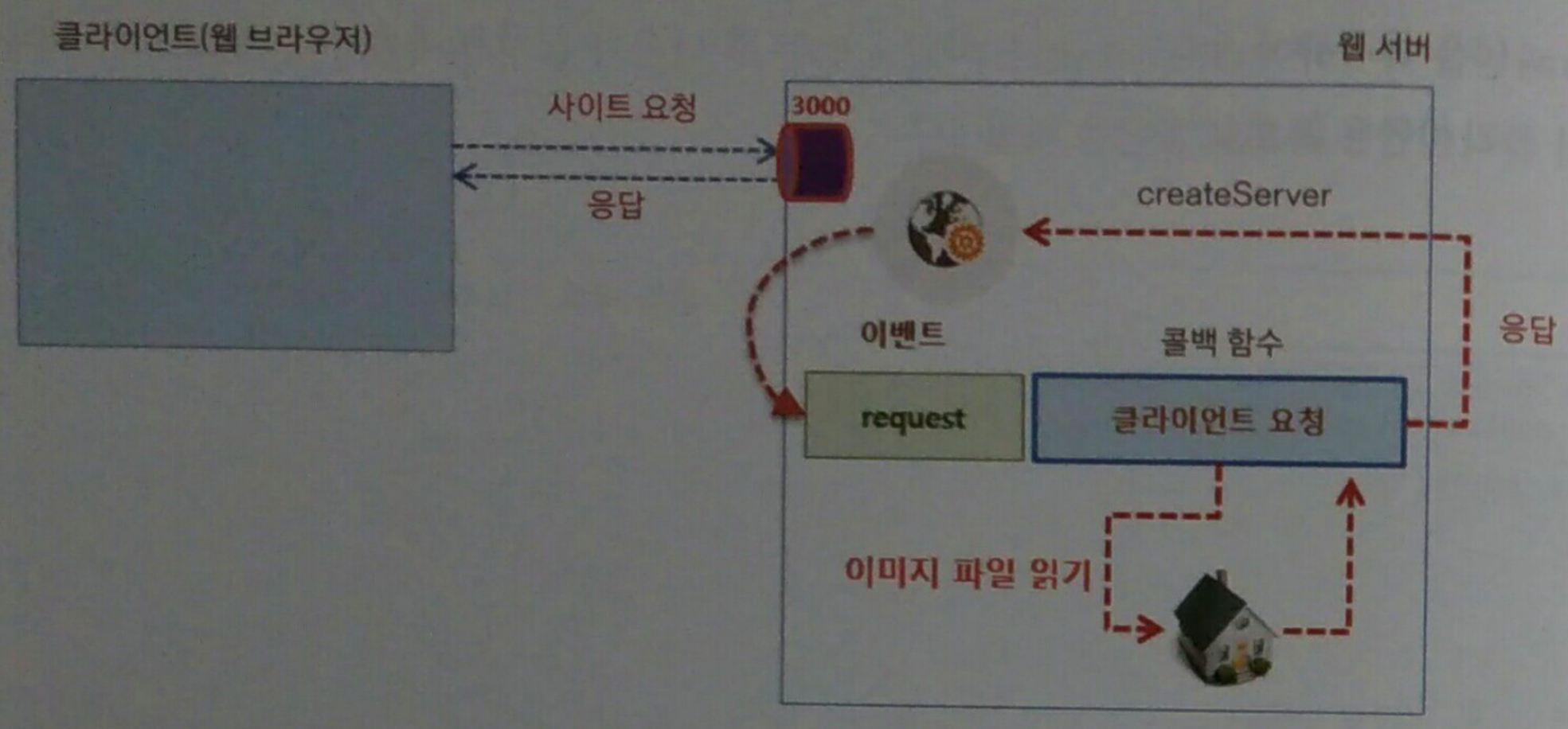
```
var http = require('http');
var fs = require('fs');

.......

// 클라이언트 요청 이벤트 처리
server.on('request', function(req, res) {
  console.log('클라이언트 요청이 들어왔습니다.');

  var filename = 'house.png';
  fs.readFile(filename, function(err, data) {
    res.writeHead(200, {"Content-Type": "image/png"});
    res.write(data);
    res.end();
  });

});

.......
```

파일을 다루어야 하기 때문에 코드의 가장 위쪽에서 require( ) 메소드로 fs 모듈을 읽어 들입니다. on 메소드를 호출하여 request 이벤트를 처리하고, 클라이언트의 요청이 들어오면 readFile( ) 메소드로 house.png 파일을 읽어 들입니다. readFile( ) 메소드는 비동기 방식으로 처리됩니다. 따라서 파일을 모두 읽으면 콜백 함수 안의 data 객체로 파일 내용이 전달됩니다. 콜백 함수 안에서는 응답 객체의 write( ) 메소드를 사용해서 파일 내용을 클라이언트로 전송합니다. 그리고 HTTP 헤더 중에서 Content-Type 헤더 값으로 image/png를 넣어 이미지 데이터임을 표시합니다. 파일을 실행한 후 웹 브라우저를 열고 사이트에 접속하면 다음과 같이 집 이미지가 보입니다.



▲ 이미지 파일을 읽어 웹 브라우저 쪽으로 전송한 결과

코드를 몇 줄 입력하지도 않았는데 웹 브라우저에 집 이미지가 보이니 신기하죠? 이것은 웹 서버를 다루는 코드나 파일을 다루는 코드가 많은 기능을 포함하고 있으면서도 쉽게 사용할 수 있도록 만들어졌기 때문입니다. 이미지 파일을 읽어 응답을 보내는 과정을 다시 정리하면 다음과 같습니다.



▲ 이미지 파일을 읽어 응답을 보내는 과정

이런 이미지 파일 이외에도 일반 텍스트 파일이나 음악 파일 등을 같은 방식으로 클라이언트에 전송할 수 있습니다. 만약 음악 파일을 읽어 응답으로 보내고 싶다면 각각의 Content-Type 헤더 값만 적절한 MIME Type으로 설정하면 됩니다. Content-Type에 설정할 수 있는 대표적인 MIME Type은 다음과 같습니다.

Content Type의 값	설명
text/plain	일반 텍스트 문서
text/html	HTML 문서
text/css	CSS 문서
text/xml	XML 문서
image/jpeg, image/png	JPEG 파일, PNG 파일
video/mpeg, audio/mp3	MPEG 비디오 파일, MP3 음악 파일
application/zip	ZIP 압축 파일

[표] Content-Type에 설정할 수 있는 대표적인 MIME Type

용어! **MIME Type이란?** Multipurpose Internet Mail Extensions의 약어로 메시지의 내용이 어떤 형식인지 알려주기 위해 정의한 인터넷 표준입니다. 메시지의 내용으로는 일반 텍스트, 이미지, 오디오, 비디오 등이 들어갈 수 있으며 이러한 형식을 간단한 값으로 표현합니다.

파일을 스트림으로 읽어 응답 보내기

지금까지 파일을 읽어 응답 보내는 과정을 코드로 만들어 보았는데, 이 과정을 더 간단하게 만들 수 있는 방법이 파이프입니다. 파일은 스트림 객체로 읽어 들일 수 있고 웹 서버의 응답 객체도 스트림으로 데

이터를 전송할 수 있기 때문에 두 개의 스트림은 파이프로 서로 연결할 수 있습니다. 다시 말해, 파일을 스트림 객체로 읽어 들인 후 pipe( ) 메소드로 응답 객체와 연결하면 별다른 코드 없이도 파일에 응답을 보낼 수 있습니다.

ch05_test5.js 파일을 복사하여 ch05_test6.js 파일로 만든 후 클라이언트 요청을 처리하는 콜백 함수의 코드를 다음과 같이 수정합니다.

참조 파일 | NodeExample1>ch05_test6.js

```
……

// 클라이언트 요청 이벤트 처리
server.on('request', function(req, res) {
  console.log('클라이언트 요청이 들어왔습니다.');

  var filename = 'house.png';
  var infile = fs.createReadStream(filename, {flags: 'r'} );

  // 파이프로 연결하여 알아서 처리하도록 설정하기
  infile.pipe(res);

});

……
```

이 파일을 실행한 후 웹 브라우저에서 요청을 보내면 같은 결과를 볼 수 있습니다. 똑같은 기능을 더 적은 양의 코드로 만들었으니 이 방법이 좋다고 생각할 수 있습니다. 하지만 실제로 이 방법으로 코딩을 했을 때 헤더를 설정할 수 없는 등의 제약이 생기므로 필요할 때만 사용하길 권합니다.

파일을 버퍼에 담아 두고 일부분만 읽어 응답 보내기

이번에는 파일을 버퍼에 담아 일정 크기만큼 읽어 응답을 보내는 방법에 대해 알아보겠습니다. 다음 코드는 파일을 스트림으로 읽은 후 일정 크기의 데이터를 읽어 응답을 보내는 방식입니다. 스트림에서 파일을 읽을 때 버퍼의 크기를 정하면 파일을 버퍼 크기만큼씩 읽어 응답할 수 있습니다. 그런 다음 마지막에 end( ) 메소드를 호출하여 응답으로 전송할 수 있습니다.

ch05_test6.js 파일의 클라이언트 요청 처리 부분을 다음과 같이 수정합니다.

```
……

// 클라이언트 요청 이벤트 처리
server.on('request', function(req, res) {
  console.log('클라이언트 요청이 들어왔습니다.');

  var filename = 'house.png';
  var infile = fs.createReadStream(filename, {flags: 'r'} );
  var filelength = 0;
  var curlength = 0;

  fs.stat(filename, function(err, stats) {
    filelength = stats.size;
  });

  // 헤더 쓰기
  res.writeHead(200, {"Content-Type": "image/png"});

  // 파일 내용을 스트림에서 읽어 본문 쓰기
  infile.on('readable', function() {
    var chunk;
    while (null != (chunk = infile.read())) {
      console.log('읽어 들인 데이터 크기 : %d 바이트', chunk.length);
      curlength += chunk.length;
      res.write(chunk, 'utf8', function(err) {
        console.log('파일 부분 쓰기 완료 : %d, 파일 크기 : %d', curlength, filelength);
        if (curlength >= filelength) {
          // 응답 전송하기
          res.end();
        }
      });
    }
  });
});

……
```

코드의 양이 조금 많아졌습니다. 그 이유는 파일을 읽기 전에 먼저 파일을 열고, 그다음에 스트림에서 일정 크기만큼만 읽어 응답을 보내는 방식이기 때문입니다.

응답 객체의 end( ) 메소드를 호출하는 시점은 write( ) 메소드가 종료되는 시점이어야 합니다. 따라서 write( ) 메소드에 콜백 함수를 전달하여 쓰기가 완료된 시점을 확인합니다. 그 콜백 함수 안에서는 파

일의 크기만큼 응답 객체에 쓰기를 모두 완료했는지 확인하고, 응답 객체에 쓰는 작업이 모두 완료되었다면 end( ) 메소드를 호출하여 응답 데이터를 전송합니다. 이 파일을 실행하고 웹 브라우저에서 접속하면 앞의 예에서처럼 집 이미지가 보입니다.

서버에서 다른 웹 사이트의 데이터를 가져와 응답하기

지금까지 일정 크기만큼 파일을 읽어 응답을 보내는 코드를 만들고 실행해 보았습니다. 이제 우리의 관심사는 웹 서버를 만들어 웹 브라우저나 스마트폰 앱에서 들어오는 요청에 응답하는 것입니다. 하지만 서버에서 다시 다른 웹 사이트를 접속하여 데이터를 가져온 후 응답하는 과정이 필요할 때도 있습니다. 이 경우에는 서버에서 HTTP 클라이언트 기능도 사용하게 됩니다. 그럼 간단한 HTTP 클라이언트를 만들려면 어떻게 해야 할까요? http 모듈은 서버 기능 이외에 클라이언트 기능도 제공합니다. 즉, HTTP 클라이언트가 GET과 POST 방식으로 다른 웹 서버에 데이터를 요청할 수 있습니다.

다음은 http 모듈을 사용해 GET 방식으로 다른 사이트에 데이터를 요청하는 코드입니다. ch05_test7.js 파일을 새로 만든 후 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch05_test7.js

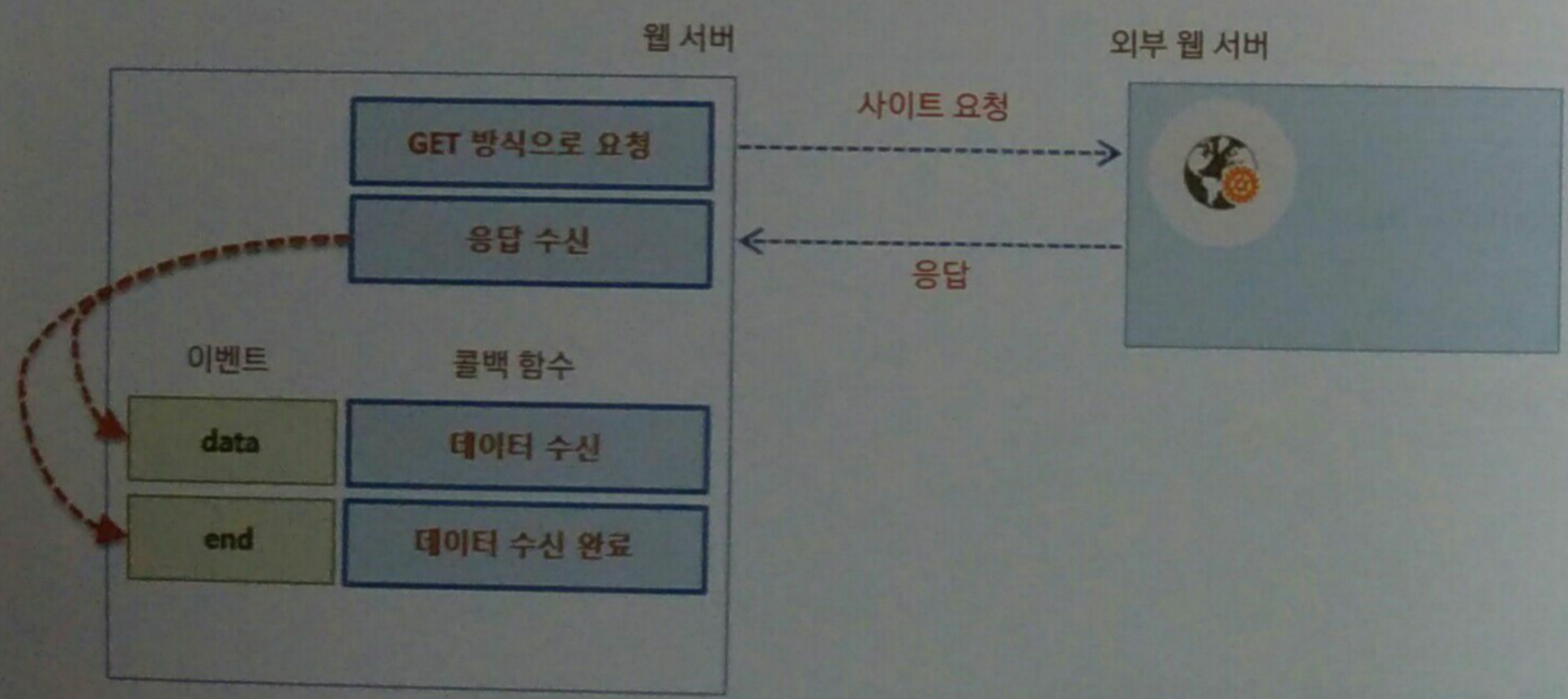
```
var http = require('http');

var options = {
  host: 'www.google.com',
  port: 80,
  path: '/'
};

var req = http.get(options, function(res) {
  // 응답 처리
  var resData = '';
  res.on('data', function(chunk) {
    resData += chunk;
  });

  res.on('end', function() {
    console.log(resData);
  });
});

req.on('error', function(err) {
  console.log("오류 발생 : " + err.message);
});
```

http 객체의 get( ) 메소드를 사용하면 다른 사이트에 요청을 보내고 응답을 받아 처리할 수 있습니다. get( ) 메소드의 첫 번째 파라미터는 다른 사이트의 정보를 담고 있는 객체입니다. 그리고 두 번째 파라미터는 콜백 함수입니다. 응답 데이터를 받을 때는 data 이벤트와 end 이벤트로 처리하면 됩니다. 데이터를 받고 있는 상태에서는 data 이벤트가 발생하므로 이때 받은 데이터를 모두 resData 변수에 담아 둡니다. end 이벤트를 처리하면 응답 데이터를 모두 받은 후에 콘솔 창에 출력할 수 있습니다. 이 파일을 실행하면 다음처럼 구글 사이트에서 받아 온 응답을 볼 수 있습니다.

```
Console  Debug
<terminated> NodeExample1-ch05_test7.js [Node Application] Node.js Process
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.co.kr/?gfe_rd=cr&amp;ei=S2lsVdSjD8nC8Afd9ICgDg">her
</BODY></HTML>
```

▲ HTTP 클라이언트 기능의 GET 방식으로 구글 사이트 데이터 확인하기

웹 서버에서 다른 서버로 요청할 때의 과정을 정리하면 다음과 같습니다.



▲ 웹 서버에서 다른 서버로 요청하여 응답을 받는 과정

다른 서버로부터 응답을 받을 때는 data 이벤트가 발생합니다. 수신 데이터의 용량에 따라 data 이벤트는 한 번 또는 여러 번 발생할 수도 있습니다. 데이터 수신이 완료되면 end 이벤트가 발생합니다.

POST 방식으로 요청할 경우에는 request( ) 메소드를 사용합니다. 이 request( ) 메소드는 get( ) 메소드와 사용 방식이 약간 다릅니다. 특히 요청을 보내려면 요청 헤더와 본문을 모두 직접 설정해야 합니다. 다음은 http 모듈을 사용해 POST 방식으로 다른 사이트에 데이터를 요청하는 코드입니다. ch05-test8.js 파일을 새로 만든 후 다음과 같이 입력합니다.

참조 파일 | NodeExample1>ch05_test8.js

```
var http = require('http');

var opts = {
  host: 'www.google.com',
  port: 80,
  method: 'POST',
  path: '/',
  headers: {}
};


var resData = '';
var req = http.request(opts, function(res) {
  // 응답 처리
  res.on('data', function(chunk) {
    resData += chunk;
  });

  res.on('end', function() {
    console.log(resData);
  });
});

opts.headers['Content-Type'] = 'application/x-www-form-urlencoded';
req.data = "q=actor";
opts.headers['Content-Length'] = req.data.length;

req.on('error', function(err) {
    console.log("오류 발생 : " + err.message);
});

// 요청 전송
req.write(req.data);
req.end();
```

정박사의 한마디

GET 방식과 POST 방식의 차이점은 무엇일까요?

웹 서버에 요청을 보내는 방식은 GET, POST, PUT, DELETE 등 여러 가지가 있습니다. 인터넷 표준으로 정해 둔 이러한 요청 방식을 메소드(Method)라고도 부릅니다. HTTP 요청 포맷을 보면 크게 헤더(Header)와 본문(Body) 부분으로 나뉘는데 GET 방식은 헤더 부분에 요청 정보들을 넣어 보내고 POST 방식은 본문 부분에 요청 정보를 넣어 보냅니다. 이 두 가지 방식은 기능상으로는 요청 정보를 보낸다는 점에서 같지만 보안 등의 이슈가 있을 때나 파일을 요청 정보로 넣어 보내야 하는 경우(파일 업로드)에는 POST 방식을 주로 사용합니다.

구글 사이트에 요청할 때 필요한 요청 파라미터는 요청 객체의 data 속성으로 설정합니다. 이 속성 값에 따라 Content-Length 헤더의 값이 달라지므로 request( ) 메소드로 만든 요청 객체에 이 정보를 추가로 설정합니다. 요청할 때는 write( ) 메소드로 요청 본문 데이터를 req 객체에 쓴 후 end( ) 메소드를 사용해 전송합니다. 이 파일을 실행하면 구글 사이트에서는 POST 요청을 받지 못하므로 오류가 표시됩니다. 하지만 POST 메소드로 요청할 수 있는 다른 사이트에서는 정상적인 응답을 받을 수 있습니다.

지금까지 간단한 웹 서버를 만들어 보았습니다. 웹 서버를 만들 때 사용하는 http 모듈의 기능은 더 많습니다. 하지만 실무에서 웹 서버를 구성할 때는 express 모듈을 주로 사용하므로 http 모듈은 이 정도만 이해해도 됩니다. 다만 express 모듈을 사용하더라도 express 모듈은 http 모듈을 기초로 만들어진 것이므로 이 장에서 설명한 내용은 어느 정도 이해하고 넘어가세요.

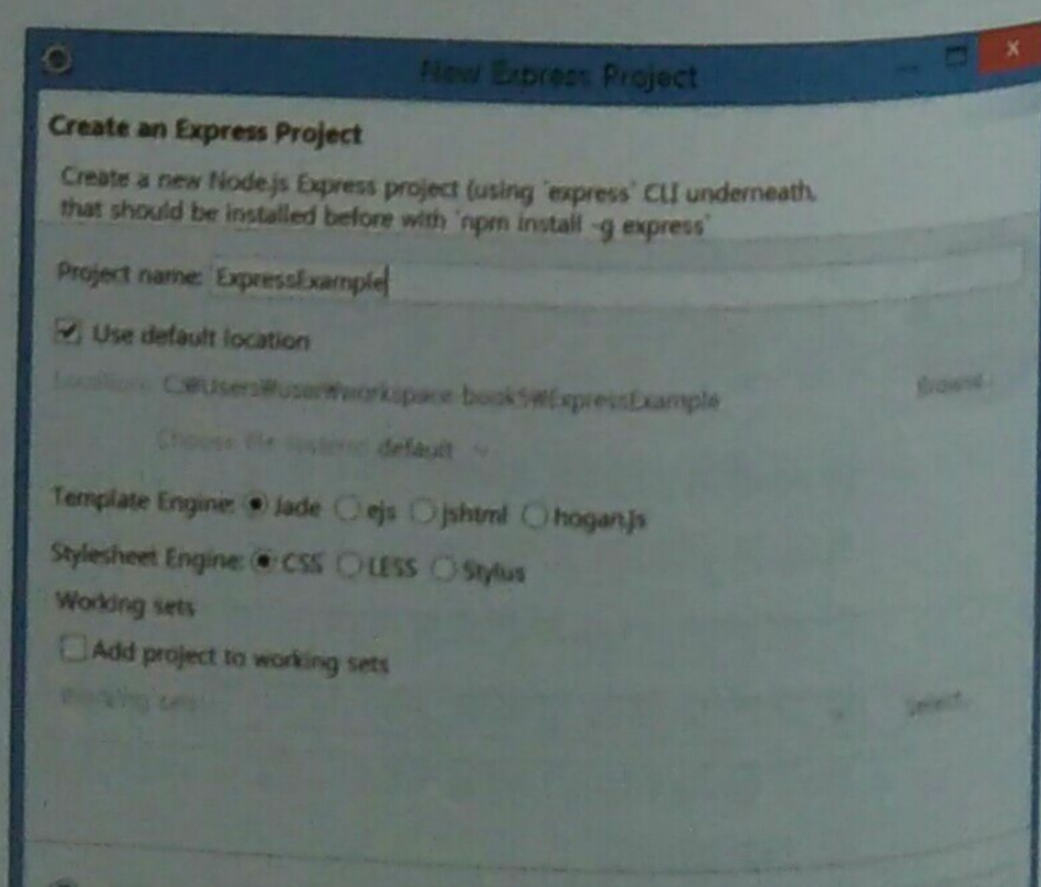
05-2 | 익스프레스로 웹 서버 만들기

http 모듈만 사용해서 웹 서버를 구성할 때는 많은 것들을 직접 만들어야 합니다. 하지만 직접 만들어야 하는 코드가 많다면 시간과 노력도 많이 든다는 문제가 생깁니다. 이 문제를 해결하기 위해 만들어진 것이 **익스프레스(Express)**입니다. express 모듈을 사용하면 간단한 코드로 웹 서버의 기능을 구현할 수 있습니다. 특히 익스프레스에서 제공하는 **미들웨어**와 **라우터**를 사용하면 여러분이 만들어야 하는 각각의 기능을 훨씬 편리하게 구성할 수 있습니다.

익스프레스를 사용하여 새로운 프로젝트를 만들면 프로젝트 안에 여러 가지 파일들이 자동으로 만들어집니다. 이 프로젝트를 기초로 하여 웹 서버의 기능을 하나씩 살펴보겠습니다.

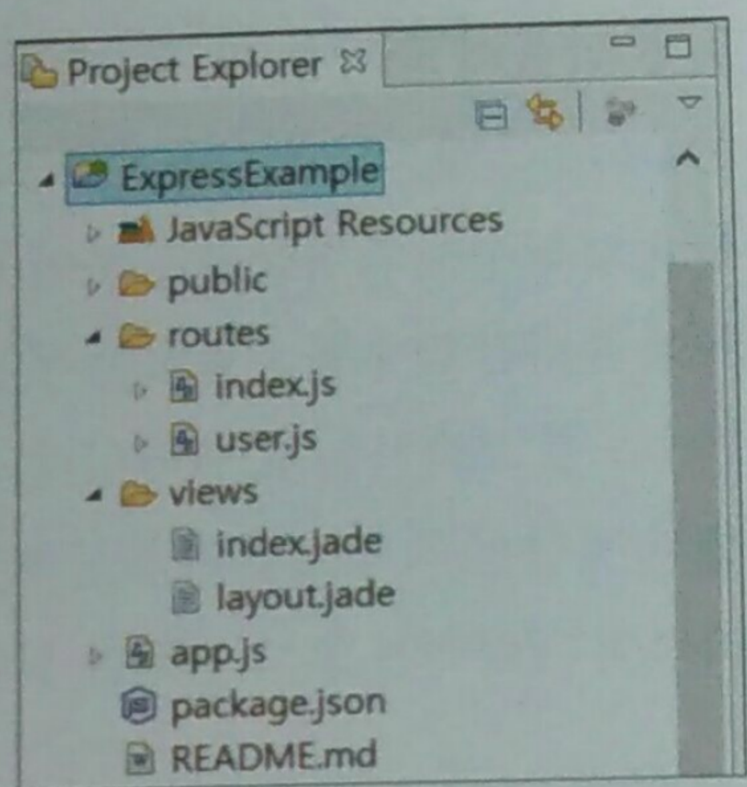
새로운 익스프레스 프로젝트 만들기

❶ 먼저 익스프레스를 사용할 수 있도록 새로운 프로젝트를 하나 만듭니다. 이클립스를 실행한 다음 [File → New → Node.js Express Project] 메뉴를 누르면 새로운 익스프레스 프로젝트를 만들 수 있는 대화상자가 나타납니다. Project name 항목에 프로젝트 이름으로 ExpressExample을 입력한 후 [Finish] 버튼을 누릅니다.



새로운 익스프레스

❷ 새로운 프로젝트 안에는 여러 가지 파일과 폴더들이 자동으로 만들어집니다.



새로 만든 프로젝트 안에 자동으로 만들어진 파일과 폴더들 ▶

자동으로 만들어진 파일 중에서 시작점이 되는 파일은 app.js 파일입니다. app.js 파일을 더블클릭해서 열면 가장 먼저 require( ) 메소드를 사용해 모듈을 불러들이는 코드를 볼 수 있습니다.

참조 파일 | ExpressExample>app.js

```
var express = require('express')
  , routes = require('./routes')
  , user = require('./routes/user')
  , http = require('http')
  , path = require('path');

……
```

./routes와 ./routes/user는 **상대 패스**로 지정하기 위해 모듈 이름 앞에 ./가 붙어 있습니다. 그러나 이 둘을 제외한 express, http, path 모듈은 ./가 없이 모듈 이름만 사용합니다. 이것은 노드에 내장된 모듈 또는 npm으로 설치한 외장 모듈일 때는 상대 패스가 아닌 이름만 지정하도록 설정되어 있기 때문입니다. 따라서 이 프로젝트 안에 직접 만들어 넣은 모듈 파일은 두 개라는 것을 알 수 있습니다.

첫 번째로 보이는 express 모듈은 웹 서버를 위해 만들어진 것으로 http 모듈 위에서 동작합니다. 이 때문에 express 모듈을 사용할 때는 항상 http 모듈도 함께 불러들여야 합니다. path 모듈은 파일 패스를 다루는 데 사용합니다. 상대 패스를 사용한 모듈 중에서 ./routes/user라고 지정된 것은 [routes] 폴더 안에 있는 user.js 파일을 불러들이고 ./routes라고 지정된 것은 [routes] 폴더 안에 있는 index.js 파일을 불러들입니다.

정박사의 한마디

직접 만든 모듈 파일을 불러들일 때 파일을 직접 지정할 수도 있고 폴더를 지정할 수도 있어요

모듈 파일을 만든 후 require( ) 메소드로 모듈을 불러들일 때는 자바스크립트 파일을 확장자 없이 지정합니다. 만약 파일을 지정하지 않고 폴더를 지정하면 그 폴더 안에 있는 index.js 파일을 불러옵니다.

직접 만들어 추가한 이 두 개의 파일에 대해서는 나중에 다시 살펴보겠습니다. 우선 app.js 파일에서 require( ) 메소드가 사용된 코드 아래쪽을 보면 다음과 같이 express( ) 함수를 호출하여 반환된 객체를 app 변수에 지정하는 부분이 보입니다. 그 아래에는 app.set( ), app.use( ), app.get( )과 같이 app 객체에 정의된 함수를 호출하는 부분이 있습니다. 마지막 부분에서는 http 모듈에 정의된 createServer( ) 메소드를 호출하는데, 파라미터로 app 변수를 전달하고 있습니다. 그리고 createServer( ) 메소드를 호출하여 만든 객체에 들어 있는 listen( ) 메소드를 호출하면 웹 서버가 시작됩니다. 주요 코드 부분만 정리하여 개략적인 코드 구조를 살펴보면 다음과 같습니다.

참조 파일 | ExpressExample>app.js

```
……

var app = express();

// 속성 설정
app.set('port', process.env.PORT || 3000);

……

// 미들웨어 사용 설정
app.use(express.favicon());

……

// 클라이언트 요청 처리
app.get('/', routes.index);
app.get('/users', user.list);

// 서버 실행
http.createServer(app).listen(app.get('port'), function() {
  console.log('Express server listening on port ' + app.get('port'));
});
```

이전 단계에서 http 모듈로 웹 서버를 만들 때 createServer( ) 메소드로 웹 서버 객체를 만들고, listen( ) 메소드를 호출하여 클라이언트의 요청을 대기하도록 설정했었습니다. 여기에서도 마지막 줄의 코드는 같습니다. 한 가지 차이점은 createServer( ) 메소드에 전달되는 파라미터로 app 객체를 전달한다는 것입니다. 이 app 객체는 express( ) 메소드 호출로 만들어지는 **익스프레스 서버 객체**입니다.

이 익스프레스 서버 객체가 가지고 있는 주요 메소드들은 다음과 같습니다.

메소드 이름	설명
set(name, value)	서버 설정을 위한 속성을 지정합니다. set( ) 메소드로 지정한 속성은 get( ) 메소드로 꺼내어 확인할 수 있습니다.
get(name)	서버 설정을 위해 지정한 속성을 꺼내 옵니다.
use([path,] function [, function...])	미들웨어 함수를 사용합니다.
get([path,] function)	특정 패스로 요청된 정보를 처리합니다.

[표] express( ) 메소드를 호출하여 만든 app 객체의 주요 메소드

set( ) 메소드는 웹 서버의 환경을 설정하는 데 필요한 메소드입니다. 만약 title 속성을 app 객체에 넣어 두었다가 필요할 때 꺼내어 사용하고 싶다면 app.set('title', 'My App')처럼 set( ) 메소드를 호출하여 넣어 둘 수 있습니다. 그런데 set( ) 메소드로 설정한 속성의 이름이 미리 정해진 이름이라면 웹 서버의 환경 설정에 영향을 미칩니다. 서버 설정을 위해 미리 정해진 주요 속성 이름은 다음과 같습니다.

속성 이름	설명
env	서버 모드를 설정합니다.
views	뷰들이 들어 있는 폴더 또는 폴더 배열을 설정합니다.
view engine	디폴트로 사용할 뷰 엔진을 설정합니다.

[표] 서버 설정을 위해 미리 정해진 app 객체의 주요 속성

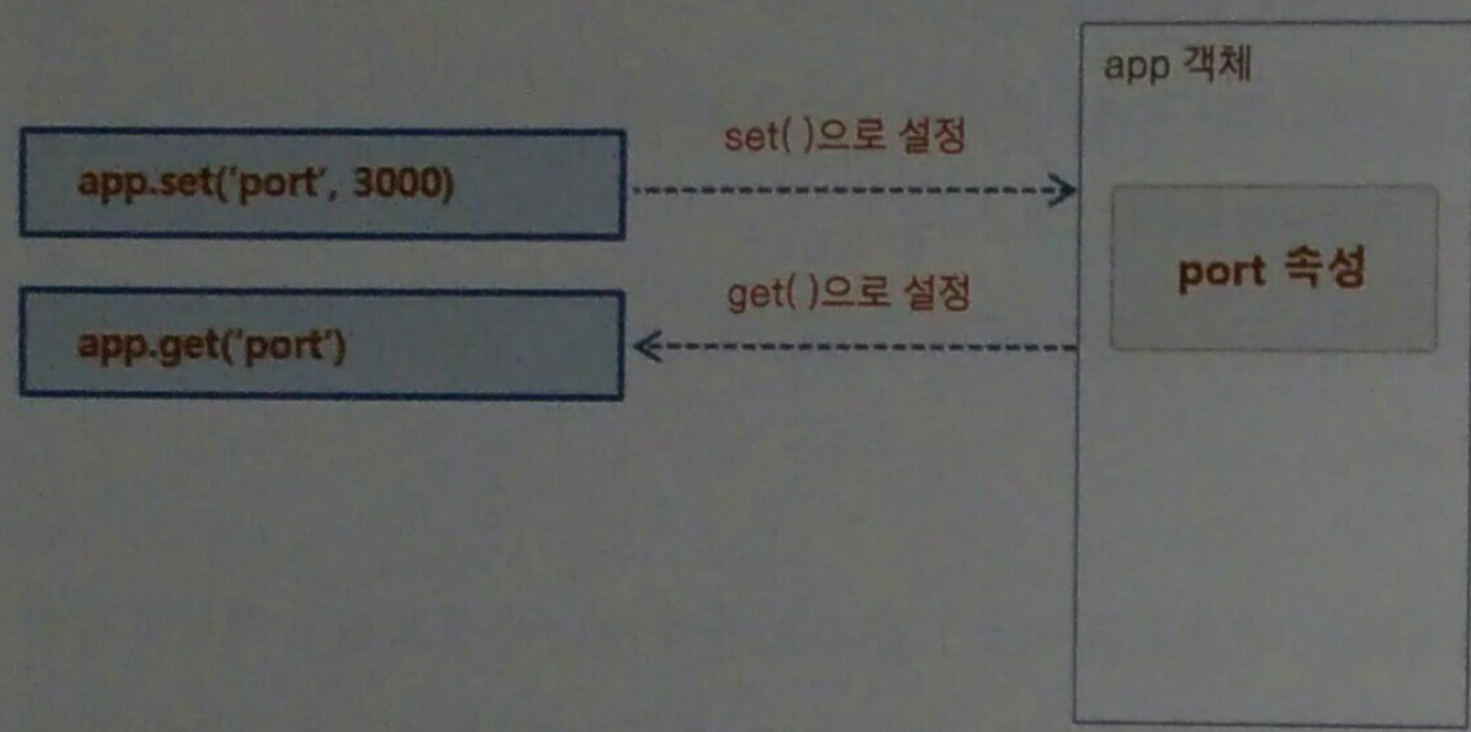
앞의 코드를 보면 익스프레스 서버를 설정하기 위해 set( ) 메소드를 세 번 사용하였습니다. 설정한 속성의 이름은 다음 소스 코드에서 확인할 수 있듯이 port, views, view engine입니다.

참조 파일 | ExpressExample>app.js

```
……

// 속성 설정
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');

……
```

첫 번째 속성의 이름은 port입니다. 이 속성은 process.env 객체에 PORT 속성이 있으면 그 속성을 사용하고, 없으면 3000번 포트 번호를 사용하도록 설정합니다. 이렇게 설정한 포트 번호는 코드의 마지막

부분에서 listen( ) 메소드를 호출할 때 app.get('port')와 같이 포트 속성을 꺼내 온 후 파라미터로 전달됩니다. 결국 port 속성은 웹 서버의 포트를 지정하는 데 사용하는 정보입니다.



▲ 익스프레스 서버 객체인 app에 속성을 넣거나 빼기

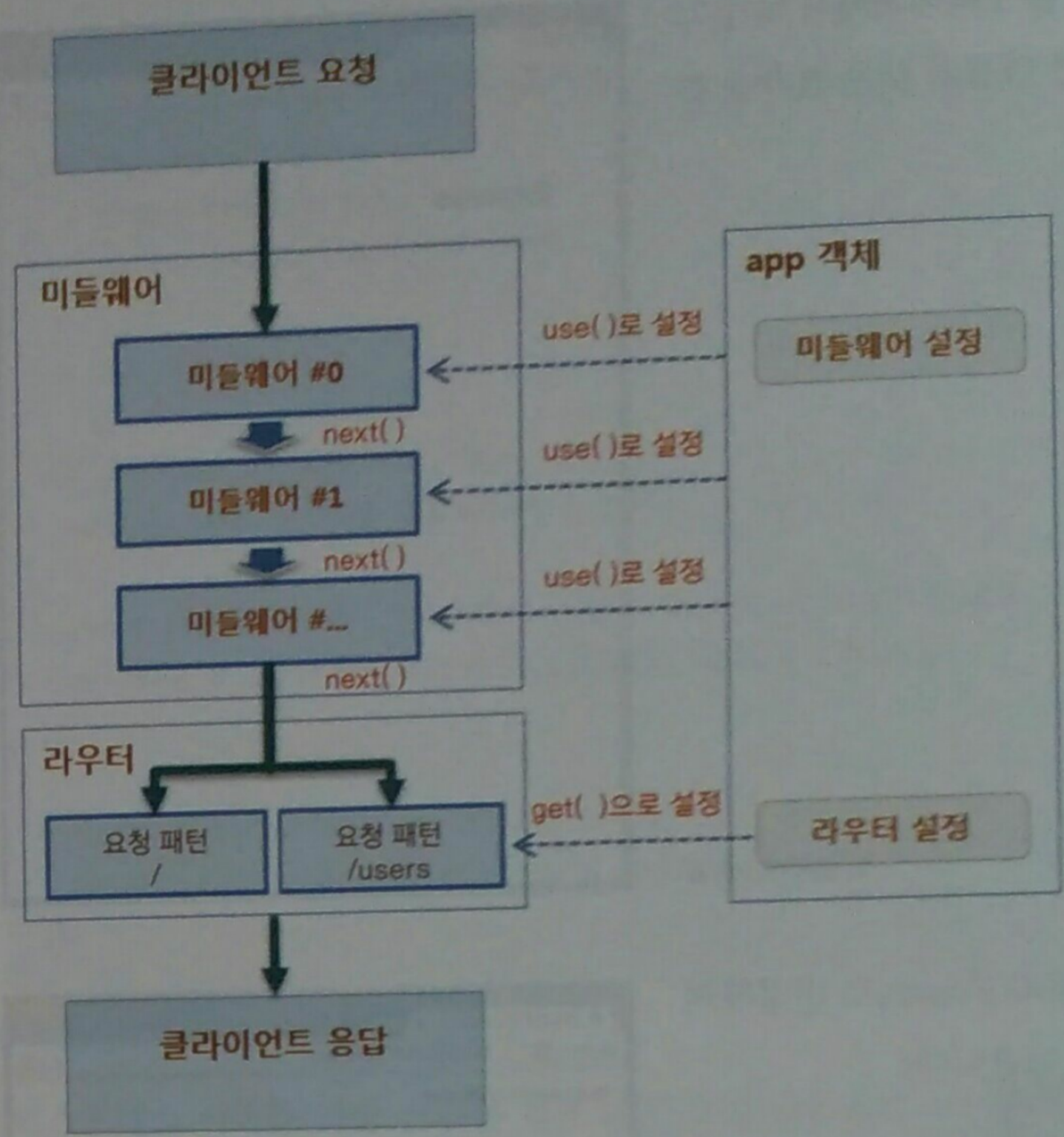
두 번째 views 속성은 뷰들이 들어 있는 디렉터리를 설정하는 것입니다. __dirname 전역 변수로 현재 프로젝트의 폴더 위치를 확인한 후 /views 패스 정보를 붙여 [views] 폴더 안에 뷰 파일들을 넣어 둘 수 있도록 설정합니다. 실제로 [views] 폴더 안을 보면 index.jade와 layout.jade 파일이 있습니다. 이 파일을 뷰 파일(View file)이라고 하는데, 클라이언트에 응답 보낼 웹 문서에서 템플릿 역할을 합니다. 이 내용에 대해서는 나중에 자세히 알아볼 것입니다. views 속성의 속성 값은 다음과 같이 path 모듈을 사용해 바꿀 수도 있습니다.

```
path.join(__dirname, 'views');
```

view engine 속성은 뷰 엔진을 설정하는 것으로 jade나 ejs를 많이 사용합니다. 클라이언트에 응답을 보낼 때 사용하는 템플릿은 여러 종류가 있는데, 그 종류를 결정하는 것이 바로 뷰 엔진입니다. 뷰 엔진에 대해서도 나중에 다시 알아볼 것입니다.

use( ) 메소드를 사용한 미들웨어 설정 알아보기

지금까지 set( ) 메소드로 속성을 설정하는 방법을 알아보았습니다. 이제 use( ) 메소드를 사용하여 미들웨어를 설정하는 방법에 대해 살펴보겠습니다. 노드에서는 미들웨어를 사용하여 필요한 기능을 순차적으로 실행할 수 있습니다. 익스프레스에서 미들웨어를 설정하여 사용하는 방식은 다음과 같습니다.

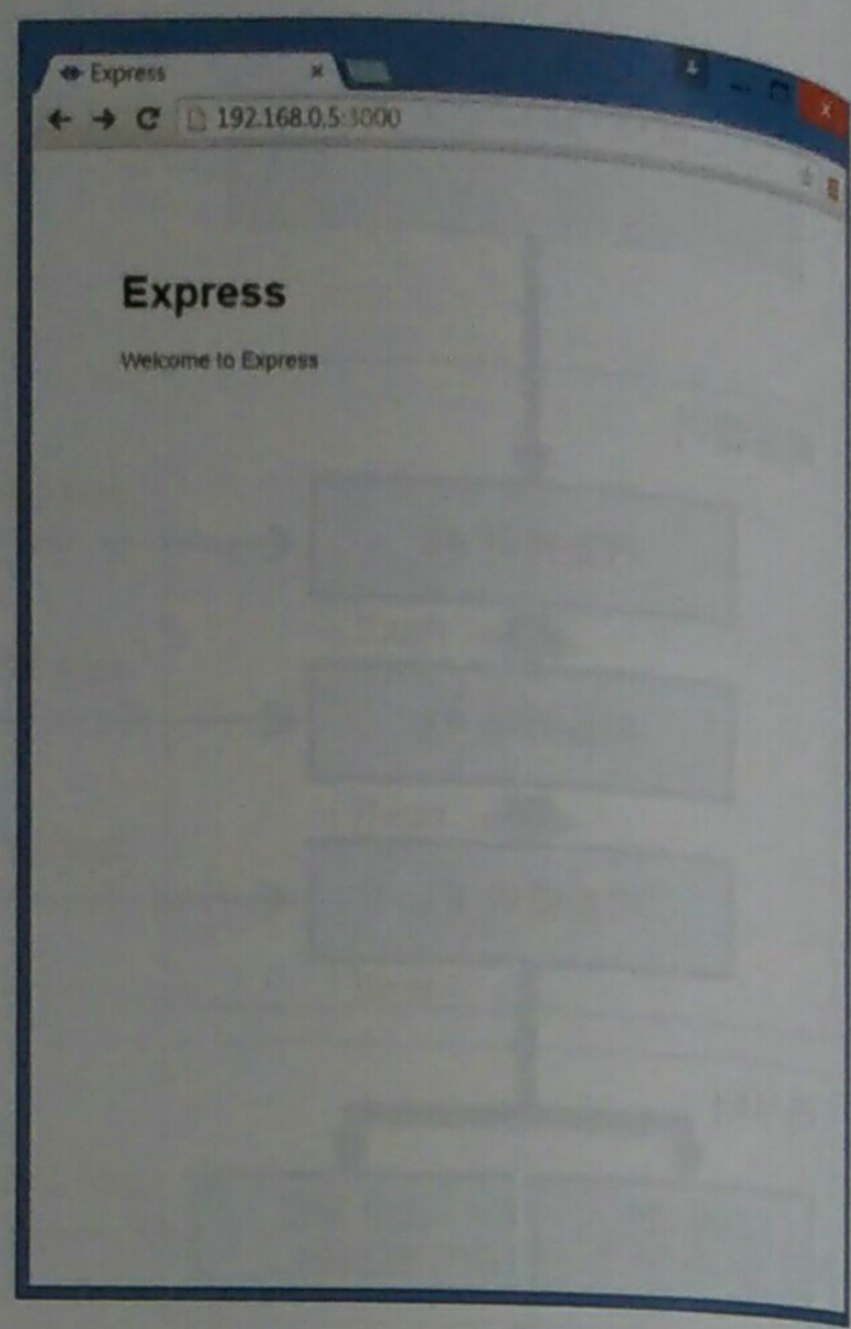


▲ 익스프레스에서 미들웨어와 라우터를 사용하는 방식

익스프레스에서는 미들웨어 이외에 라우터도 사용하는데 미들웨어나 라우터는 하나의 독립된 기능을 가진 함수라고 생각할 수 있습니다. 다시 말해, 익스프레스에서는 웹 요청과 응답에 관한 정보를 사용해 필요한 처리를 진행할 수 있도록 독립된 함수로 분리합니다. 이렇게 분리한 각각의 것들을 미들웨어라고 부릅니다. 예를 들어, 클라이언트에서 요청했을 때 로그로 남기는 간단한 기능을 함수로 만든 후 use( ) 메소드를 사용해 미들웨어로 등록해 두면, 모든 클라이언트 요청이 이 미들웨어를 거치면서 로그를 남기게 됩니다. 각각의 미들웨어는 **next( ) 메소드**를 호출하여 그다음 미들웨어가 처리할 수 있도록 순서를 넘길 수 있습니다.

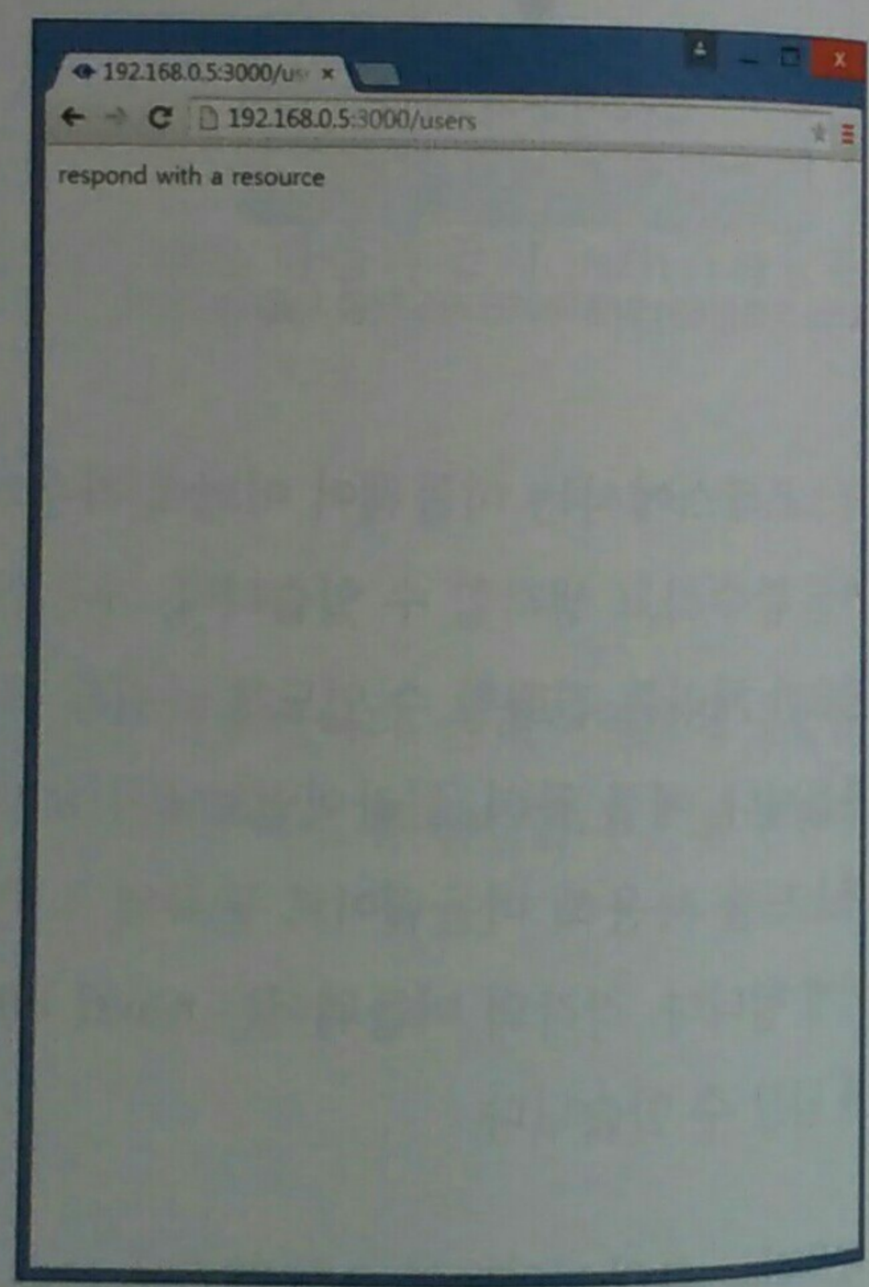
라우터는 클라이언트의 요청 패스를 보고 이 요청 정보를 처리할 수 있는 곳으로 기능을 전달해 주는 역할을 합니다. 이러한 역할을 흔히 **라우팅(Routing)**이라고 부르는데, 클라이언트의 요청 패스에 따라 각각을 담당하는 함수로 분리시키는 것입니다. 예를 들어, 클라이언트가 /users 패스로 요청한다면 이에 대한 응답 처리를 하는 함수를 별도로 분리해서 만든 다음 get( ) 메소드를 호출하여 라우터로 등록할 수 있습니다. 그러면 등록해 둔 라우터 정보로 찾은 함수가 호출되며, 이 함수 안에서 클라이언트로 응답을 보낼 수 있습니다.

app.js 파일을 선택하여 실행한 후 웹 브라우저에서 http://localhost:3000 사이트로 접속하면 다음과 같은 결과를 볼 수 있습니다.



/로 요청한 경우▶

웹 브라우저에서 http://localhost:3000/users로 변경하여 접속하면 다음과 같은 결과를 볼 수 있습니다.



/users로 요청한 경우▶

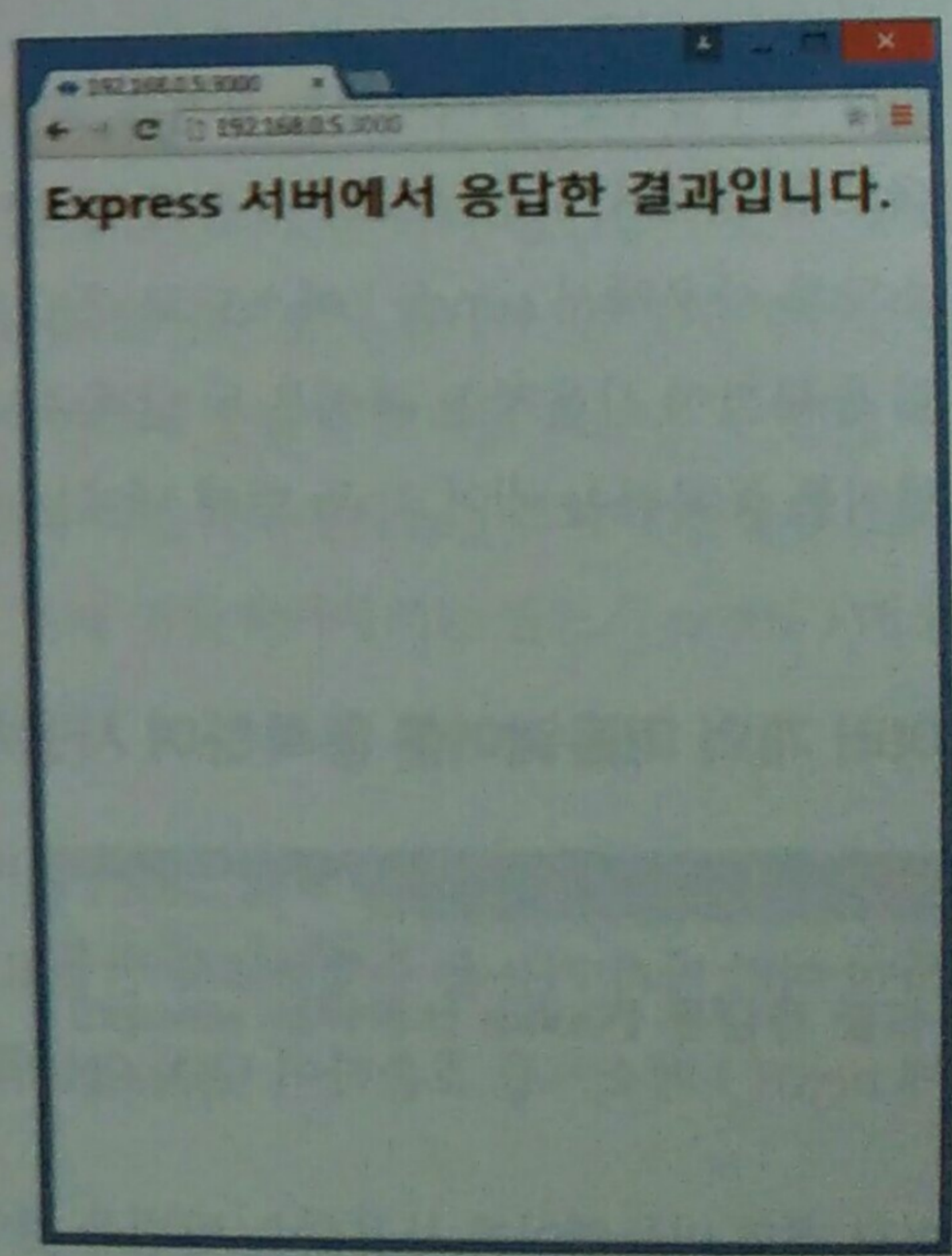
미들웨어 직접 만들어 보기

익스프레스에서 미들웨어와 라우터 사용 방식을 어느 정도 이해했다면, 최소한의 코드만 입력하여 간단한 기능을 직접 만들어 볼 수 있습니다. 이클립스의 왼쪽 프로젝트 영역에서 ExpressExample 프로젝트를 선택한 후 마우스 오른쪽 버튼을 누릅니다. [New → Javascript Source File] 메뉴를 눌러 새로운 자바스크립트 파일을 추가합니다. 새로운 자바스크립트 파일의 이름은 app2.js로 하고 파일이 만들어지면 다음과 같이 입력합니다.

참조 파일 | ExpressExample>app2.js

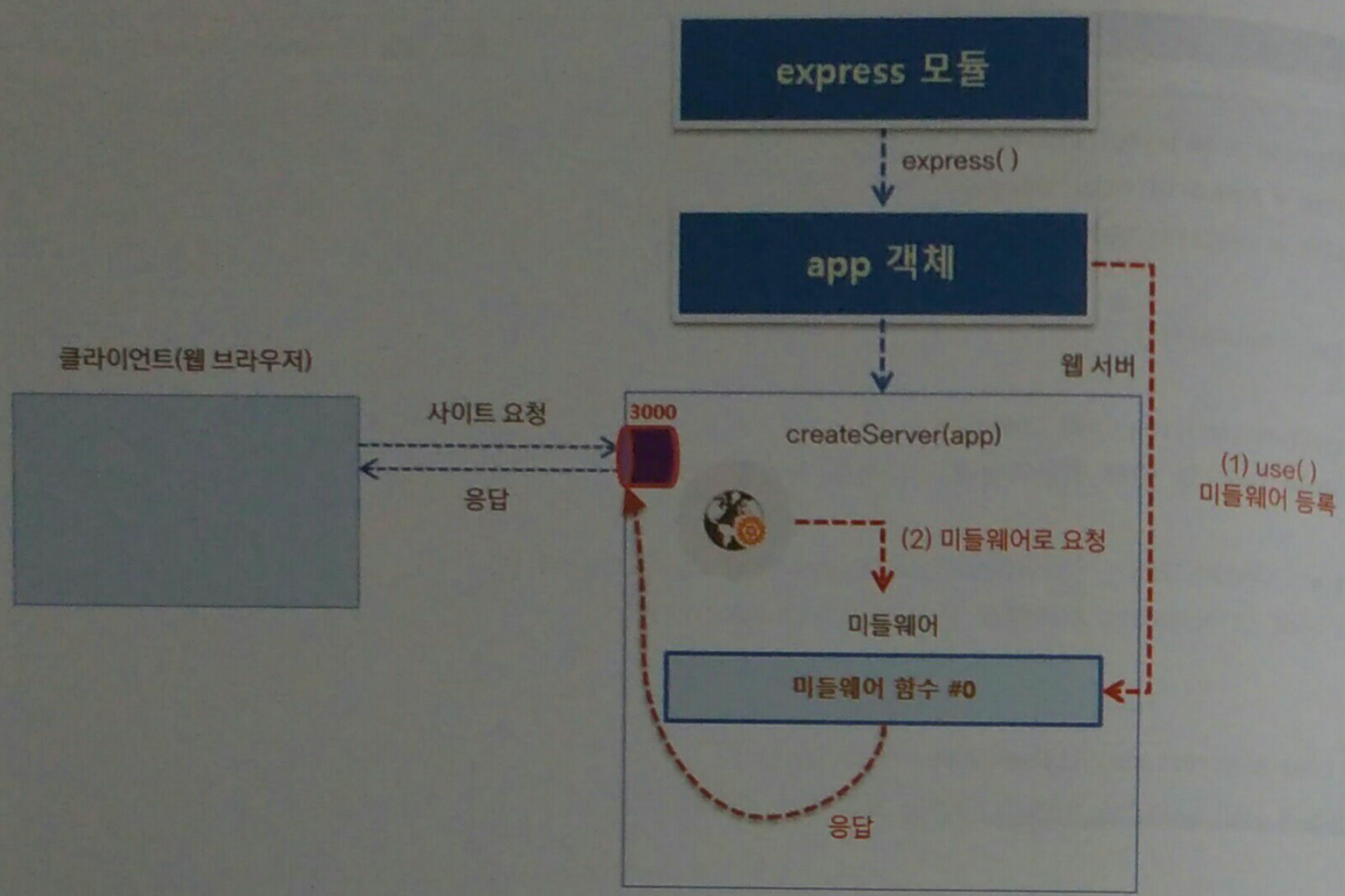
```
var express = require('express')
  , http = require('http')
  , path = require('path');

var app = express();

app.use(function(req, res, next) {
  console.log('첫 번째 미들웨어에서 요청을 처리함.');

  res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
  res.end('<h1>Express 서버에서 응답한 결과입니다.</h1>');
});

http.createServer(app).listen(3000, function() {
  console.log('Express 서버가 3000번 포트에서 시작됨.');
});
```

익스프레스 서버를 가장 간단한 형태로 만들었으며 use( ) 메소드를 호출하여 미들웨어를 하나 등록했습니다. use( ) 메소드로 등록한 함수들은 등록 순서에 따라 클라이언트 요청을 처리합니다. 여기에서는 하나의 미들웨어만 등록했으므로 하나만 호출되며, 이 미들웨어에서 클라이언트로 응답을 보냅니다. 웹 브라우저를 열고 사이트에 접속하면 다음과 같은 결과를 볼 수 있습니다.



사이트에 접속하여 미들웨어에서 처리한 결과를 본 경우 ▶

이 코드가 익스프레스로 웹 서버를 만드는 가장 간단한 형태입니다. 사용자 입장에서 보면, 웹 브라우저에서 웹 서버로 페이지를 요청하고 웹 서버에서 응답을 받아 웹 문서로 볼 수 있게 되었습니다. 그럼에도 불구하고 이러한 기능을 구현하는 코드는 그리 길지 않습니다. 코드가 어떻게 구성되어 있는지 정리해 보면 다음과 같습니다.



▲ 익스프레스로 웹 서버를 만들 때 사용되는 가장 간단한 구성

미들웨어 함수는 클라이언트 요청을 전달받을 수 있습니다. 클라이언트 요청은 등록된 미들웨어를 순서대로 통과하며, 요청 정보를 사용해 필요한 기능을 수행할 수 있습니다. 만약 미들웨어 함수가 end( ) 메소드를 사용해서 send( ) 메소드로 응답을 보내면 처리 과정은 완전히 끝납니다. 이렇듯 미들웨어를 직접 등록하여 사용하는 과정은 두 단계로 나눌 수 있습니다. 첫 번째 단계는 use( ) 메소드를 사용해 미들웨어를 등록하는 것이고, 두 번째 단계는 클라이언트의 요청 정보를 처리하는 것입니다.

여러 개의 미들웨어를 등록하여 사용하는 방법 알아보기

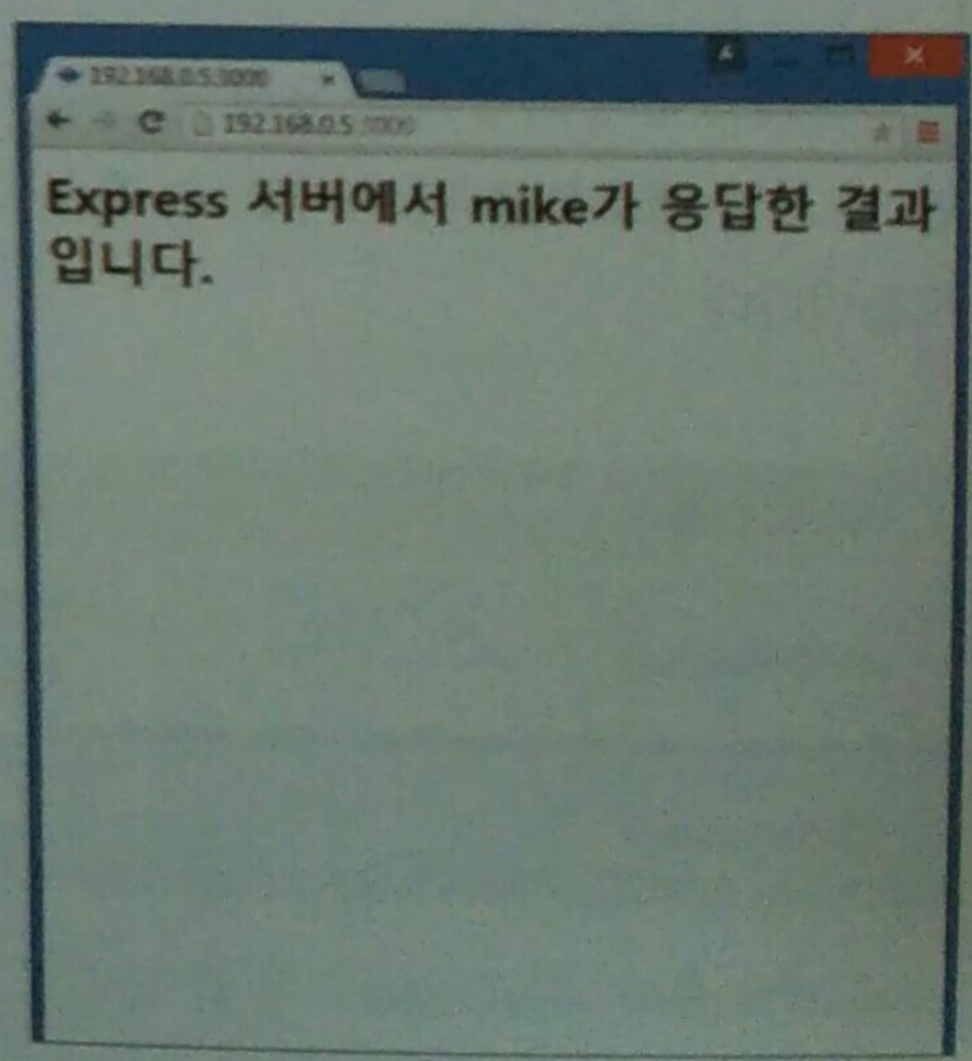
만약 하나의 미들웨어에서 클라이언트에 바로 응답을 보내는 방식이 아니라, 여러 개의 미들웨어를 등록하여 여러 가지 기능을 수행하도록 만들고 싶다면 어떻게 해야 할까요? 각각의 미들웨어 안에서 마지막에 next( ) 메소드를 호출하여 다음 미들웨어로 처리 결과를 넘겨주어야 합니다.

여러 개의 미들웨어를 사용하는 방법을 알아보기 위해 app2.js 파일을 복사하여 app3.js 파일을 만듭니다. 그런 다음 두 개의 미들웨어를 등록하는 코드를 입력합니다.

```
……

app.use(function(req, res, next) {
  console.log('첫 번째 미들웨어에서 요청을 처리함.');

  req.user = 'mike';

  next();
});

app.use('/', function(req, res, next) {
  console.log('두 번째 미들웨어에서 요청을 처리함.');

  res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
  res.end('<h1>Express 서버에서 ' + req.user + '가 응답한 결과입니다.</h1>');

});

……
```

첫 번째 미들웨어에서는 req 객체에 user 속성을 추가하고 그 값으로 문자열을 하나 넣었습니다. 이 정보를 사용자 정보라고 한다면 첫 번째 미들웨어에서 사용자 정보를 설정한 것이 됩니다. 만약 클라이언트에 응답을 보낼 때 사용자 정보를 같이 알려주고 싶은데 클라이언트에 응답을 보내는 기능은 두 번째 미들웨어에서 담당한다고 가정해 보겠습니다. 이런 경우에 두 번째 미들웨어에서 req 객체에 설정된 user 속성을 확인하여 사용자 정보를 사용할 수 있습니다. 이때 첫 번째 미들웨어에서는 반드시 next( ) 메소드를 호출하여 두 번째 미들웨어로 처리 순서를 넘겨주어야 합니다.

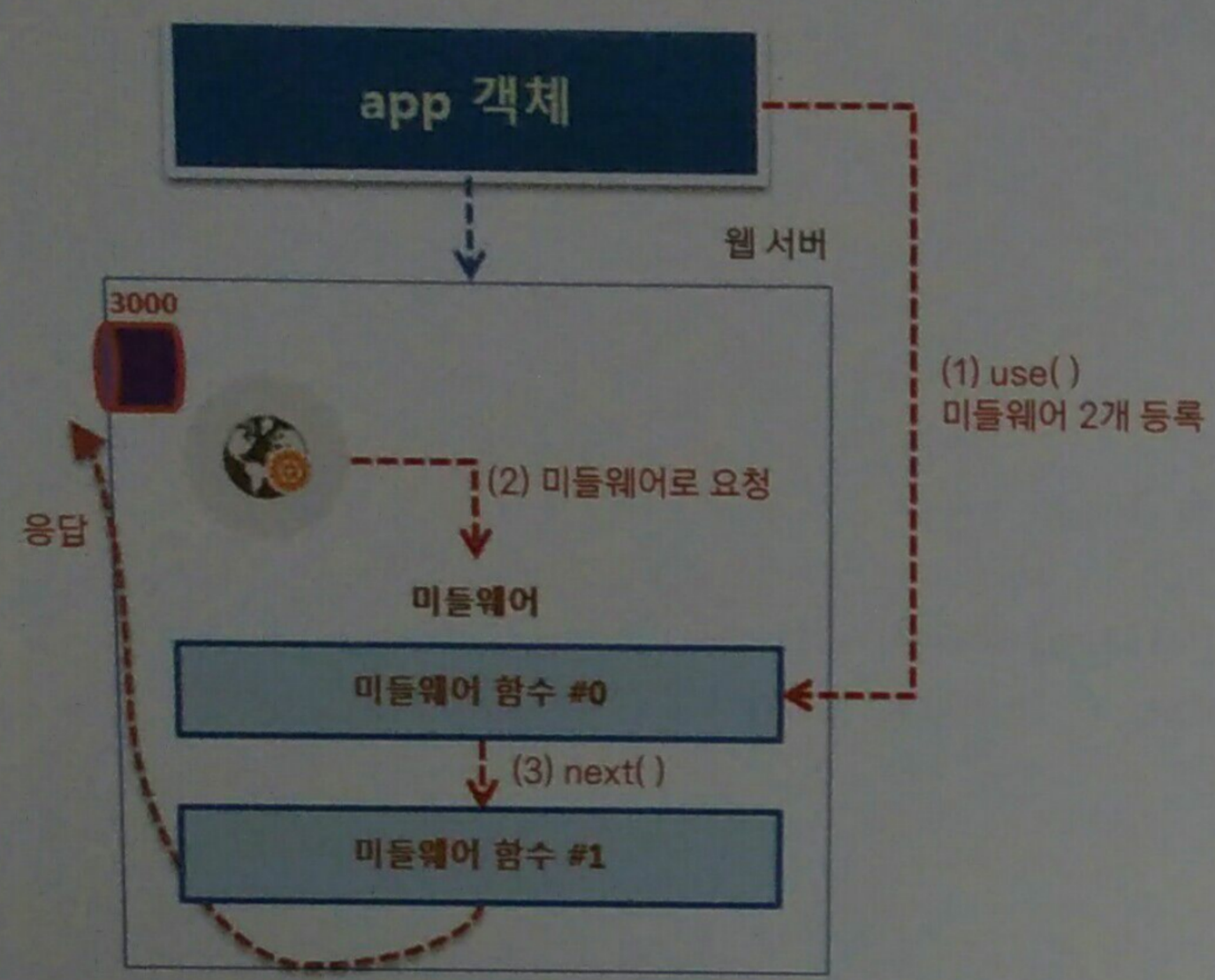
미들웨어 안에서는 기본적으로 요청 객체인 req와 응답 객체인 res 객체를 파라미터로 전달받아 사용할 수 있습니다. 그리고 이 미들웨어 함수를 호출한 app 객체도 참조할 수 있도록 req 객체의 속성으로 app 객체가 들어 있습니다. 즉, req.app 코드를 사용하면 app 객체를 참조하여 사용할 수 있습니다. app3.js 파일을 실행한 후 웹 브라우저에서 사이트에 접속하면 다음과 같이 사용자 정보가 포함된 결과가 표시됩니다.



사이트에 접속하여 두 개의 미들웨어에서 처리한 결과 ▶

미들웨어 함수를 보면 요청 객체와 응답 객체가 파라미터로 전달되며, 그다음 미들웨어로 넘길 수 있는 **next 함수 객체**도 전달됩니다. 따라서 next 함수 객체를 실행하면 다음 미들웨어로 넘길 수 있습니다.

두 개의 미들웨어를 사용할 때 구성을 보면 다음과 같습니다.



▲ 두 개의 미들웨어를 사용하는 경우의 구성

미들웨어를 사용하기 위해 먼저 use( ) 메소드를 호출해야 하며, 미들웨어 2개를 등록하였습니다. 클라이언트에서 요청이 들어오면 미들웨어 함수는 순서대로 실행되는데, 첫 번째 미들웨어에서 응답을 보내지 않고 next( ) 함수를 호출하면 다음 미들웨어로 넘어가도록 구성되어 있습니다. 그리고 두 번째 미들웨어에서 보낸 응답은 클라이언트에 전달됩니다.

익스프레스의 요청 객체와 응답 객체 알아보기

익스프레스에서 사용하는 요청 객체와 응답 객체는 http 모듈에서 사용하는 객체들과 같습니다. 하지만 몇 가지 메소드를 더 추가할 수 있습니다. 다음은 익스프레스에서 추가로 사용할 수 있는 주요 메소드들입니다.

메소드 이름	설명
send([body])	클라이언트에 응답 데이터를 보냅니다. 전달할 수 있는 데이터는 HTML 문자열, Buffer 객체, JSON 객체, JSON 배열입니다.
status(code)	HTTP 상태 코드를 반환합니다. 상태 코드는 end( )나 send( ) 같은 전송 메소드를 추가로 호출해야 전송할 수 있습니다.
sendStatus(statusCode)	HTTP 상태 코드를 반환합니다. 상태 코드는 상태 메시지와 함께 전송됩니다.

redirect([status,] path)	웹 페이지 경로를 강제로 이동시킵니다.
render(view [, locals][, callback])	뷰 엔진을 사용해 문서를 만든 후 전송합니다.

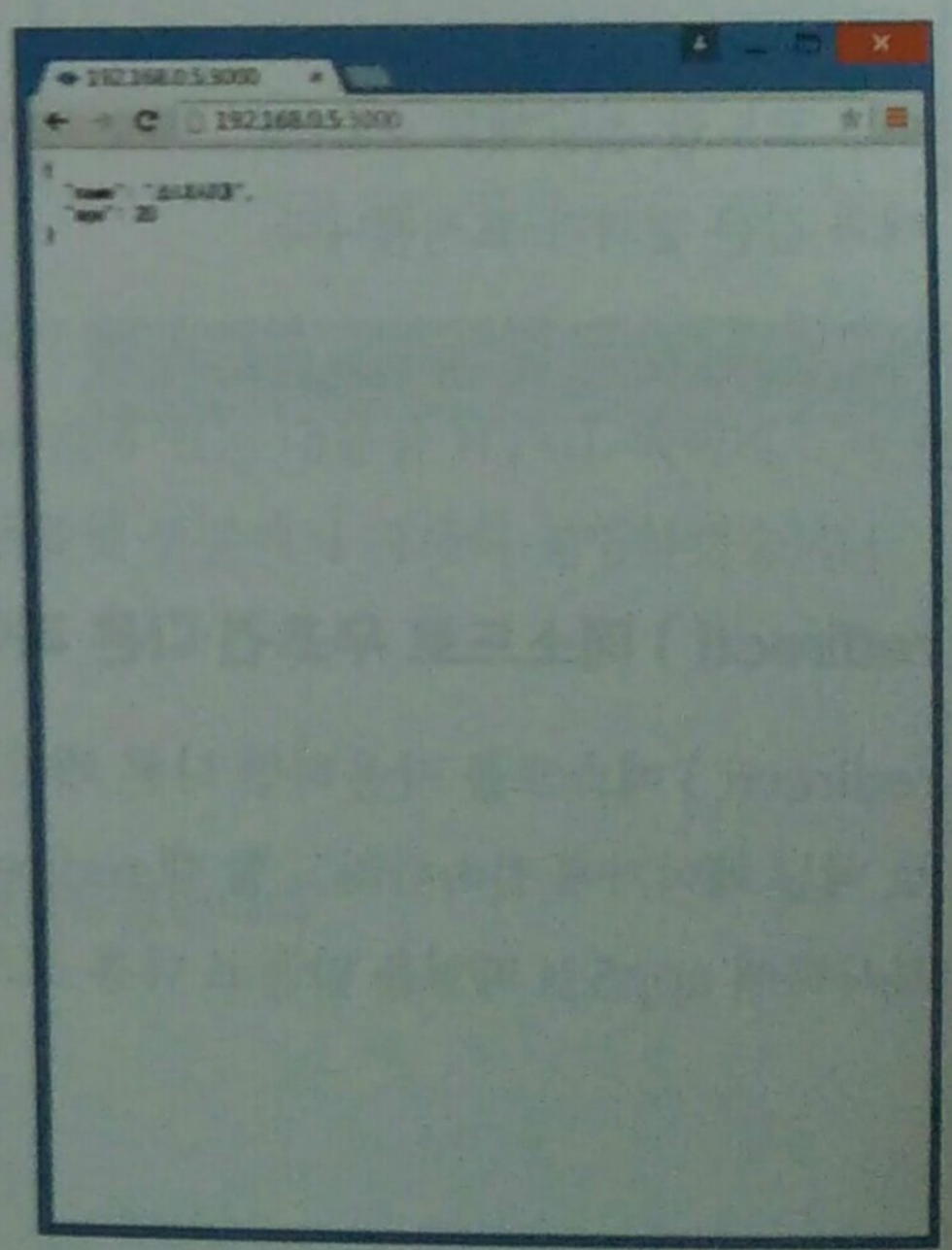
[표] 익스프레스에서 추가로 사용할 수 있는 응답 객체의 메소드

send( ) 메소드는 응답 데이터를 좀 더 간단하게 전송하기 위해 익스프레스에서 추가된 것입니다. send( ) 메소드로 클라이언트에 JSON 데이터를 전송하는 기능을 확인해 보기 위해 app3.js 파일을 복사하여 app4.js 파일을 만듭니다. 그런 후 다음 코드로 수정합니다.

참조 파일 | ExpressExample>app4.js

```
……

app.use(function(req, res, next) {
  console.log('첫 번째 미들웨어에서 요청을 처리함.');

  res.send({name:'소녀시대', age:20});
});

……
```

파일을 실행한 후 브라우저에서 사이트에 접속하면 다음과 같이 JSON 객체를 응답으로 받아 표시합니다.



사이트에 접속하여 확인한 두 개의 미들웨어 처리 결과 ▶

사실 JSON 객체를 그대로 받아 웹 페이지로 보여 주는 경우는 거의 없습니다. 하지만 모바일 단말기에서 사용하는 웹 앱이나 일반 앱에서는 화면을 보여 주기 위해 필요한 웹 문서들을 단말 쪽에 두고 Ajax

또는 웹 소켓으로 데이터만 수신하여 화면에 보여 줄 때가 많습니다. 따라서 JSON 객체를 전송받는 기능이 필요합니다.

정박사의 한마디 문서가 아니라 JSON 데이터만 전달받는 경우가 점점 많아지고 있어요

게시판을 예로 들어 설명해 보겠습니다. 글 목록에서 항목 하나를 눌러 글의 내용을 조회할 때마다 웹 서버에서 웹 문서를 새로 받아 오는 방식은 매번 웹 브라우저 화면을 업데이트하기 때문에 하얀 화면이 보이는 경우가 많을 뿐 아니라 받아 오는 데이터 양도 많아집니다. 이런 이유 때문에 JSON 데이터만 받아 와서 게시물을 보여 줄 때 해당 데이터만 업데이트하는 것이 효율적입니다. 특히 모바일 단말에서는 화면을 위해 만든 웹 문서를 웹 서버가 아니라 단말에 저장해 두고 보여 줄 때가 많기 때문에 웹 서버에서는 데이터만 받아 옵니다.

status( )와 sendStatus( ) 메소드를 사용하여 상태 코드를 전송할 수 있습니다. status( ) 메소드는 상태 코드를 작성하는 기능만 있으므로 상태 코드를 전송하려면 send( ) 메소드를 추가로 호출해야 합니다. 예를 들어, 다음 코드를 사용할 수 있습니다.

```
res.status(403).send('Forbidden');
```

만약 sendStatus( ) 메소드를 사용한다면 다음과 같이 코드만 파라미터로 전달합니다.

```
res.sendStatus(403);
```

403 코드만 전달해도 status( ) 메소드와 send( ) 메소드를 연속으로 호출해서 다음 코드를 호출했을 때와 같은 결과가 표시됩니다.

```
res.status(403).send('Forbidden');
```

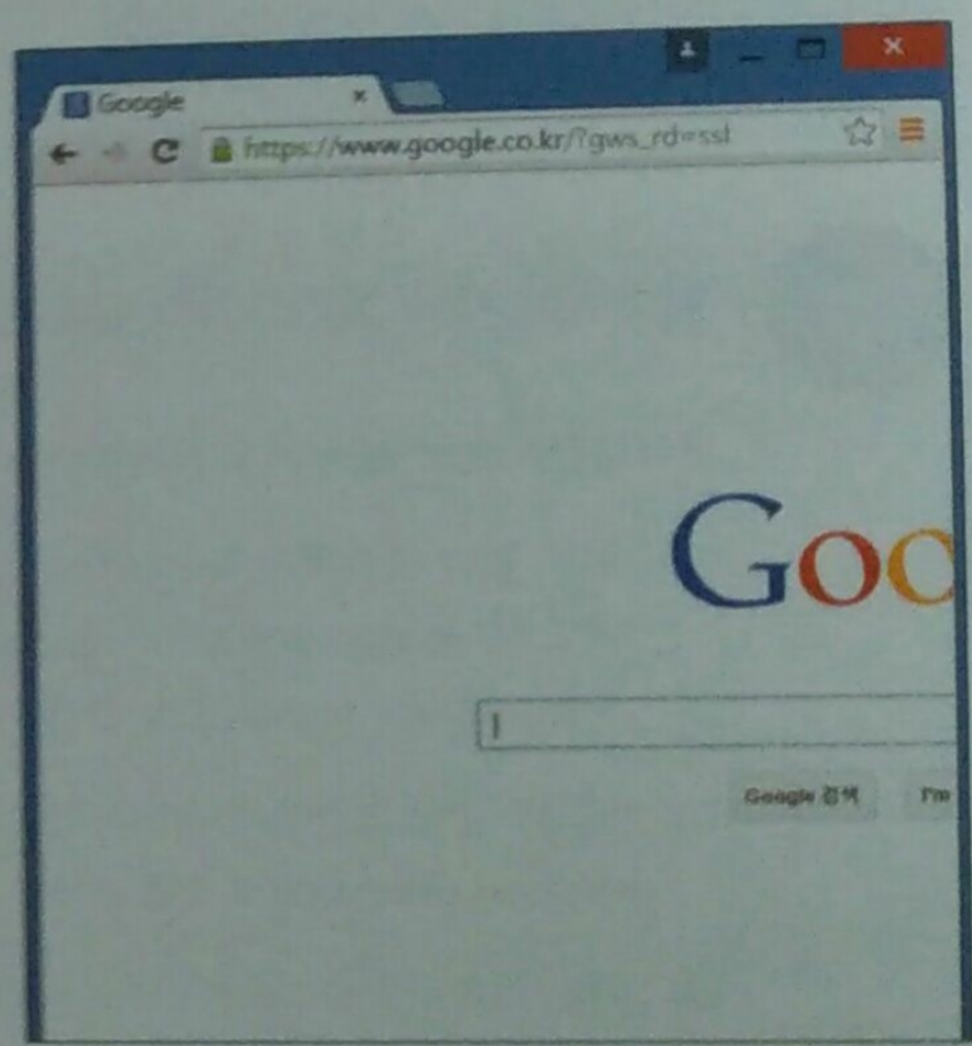
redirect( ) 메소드로 무조건 다른 페이지로 이동하기

redirect( ) 메소드를 사용하면 다른 페이지로 이동할 수 있습니다. 예를 들어, 로그인을 하지 않은 상태로 메뉴 페이지에 접속하려고 할 때 redirect( ) 메소드로 페이지를 옮겨 갈 수 있습니다. app4.js 파일을 복사하여 app5.js 파일을 만들고 다음 코드를 입력하여 무조건 다른 페이지로 이동하게 만들어 봅니다.

참조 파일 | ExpressExample>app5.js

```
……

app.use(function(req, res, next) {
  console.log('첫 번째 미들웨어에서 요청을 처리함.');

  res.redirect('http://google.co.kr');
});

……
```

위의 코드는 redirect( ) 메소드를 호출하면서 구글 사이트의 주소를 파라미터로 전달하도록 만들었습니다. app5.js 파일을 실행하고 웹 브라우저로 접속하면 구글 사이트가 결과 페이지로 나타납니다.



사이트에 접속한 후 redirect( ) 메소드로 띄운 구글 사이트 ▶

redirect( ) 메소드를 사용하면 URL 주소뿐만 아니라 프로젝트 폴더 안에 있는 다른 페이지를 보여 줄 수 있습니다. render( ) 메소드는 뷰 엔진을 사용해 템플릿 파일에 있는 내용을 HTML 페이지로 바꾼 후 그 결과물을 전송합니다. 이 내용에 대해서는 나중에 뷰 템플릿 부분에서 자세히 설명하겠습니다.

익스프레스에서 요청 객체에 추가한 메소드 알아보기

이번에는 익스프레스에서 요청 객체에 추가한 메소드들을 살펴보겠습니다.

메소드 이름	설명
param(name)	요청 파라미터를 확인합니다.
header(name)	헤더를 확인합니다.

[표] 익스프레스에서 추가로 사용할 수 있는 요청 객체의 메소드

클라이언트의 요청으로 보내진 파라미터들은 요청 객체의 param( ) 메소드로 확인할 수 있습니다. 예를 들어, http://localhost:3000/?name=mike와 같은 주소로 접속한 경우에 요청 파라미터인 name의 값은 다음 코드로 확인할 수 있습니다.

```
var paramName = req.param('name');
```

요청 파라미터는 영어로 query string이라고 합니다. 즉, 클라이언트에서 서버로 정보를 요청할 때 문자열로 데이터를 전달할 수 있는 방법입니다. 이 요청 파라미터는 서버 쪽에서 받아 사용할 수 있어야 하므로 위와 같이 req 객체의 param 메소드를 사용하는데, 이 요청 파라미터의 값은 req.query 객체의 속성으로 들어 있어 다음과 같은 코드를 사용할 수도 있습니다.

```
var paramName = req.query.name;
```

정박사의 한마디 파라미터 사용법을 정리해 볼게요

클라이언트에서 웹 서버로 파라미터를 보내는 방법은 세 가지로 구별할 수 있습니다.

(1) GET 방식 요청에서 파라미터로 보내기
http://localhost:3000/?name=mike
⇢ req.query.name 으로 참조 가능

(2) POST 방식 요청에서 파라미터로 보내기
req.body.name으로 참조 가능

(3) URL 파라미터로 보내기(URL 파라미터 설명 부분을 참조)
⇢ req.params.name 으로 참조 가능

req.param( ) 메소드는 (1)번과 (2)번 모두 사용할 수 있습니다. 이 책에서는 req.param( ) 메소드를 사용합니다. 단 최신 버전이라면 (1)번과 (2)번을 각각 사용하길 권합니다.

요청할 때 헤더 값들은 header( ) 메소드로 확인할 수 있습니다. app5.js 파일을 복사하여 app6.js 파일을 만든 후 클라이언트가 요청했을 때 User-Agent 헤더와 요청 파라미터 중 name 값을 확인하는 코드를 다음과 같이 입력합니다.

참조 파일 | ExpressExample>app6.js

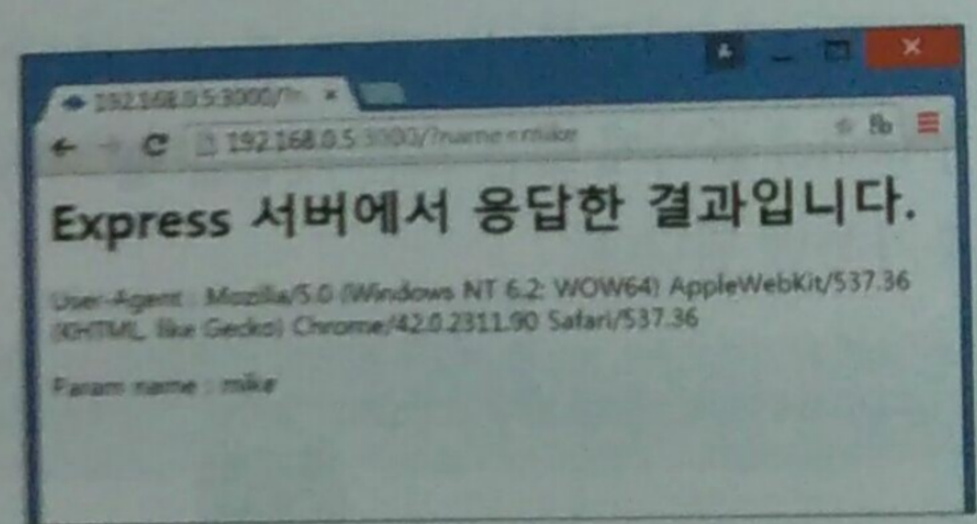
```
.....

app.use(function(req, res, next) {
  console.log('첫 번째 미들웨어에서 요청을 처리함.');

  var userAgent = req.header('User-Agent');
  var paramName = req.param('name');
```

```
  res.writeHead('200', {'Content-Type':'text/html;charset=utf8'});
  res.write('<h1>Express 서버에서 응답한 결과입니다.</h1>');
  res.write('<div><p>User-Agent : ' + userAgent + '</p></div>');
  res.write('<div><p>Param name : ' + paramName + '</p></div>');
  res.end();
});

......
```

이 파일을 실행하고 웹 브라우저를 연 후 주소 창에 다음 주소를 입력합니다.

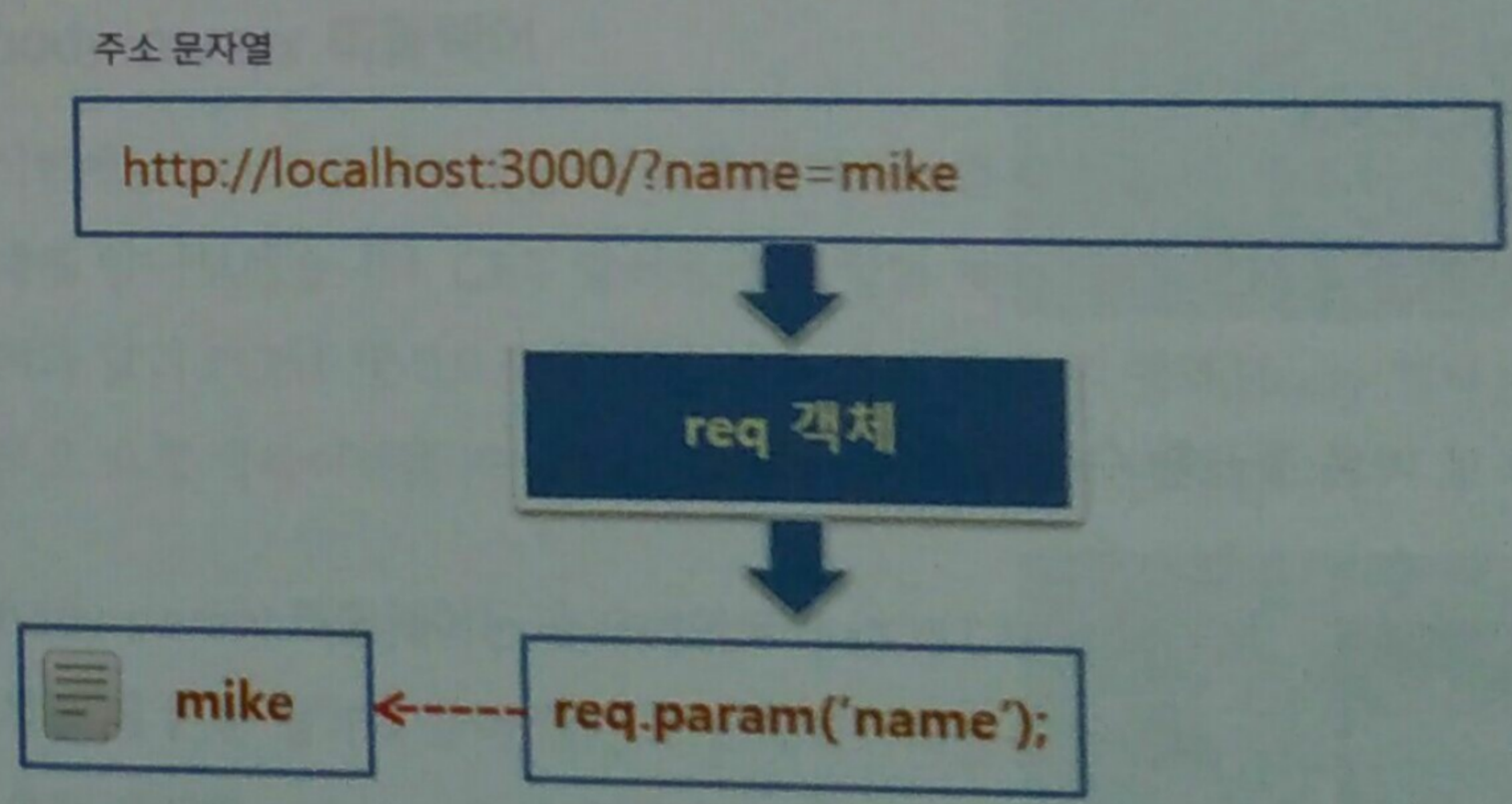
▶ http://localhost:3000/?name=mike

사이트에 접속하면 다음 결과를 볼 수 있습니다.



▲ 사이트에 접속하여 헤더와 요청 파라미터 값을 확인한 경우

다음 그림은 익스프레스에서 클라이언트의 요청 파라미터를 확인하는 과정을 간단하게 정리한 것입니다.



▲ 익스프레스에서 param( ) 메소드로 확인하는 요청 파라미터

클라이언트가 주소 문자열에 포함시켜 전달하는 요청 파라미터를 웹 서버에서 받아 확인할 때는 보통 여러 줄의 코드가 필요합니다. 하지만 익스프레스를 사용하면 익스프레스 내부에서 요청 파라미터를 어떻게 관리하는지 잘 모르더라도 한 줄의 코드만으로 요청 파라미터의 값을 확인할 수 있습니다.

05-3 | 미들웨어 사용하기

지금까지 use( ) 메소드로 설정하는 미들웨어 함수 안에 코드를 직접 넣어 클라이언트로 응답을 전송해 보았습니다. 그런데 웹 서버를 다루기 위해 필요한 모든 기능을 여러분이 직접 미들웨어로 만들어야 한다면 실무에서 사용할 수 있는 웹 서버를 구성하는 것이 쉽지 않겠죠? 익스프레스에서는 개발자가 다양한 기능을 사용할 수 있도록 미리 만들어 둔 여러 가지 미들웨어를 제공합니다. 특히 다른 개발자들이 만들어서 올려 둔 미들웨어를 쉽게 끼워 넣을 수 있어 아주 편리하게 사용할 수 있습니다. 여기에서는 자주 사용하는 몇 가지 기본 미들웨어에 대해 알아보겠습니다.

static 미들웨어

먼저 static 미들웨어는 특정 폴더의 파일들을 특정 패스로 접근할 수 있도록 만들어 줍니다. 예를 들어, [public] 폴더에 있는 모든 파일을 웹 서버의 루트 패스로 접근할 수 있도록 만들고 싶다면 다음 코드를 추가하면 됩니다.

```
app.use(express.static(path.join(__dirname, 'public' )));
```

이 코드는 익스프레스 프로젝트가 만들어질 때 app.js 파일에 자동으로 추가됩니다. 따라서 [public] 폴더 안에 있는 파일들은 클라이언트에서 바로 접근할 수 있습니다. 예를 들어, 프로젝트 폴더 안에 다음과 같은 폴더나 파일들이 들어 있다면,

```
ExpressExample/public/index.html
ExpressExample/public/images/house.png
ExpressExample/public/js/main.js
ExpressExample/public/css/style.css
```

웹 브라우저에서 다음과 같은 주소로 바로 접근할 수 있습니다.

```
http://localhost:3000/index.html
http://localhost:3000/images/house.png
http://localhost:3000/js/main.js
http://localhost:3000/css/style.css
```

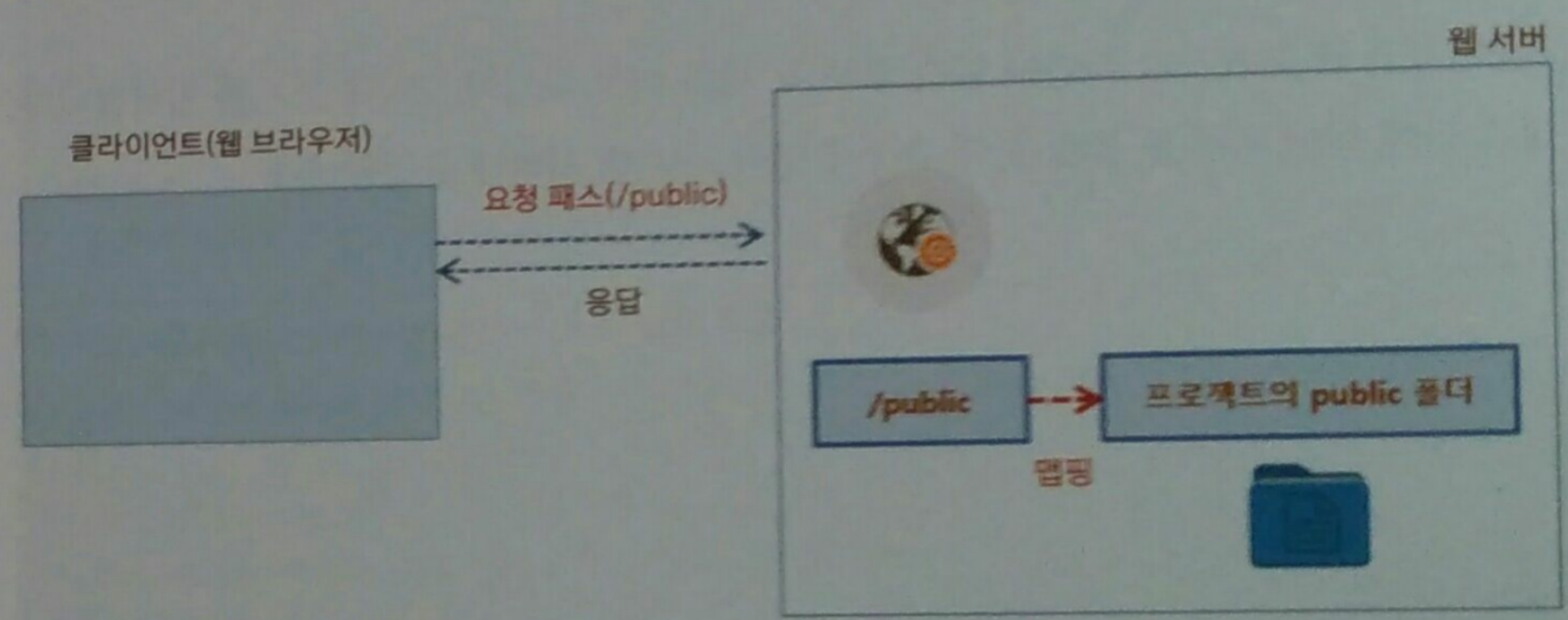
[public] 폴더를 바로 접근할 수 있게 되었습니다. ExpressExample 프로젝트 안의 [/public/images] 폴더에 들어 있는 house.png 이미지 파일을 웹 브라우저에서 보려면 app.js 파일 안에서 다음과 같이 응답을 보내면 됩니다.

```
res.end("<img src=' /images/house.png' width=' 50%' ");
```

만약 [public] 폴더 안에 있는 파일을 사이트의 /public 패스로 접근하도록 만들고 싶다면 static( ) 메소드를 호출할 때 다음과 같이 패스를 지정해 주면 됩니다.

```
app.use( '/public' , express.static(path.join(__dirname, 'public')));
```

코드가 조금 길어 보이지만 자세히 살펴보면, use( ) 메소드의 첫 번째 파라미터로 요청 패스를 지정했으며 두 번째 파라미터 static( ) 함수로 특정 폴더를 지정했습니다. 이렇게 하면 요청 패스와 특정 폴더가 맵핑(Mapping, 서로 연결되는 것)되어 접근할 수 있습니다.

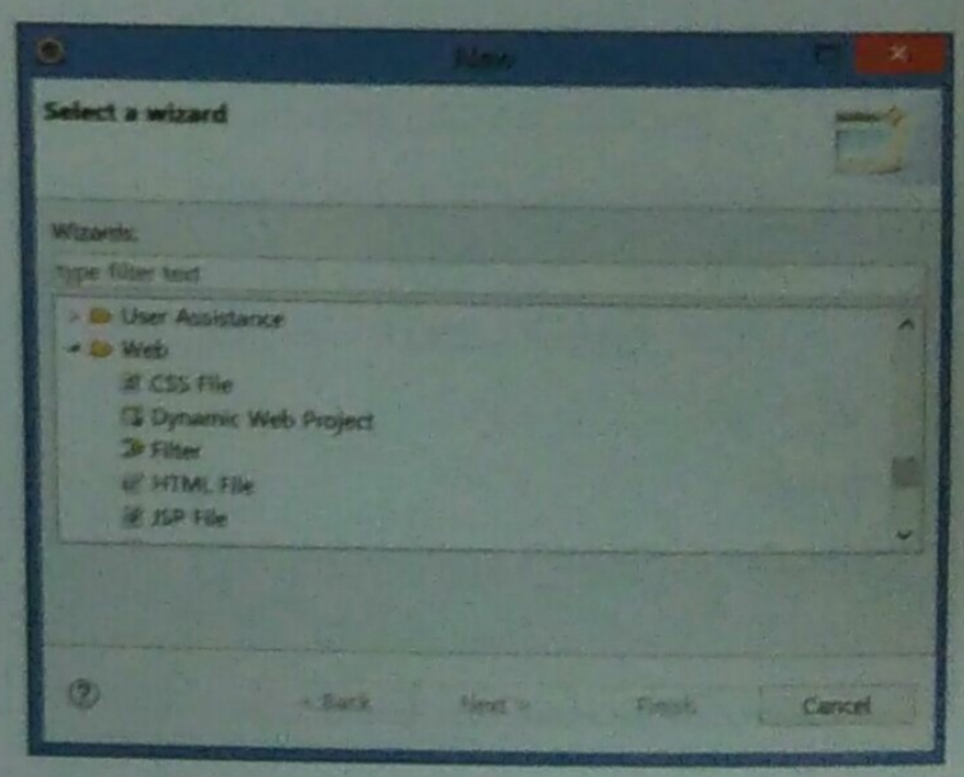


▲ 요청 패스와 특정 폴더를 static( ) 메소드로 맵핑하기

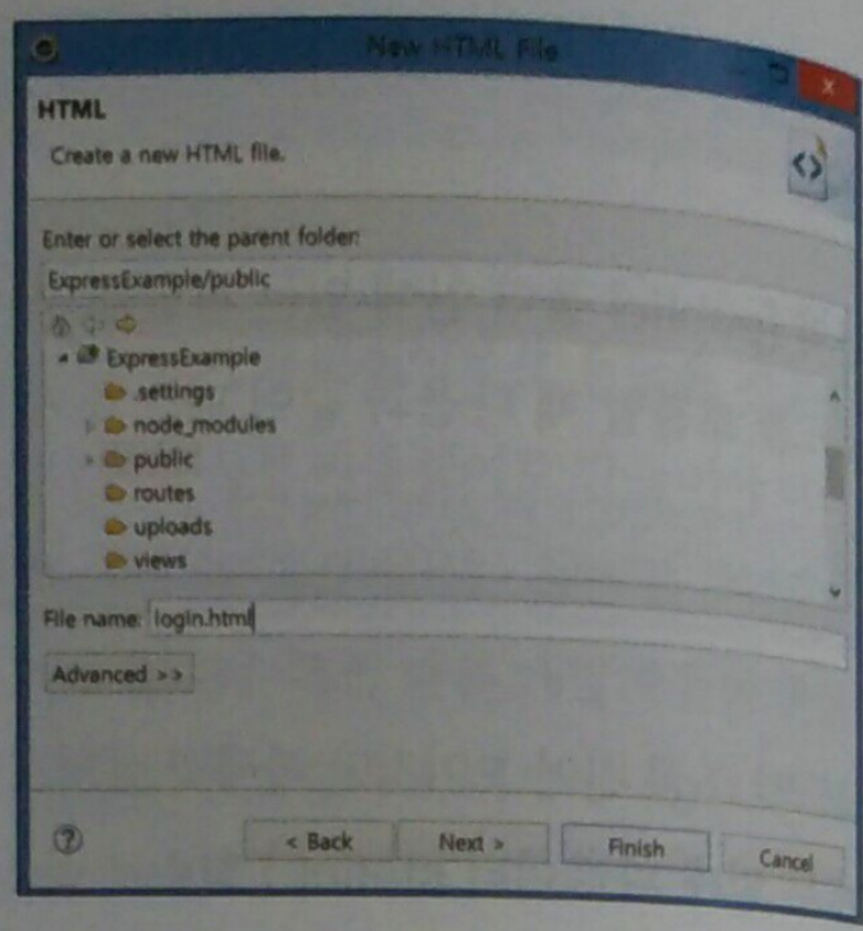
body-parser 미들웨어

이번에는 POST로 요청했을 때 요청 파라미터를 확인할 수 있도록 만들어 둔 body-parser 미들웨어에 대해 알아보겠습니다. GET 방식으로 요청할 때는 주소 문자열에 요청 파라미터가 들어갑니다. 하지만 이와 달리 POST 방식으로 요청할 때는 본문인 본문 영역(Body 영역)에 요청 파라미터가 들어 있게 되므로 요청 파라미터를 파싱하는 방법이 GET 방식과 달라집니다.

body-parser 미들웨어는 클라이언트가 POST 방식으로 요청할 때 본문 영역에 들어 있는 요청 파라미터들을 파싱하여 요청 객체의 params 속성에 넣어 줍니다. POST 방식으로 요청하여 어떻게 요청 파라미터를 받는지 확인해 보겠습니다. 먼저 웹 브라우저에 접속하여 볼 수 있는 간단한 웹 문서를 하나 만듭니다. 이클립스를 실행한 다음 [File → New → Other] 메뉴를 누른 후 대화상자가 보이면 [Web] 폴더 안에 있는 HTML File 항목을 선택하고 [Next] 버튼을 누릅니다.



HTML 파일 정보를 입력하는 대화상자가 보이면 [Express Example] 폴더 안에 있는 [public] 폴더를 선택한 후 파일 이름으로 login.html을 입력하고 [Finish] 버튼을 누릅니다.



▲HTML 파일 정보를 입력하는 대화상자

새로운 파일이 만들어지면 웹 페이지의 화면에 입력 상자 두 개와 버튼 하나가 보이게 코드를 입력합니다. 그리고 〈form〉 태그 안에 넣어 버튼을 눌렀을 때 POST 방식으로 웹 서버에 요청합니다.

참조 파일 | ExpressExample>/public/login.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>로그인 테스트</title>
  </head>
<body>
  <h1>로그인</h1>
  <br>
  <form method="post">
    <table>
      <tr>
        <td><label>아이디</label></td>
        <td><input type="text" name="id"></td>
      </tr>
      <tr>
        <td><label>비밀번호</label></td>
        <td><input type="password" name="password"></td>
      </tr>
    </table>
    <input type="submit" value="전송" name="">
  </form>
</body>
</html>
```

이제 app6.js 파일을 복사하여 app7.js 파일을 만들고, POST 방식으로 정보를 요청했을 때 요청 파라미터를 확인할 수 있도록 다음과 같이 입력합니다.

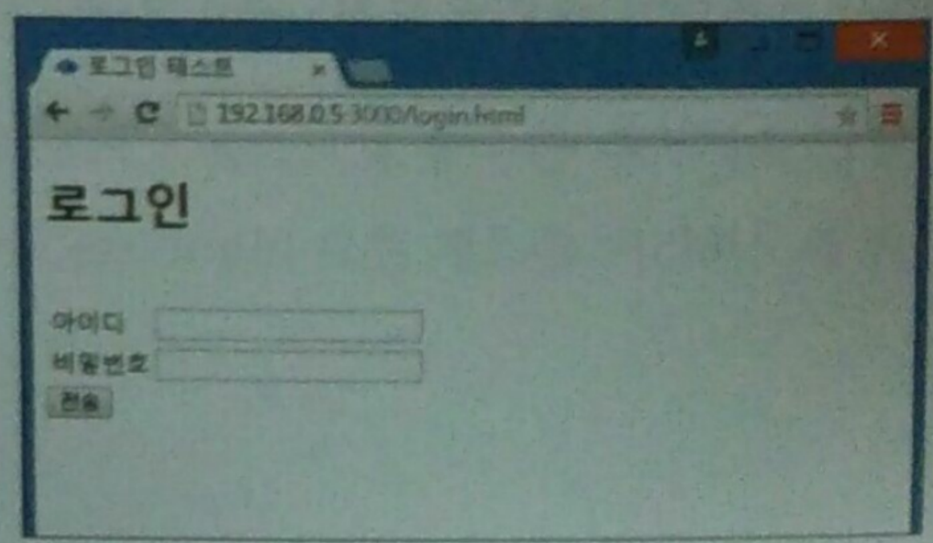
참조 파일 | ExpressExample>app7.js

```
.....

var bodyParser = require('body-parser');

var app = express();

.....

app.use(bodyParser.urlencoded({ extended: true }));

app.use(function(req, res, next) {
  console.log('첫 번째 미들웨어에서 요청을 처리함.');

  var paramId = req.param('id');
  var paramPassword = req.param('password');

  res.writeHead('200', {'Content-Type' : 'text/html;charset=utf8'});
  res.write('<h1>Express 서버에서 응답한 결과입니다.</h1>');
  res.write('<div><p>Param id : ' + paramId + '</p></div>');
  res.write('<div><p>Param password : ' + paramPassword + '</p></div>');
  res.end();
});

.....
```

body-parser 미들웨어는 외장 모듈로 만들어져 있으므로 먼저 설치해야 합니다. 명령 프롬프트에 다음 코드를 입력하여 body-parser 모듈을 설치합니다.

```
% npm install body-parser --save
```

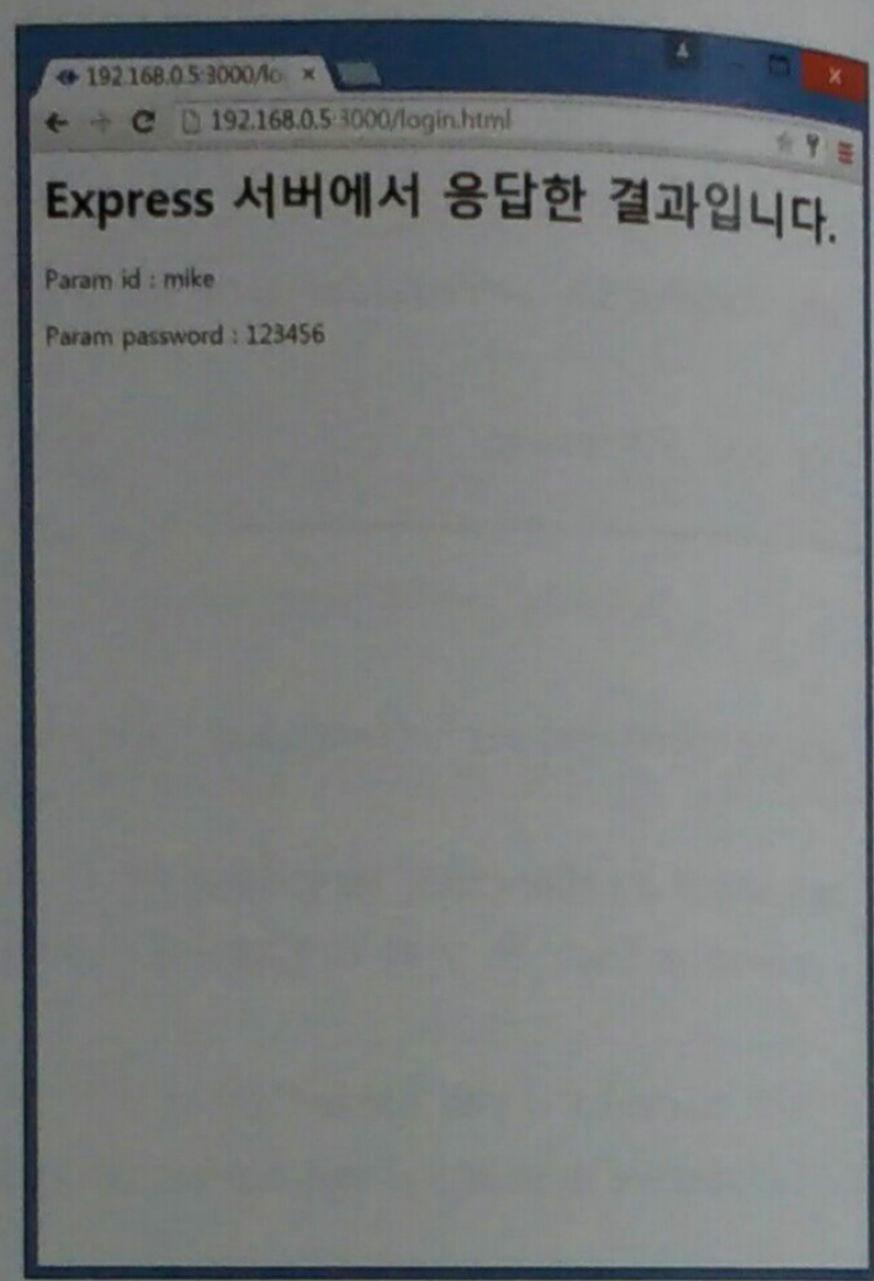
이제 웹 서버가 클라이언트 요청을 기다리도록 app7.js 파일을 실행합니다. 그런 다음 웹 브라우저에 http://localhost:3000/login.html 주소를 입력해서 접속하면 다음 화면이 표시됩니다.



웹 브라우저에 띄운 로그인 문서 ▶

서버 코드에서 use( ) 메소드로 설정한 함수는 login.html 문서에 접근할 때는 호출되지 않습니다. 따라서 처음에는 login.html 문서가 웹 브라우저에 보이며, 버튼을 클릭하여 POST로 요청했을 때는 use( ) 메소드로 설정한 함수가 호출되면서 필요한 작업을 수행합니다.

아이디와 비밀번호를 입력한 후 [전송] 버튼을 누르면 다음 화면이 표시됩니다.



[전송] 버튼을 눌러 결과 화면을 본 경우▶

이제 GET 방식으로 요청했을 때와 마찬가지로 POST 방식으로 요청했을 때도 param( ) 메소드로 요청 파라미터를 확인할 수 있게 되었습니다. 이것이 어떻게 가능한지 이해했다면 노드에서 미들웨어를 사용한다는 것이 얼마나 편리한지 느낄 수 있을 것입니다.

05-4 | 요청 라우팅하기

앞에서 로그인 페이지를 열고 버튼을 눌렀을 때 로그인 처리를 하는 간단한 예를 살펴보았습니다. 그런데 로그인이 아니라 사용자 리스트 등 다른 요청이 들어왔을 때도 use( ) 메소드로 설정한 미들웨어 함수가 항상 호출되기 때문에 요청 url이 무엇인지 일일이 확인해야 하는 번거로움이 생깁니다.