

どうぞ、よろしくおねがいたします。

田中朋也
tiny-studio.com

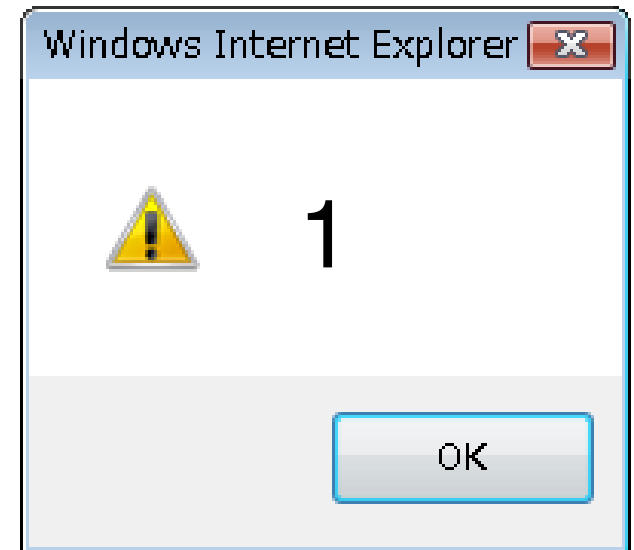
えっと、突然ですが。

このプログラムを実行すると

```
alert(1);
```

こうなります。

```
alert(1);
```



なので、プログラムをこう変えると

```
alert(5);
```

もちろん、こうなります。

```
alert(5);
```



計算式にかえると

```
alert(5 + 4);
```

計算結果が表示されます。

```
alert(5 + 4);
```



で、このプログラムを実行すると

```
var a = 5;  
alert(a);
```

こうなるわけです。

```
var a = 5;  
alert(a);
```



いわゆる、変数です。

```
var a = 5;  
alert(a);
```



変数名を変えても

```
var foo = 5;  
alert(foo);
```

ただしく動作します。

```
var foo = 5;  
alert(foo);
```

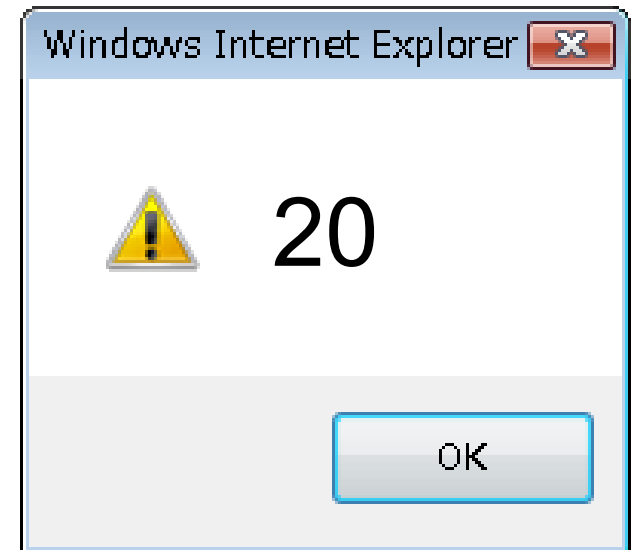


変数を使った計算も

```
var foo = 5;  
var baz = foo * 4;  
alert(baz);
```

ご覧の通りです。

```
var foo = 5;  
var baz = foo * 4;  
alert(baz);
```



えーと、

「いやいや、説明せえよ」
と、いう方も多いんじゃないかと
思います

実は、このスライドは
経験者向けに制作したものです。

全くの初心者向けのスライドも
用意してありますので、

今は退屈かとは思いますが、
ぜひ最後までご覧ください。

えーと、逆に、

「それくらいなら、わかるよ。」
と、いう方も多いんじゃないかと
思います

そういう方にとっては、
今回のお話では、
新しい知識は、
一つも手に入りません。

このスライドは、
すでに知っている知識を再確認して、
プログラムの処理を「イメージ」できる事を
目的にお話してみたいと思います。

退屈かとは思いますが、
ぜひ最後までご覧ください。

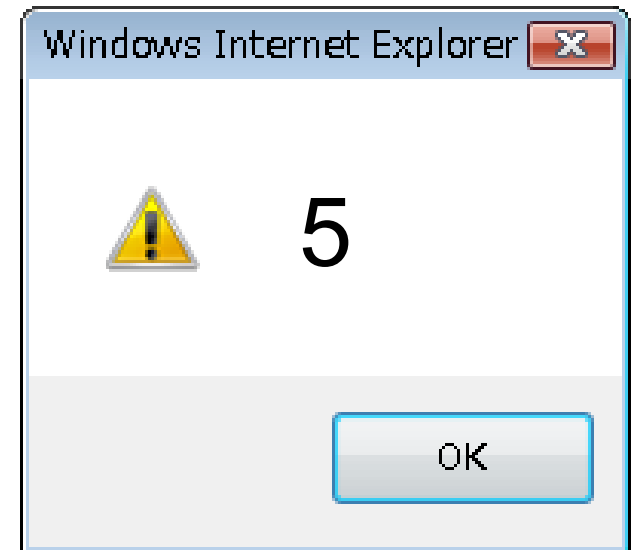
経験者の方はご存知の通り、
プログラムは基本的には、
上の行から順番に実行されます。

たとえば、こう書くと

```
alert(5);  
alert(8);  
alert(4);
```

まず5が表示されて

```
alert(5);  
alert(8);  
alert(4);
```



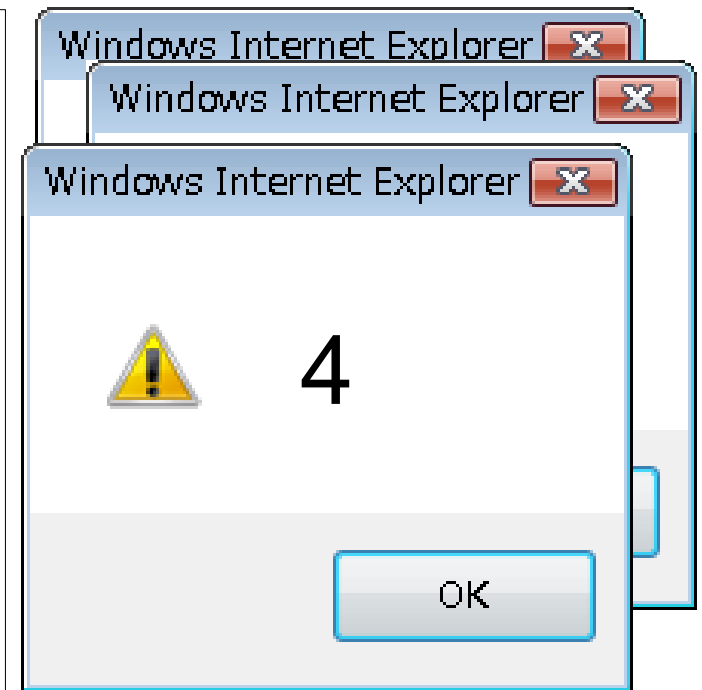
つぎに8が表示されて

```
alert(5);  
alert(8);  
alert(4);
```



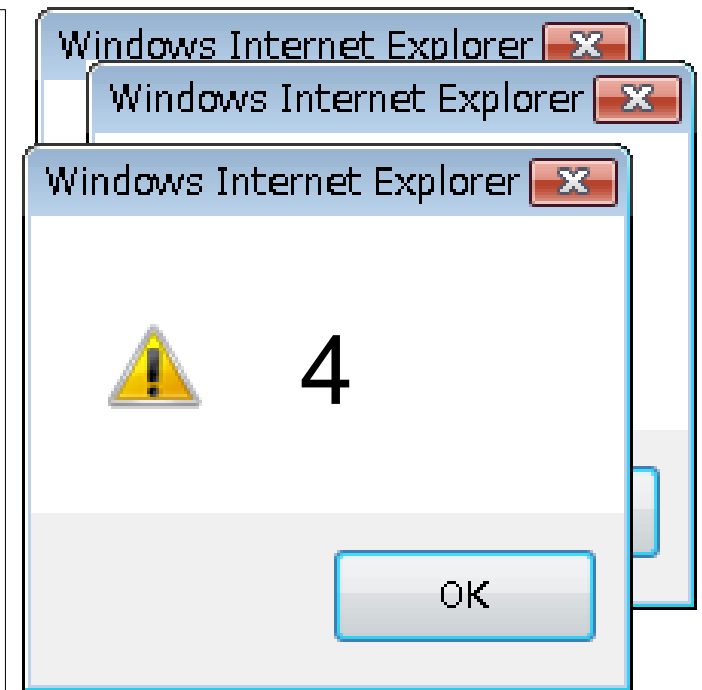
その次に4が表示されます。

```
alert(5);  
alert(8);  
alert(4);
```



この順序が狂うことはありません。

```
alert(5);  
alert(8);  
alert(4);
```



ただ、プログラムには縦方向の順序だけでなく、横方向にも順序を持っているのです。

プログラムの教科書には

「演算子の実行順序」

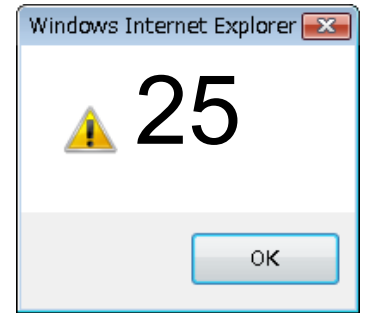
という形でよく紹介されています。

ここに一行のプログラムがあります。

```
alert (2 + 3 + 5 * 4);
```

実行すると、こうなります。

```
alert (2 + 3 + 5 * 4);
```

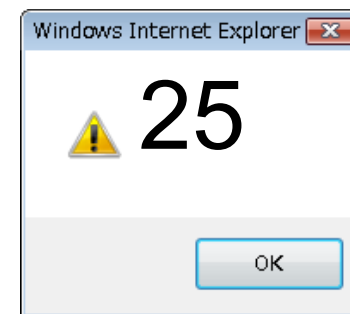


このプログラムは1行ですが、
じつは処理は1つだけではないんです。

1行の命令を与えられたブラウザは、
まず演算子を全て見つけ出します。

```
alert (2 + 3 + 5 * 4);
```

Diagram illustrating operator precedence in the JavaScript expression `2 + 3 + 5 * 4`. Red arrows point down to the operators `+`, `+`, and `*` in the expression, indicating that the parser first identifies all operators before evaluating the expression.



この例では、3つありますね。

```
alert (2 + 3 + 5 * 4);
```

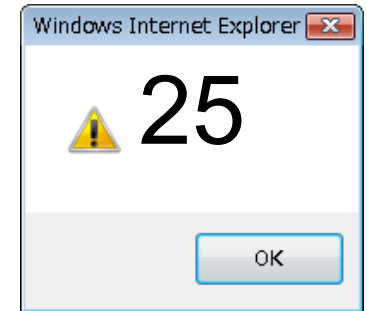
Three red arrows point downwards to the operators '+', '+', and '*' in the code snippet above.



次に、ブラウザはそれぞれの演算子ごとに実行順序を決めます。

alert (2 + 3 + 5 * 4);

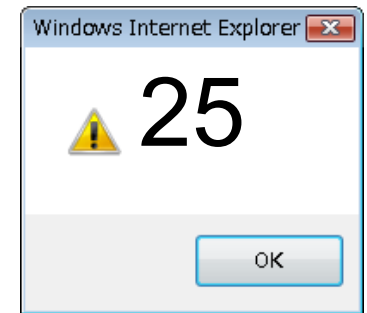
② ③ ①
↓ ↓ ↓



優先順位がかぶっている場合は、
「結合性」という決まりに従います。

alert (2 + 3 + 5 * 4) ;

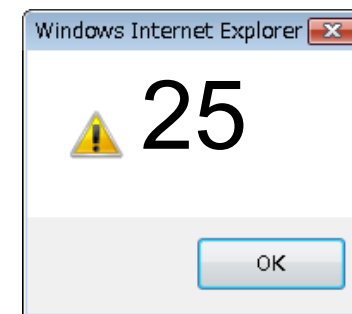
② ③ ①
↓ ↓ ↓



足し算の場合は左のが優先されます。
計算に使うのはたいてい左優先です。

alert (2 + 3 + 5 * 4);

② ③ ①
↓ ↓ ↓



順序が決まったら、一つずつ実行して
いくわけです。

alert (2 + 3 + 5 * 4) ;

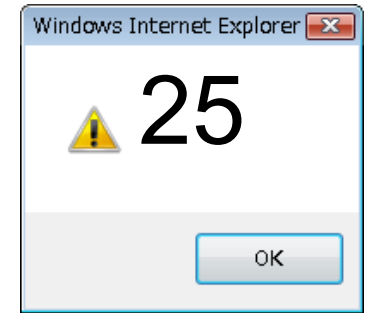
② ③ ①
↓ ↓ ↓



こんな感じのイメージ

alert (2 + 3 + 5 * 4) ;

② ↓ ③ ↓ ① ↓

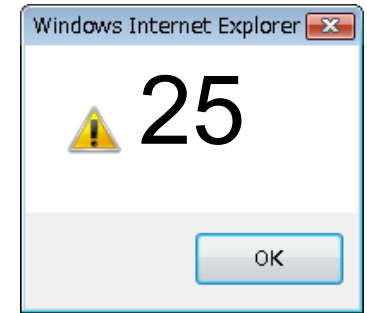


こんな感じのイメージ

alert (2 + 3 + 5 * 4) ;

② ↓ ③ ↓ ① ↓

(2 + 3 + 5 * 4) ;



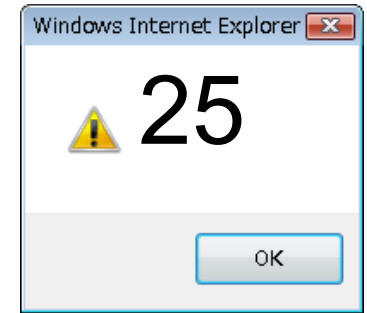
こんな感じのイメージ

alert (2 + 3 + 5 * 4) ;

② ↓
③ ↓
① ↓

(2 + 3 + 5 * 4)

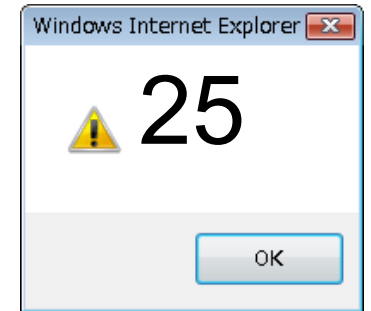
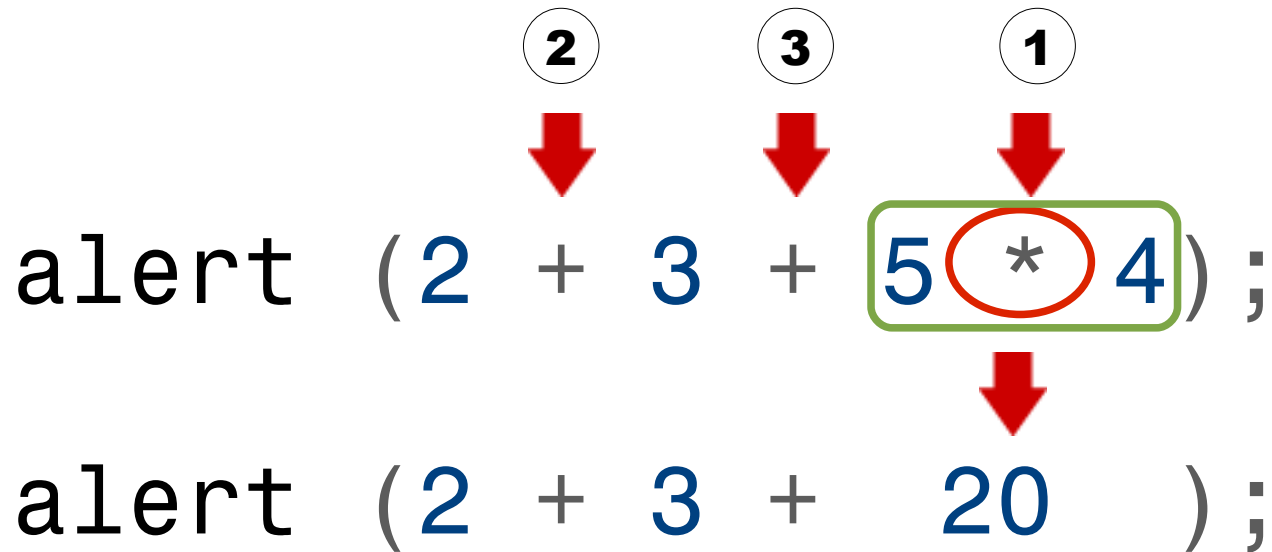
The diagram illustrates the order of operations in the JavaScript code snippet `alert (2 + 3 + 5 * 4) ;`. Three circled numbers (2, 3, and 1) are positioned above the code, with red arrows pointing down to the corresponding operators: the first '+' (between 2 and 3), the second '+' (between 3 and 5), and the '*' (between 5 and 4). A green rounded rectangle encloses the entire expression `5 * 4`, and a red circle highlights the '*' operator within this expression.



こんな感じのイメージ

$\text{alert } (2 + 3 + 5 * 4);$

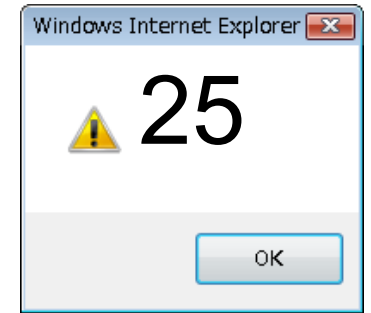
$\text{alert } (2 + 3 + 20);$



こんな感じのイメージ

alert (2 + 3 + 5 * 4) ;

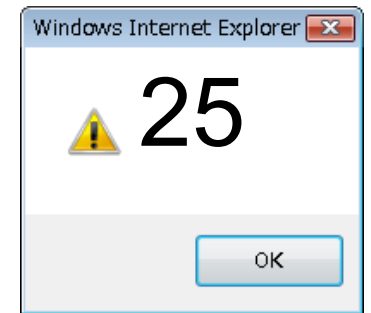
alert (2 + 3 + 20) ;



こんな感じのイメージ

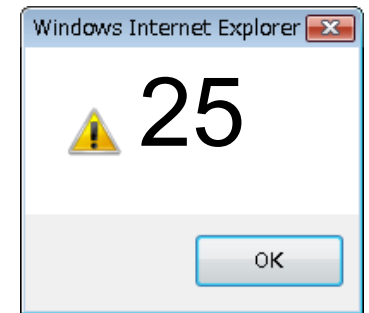
alert (2 + 3 + 5 * 4) ;

alert (2 + 3 + 20) ;



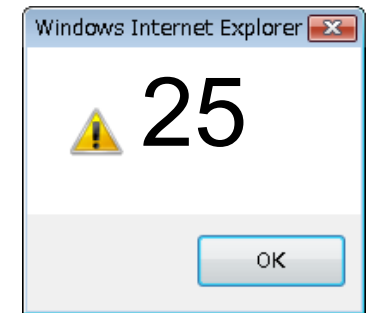
こんな感じのイメージ

② ③ ①
↓ ↓ ↓
alert (2 + 3 + 5 * 4) ;
↓
alert (2 + 3 + 20) ;
↓
alert (5 + 20) ;



こんな感じのイメージ

② ③ ①
↓ ↓ ↓
alert (2 + 3 + 5 * 4) ;
↓
alert (2 + 3 + 20) ;
↓
alert (5 + 20) ;
↓
alert (25) ;



プログラムの実行のイメージが、
少しつかめたでしょうか？

演算子の処理の直前に、
コンピュータが他にやっていることもあります。

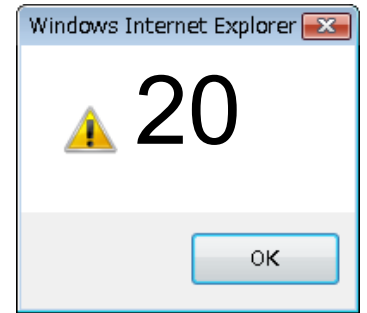
変数や、関数についての処理です。

もうひとつ、プログラムを用意しました。

```
var foo = 5;  
var baz = foo * 4;  
alert(baz);
```

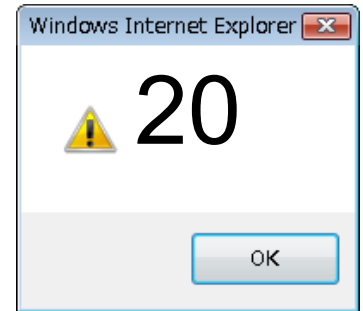
実行結果はこんな感じですよ。

```
var foo = 5;  
var baz = foo * 4;  
alert(baz);
```



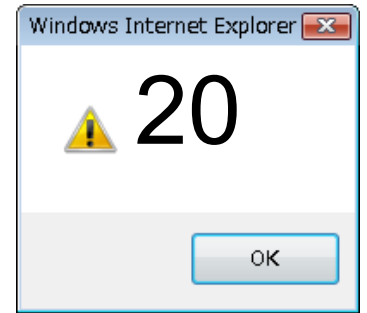
この行に注目してください。

```
var foo = 5;  
var baz = foo * 4;  
alert(baz);
```



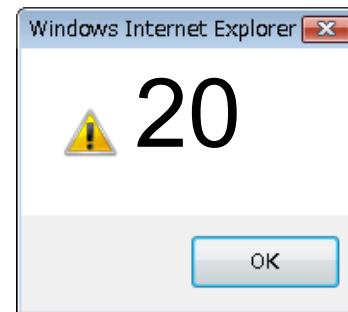
これが、演算子です・・・が、

```
var foo = 5;
var baz = foo * 4;
alert(baz);
```




コンピュータは演算子を処理するより
先に、この変数を処理します。

```
var foo = 5  
var baz = foo * 4;  
alert(baz);
```



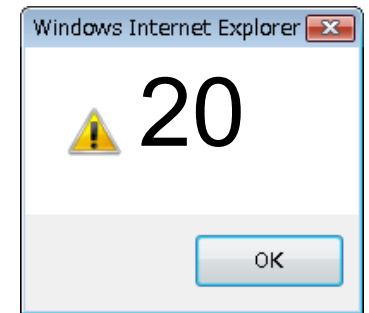
この変数には5が代入されていました。

```
var foo = 5;  
var baz = foo * 4;  
alert(baz);
```



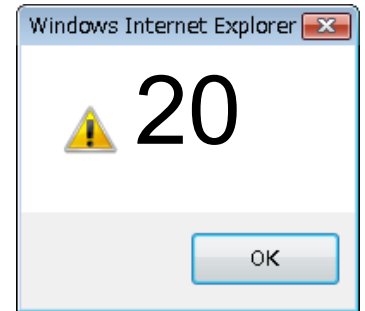
なので、ブラウザは今後、
この変数を5として扱います。

```
var foo = 5  
var baz = foo * 4;  
alert(baz);
```



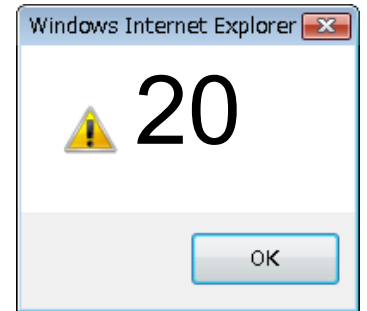
なので、ブラウザは今後、
この変数を5として扱います。

```
var foo = 5  
var baz = 5 * 4;  
alert(baz);
```



あとの処理は、今までどおりです。

```
var foo = 5;  
var baz = 5 * 4;  
alert(baz);
```



変数については、こんな感じでした。

では、関数の場合はどうなるのでしょうか？

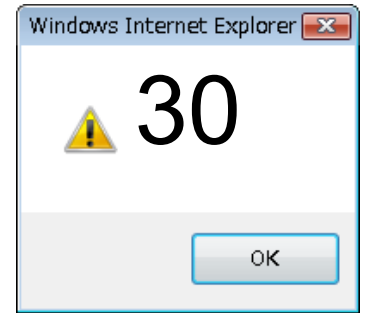
最後のプログラムです。

```
function rex() {  
    return 5 * 4;  
}  
  
var foo = rex() + 10;  
alert(foo);
```

こんな実行結果です。

```
function rex() {  
    return 5 * 4;  
}
```

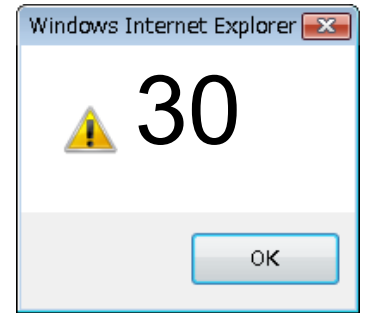
```
var foo = rex() + 10;  
alert(foo);
```



関数の宣言箇所は、これ全部が「1行」
として扱われます。

```
function rex() {  
    return 5 * 4;  
}
```

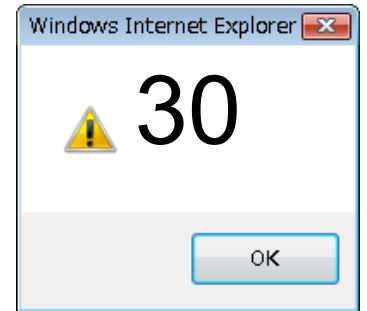
```
var foo = rex() + 10;  
alert(foo);
```



ブロックの中身は実行されません。

```
function rex() {  
    return 5 * 4;  
}
```

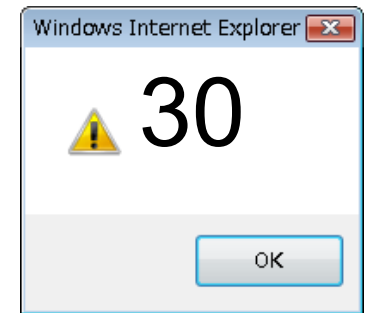
```
var foo = rex() + 10;  
alert(foo);
```



「こういう関数を作ったから、覚えとけ」
という一行の命令なのです。

```
function rex() {  
    return 5 * 4;  
}
```

```
var foo = rex() + 10;  
alert(foo);
```

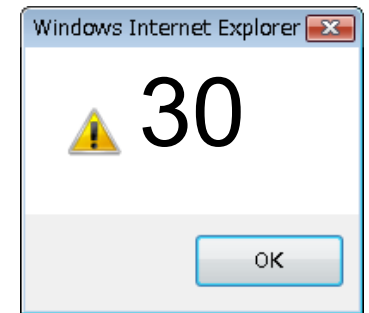


で、「2行目」にあたるこの行の、
ここに注目してください。

```
function rex(){  
    return 5 * 4;  
}
```



```
var foo = rex() + 10;  
alert(foo);
```

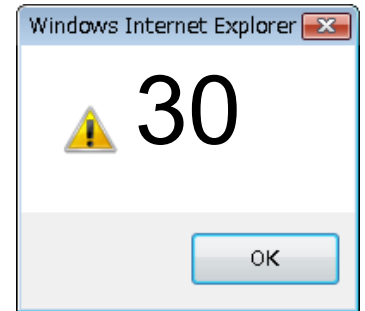


ブラウザはこのような記述を見つけたら、「関数を実行」しようとしています。

```
function rex() {  
    return 5 * 4;  
}
```



```
var foo = rex() + 10;  
alert(foo);
```

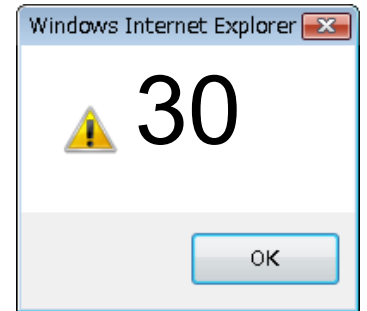


さっきのこれです。



```
function rex() {  
    return 5 * 4;  
}
```

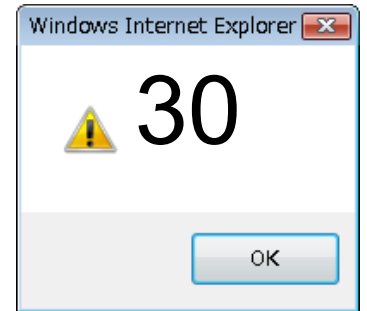
```
var foo = rex() + 10;  
alert(foo);
```



処理を行うと、

```
function rex() {  
    return 5 * 4;  
}
```

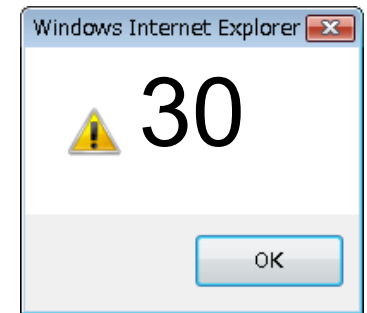
```
var foo = rex() + 10;  
alert(foo);
```



「reutrnr」の右側の値が
「20」になりました。

```
function rex() {  
    return 20;  
}
```

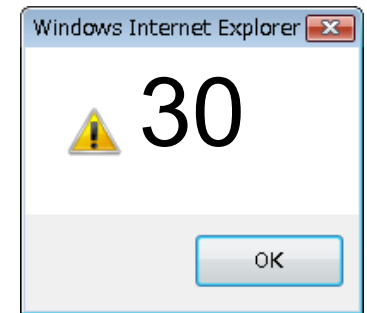
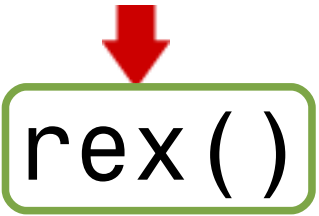
```
var foo = rex() + 10;  
alert(foo);
```



関数の呼び出し部分は今後、
この「return」の右側の値として扱われます。

```
function rex() {  
    return 20;  
}
```

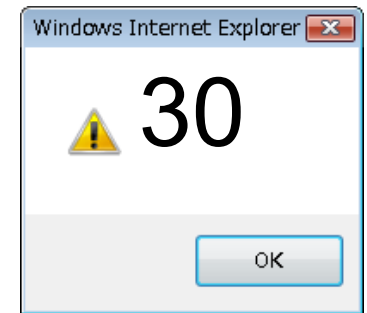

```
var foo = rex() + 10;  
alert(foo);
```



関数の呼び出し部分は今後、
この「return」の右側の値として扱われます。

```
function rex() {  
    return 20;  
}
```

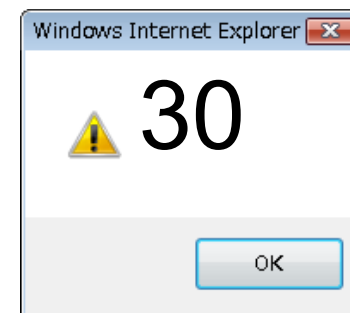
```
var foo = 20 + 10;  
alert(foo);
```



この「return」の右側の値を「戻り値」と呼びます

```
function rex() {  
    return 20;  
}
```

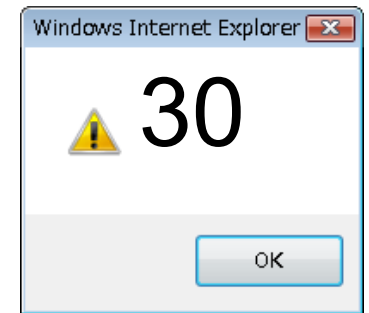
```
var foo = 20 + 10;  
alert(foo);
```



20 + 10の説明は、もう大丈夫ですね。

```
function rex() {  
    return 20;  
}
```

```
var foo = 20 + 10;  
alert(foo);
```



とりあえず

このスライドは以上です。

ありがとうございました。

では、ちょっと挙手アンケートにご協力ください。

自己紹介

- 田中朋也

- tiny-studio.com
- (タイニースタジオと読みます。)

- やっていること

- プログラム
- 講師
- 執筆

- 各種講演など



- オープンソース開発



- 言語

- PHP
- JAVA
- JavaScript
- ActionScript3.0

自己紹介

- 田中朋也
 - tiny-studio.com
 - (タイニースタジオと読みます。)
- やっていること
 - プログラム
 - 講師
 - 執筆

- 執筆
 - webcreators
 - 2010年1月号
 - などなど…



- WEBメディア

MdN DESIGN
INTERACTIVE