

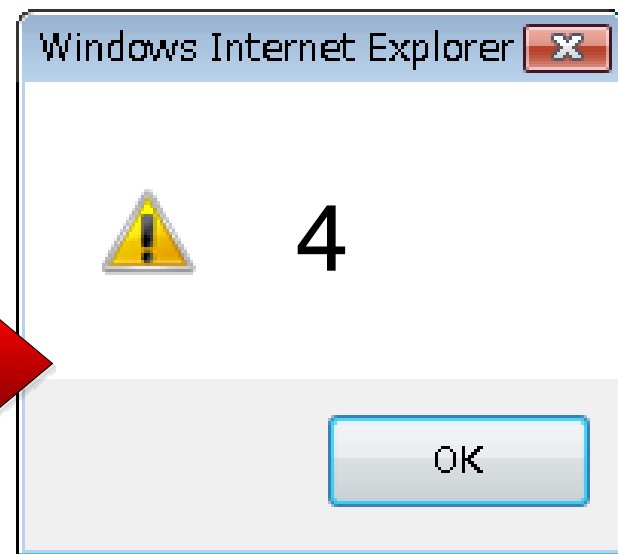
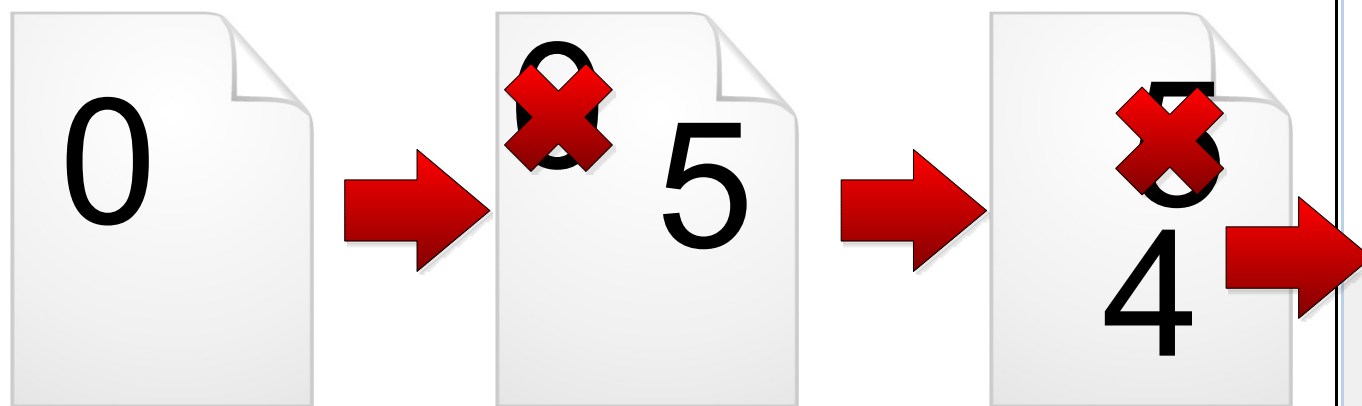
なんとなくな知識を再確認して、
苦手意識を克服しましょう。

順序がとても大切

変数について

- メモ用紙のようなもの
- 代入演算子で、書き込めます。
- 何らかの値が代入済みの変数に、別の値を代入すると上書きされます。

```
var a = 0;  
a = 5;  
a = 4;  
alert(a);
```



演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方

```
var a = 4 * 5 + 8;
```

演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方



```
var a = 4 * 5 + 8;
```

演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方



```
var a = 4 * 5 + 8;
```

演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方

①

```
var a = 4 * 5 + 8;
```

```
var a = 20 + 8;
```

```
var a = 28;
```

演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方

①

```
var a = 4 * 5 + 8;
```

```
var a = 20 + 8;
```

```
var a = 28;
```

演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方

①
`var a = 4 * 5 + 8;`



`var a = 20 + 8;`

`var a = 28;`

演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方

①
`var a = 4 * 5 + 8;`



②
`var a = 20 + 8;`

`var a = 28;`

演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方

①
`var a = 4 * 5 + 8;`



②
`var a = 20 + 8;`

`var a = 28;`

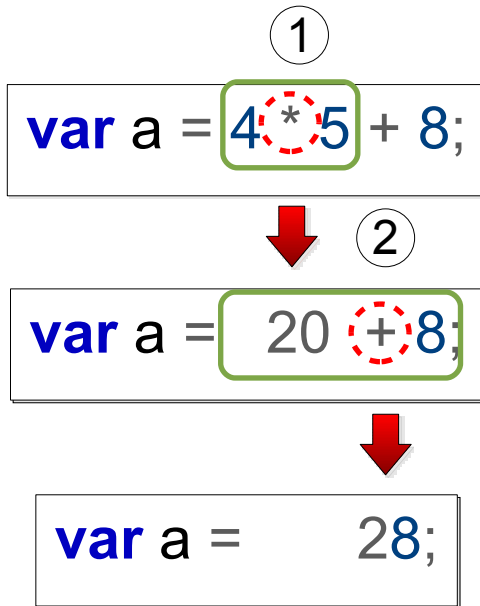
演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方



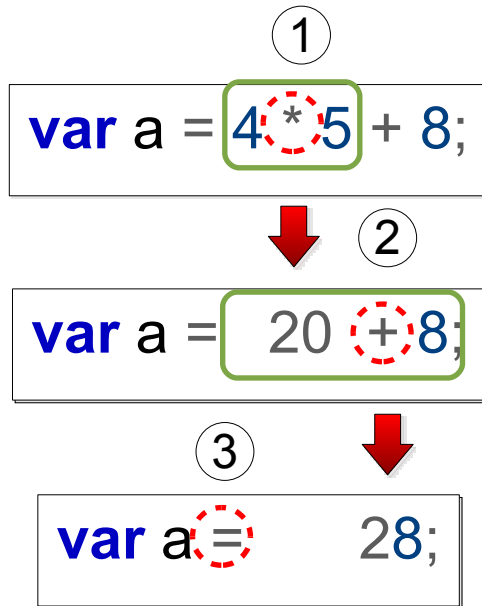
演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方



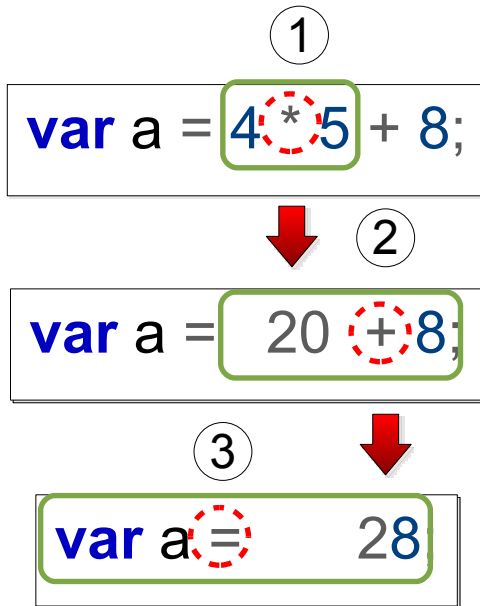
演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方



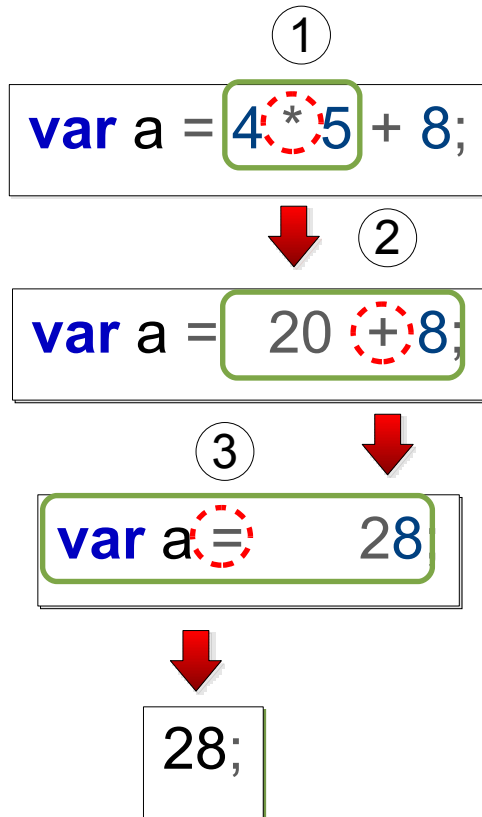
演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方



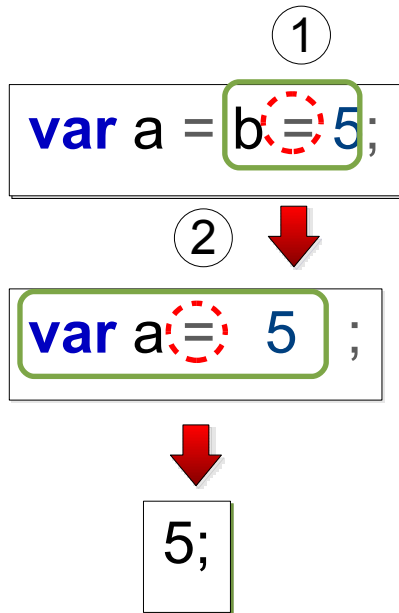
演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

演算子の実行順序

mdc 演算子 優先順位

検索

• この表の使い方



演算子の種類	個々の演算子
メンバ	. []
個々の演算子	() new
否定/インクリメント	! ~ - + ++ -- typeof void delete
乗算/除算	* / %
加算/減算	+ -
ビットシフト	<< >> >>>
関係	< <= > >= in instanceof
等価	== != === !==
ビットごとの and	&
ビットごとの xor	^
ビットごとの or	
論理積	&&
論理和	
条件	?:
代入	= += -= *= /= %= <<= >>= >>>= &= ^= =
コンマ	,

制御構造

- 上から順番に実行されるのがプログラムの基本
- ただ、条件に応じて実行されない行を作ることも時には必要
- また、前の行に戻ってもう一度同じ処理を行うこともよくある
- これを実現する仕組みが制御文

制御構造

- if文

- ここも、「はいはい」と読み飛ばしがちなので注意。
- if文には二つのエリアがある。
- ()のなかと、その次の行。
- {}は次の行が複数行の場合に使う。(誤解によるバグを防ぐため、一行でも使った方がいいです。)

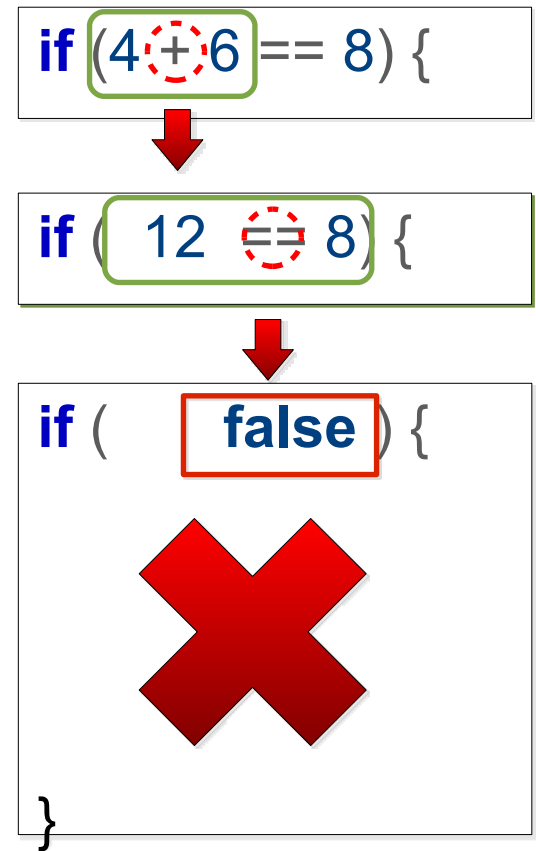
```
if (4 + 6 == 8)  ;
```

```
if (4 + 6 == 8) {  
      
}
```

制御構造

- if文

- 注意して欲しいのは、()の中。
- if文は、()の中が「true/false」のどちらであるかにしか興味がない。
- 「 $4 + 6 == 8$ 」という状態には、興味がない。
- 今回も演算子进行处理して、一つだけの答えを出しましょう。



制御構造

- if文

- ()の中は、どんな条件でも実行されるので注意。
- 実行されない可能性があるのは{}の中。



```
var hour = 20;
```



```
if (22 <= hour) {
```

```
  ✖ alert("ラストオーダーは0時になっていきますのでご注意ください");  
}
```



```
alert("お席はこちらのテーブルになります。");
```

制御構造

- while文

```
while (4 + 6 == 8)  ;
```

- ここも、「はいはい」と読み飛ばしがちなので注意。
- while文にも二つのエリアがある。
- ()のなかと、その次の行。
- {}は次の行が複数行の場合に使う。(誤解によるバグを防ぐため、一行でも使った方がいいです。)

```
while (4 + 6 == 8) {  
      
}
```

制御構造

- while文

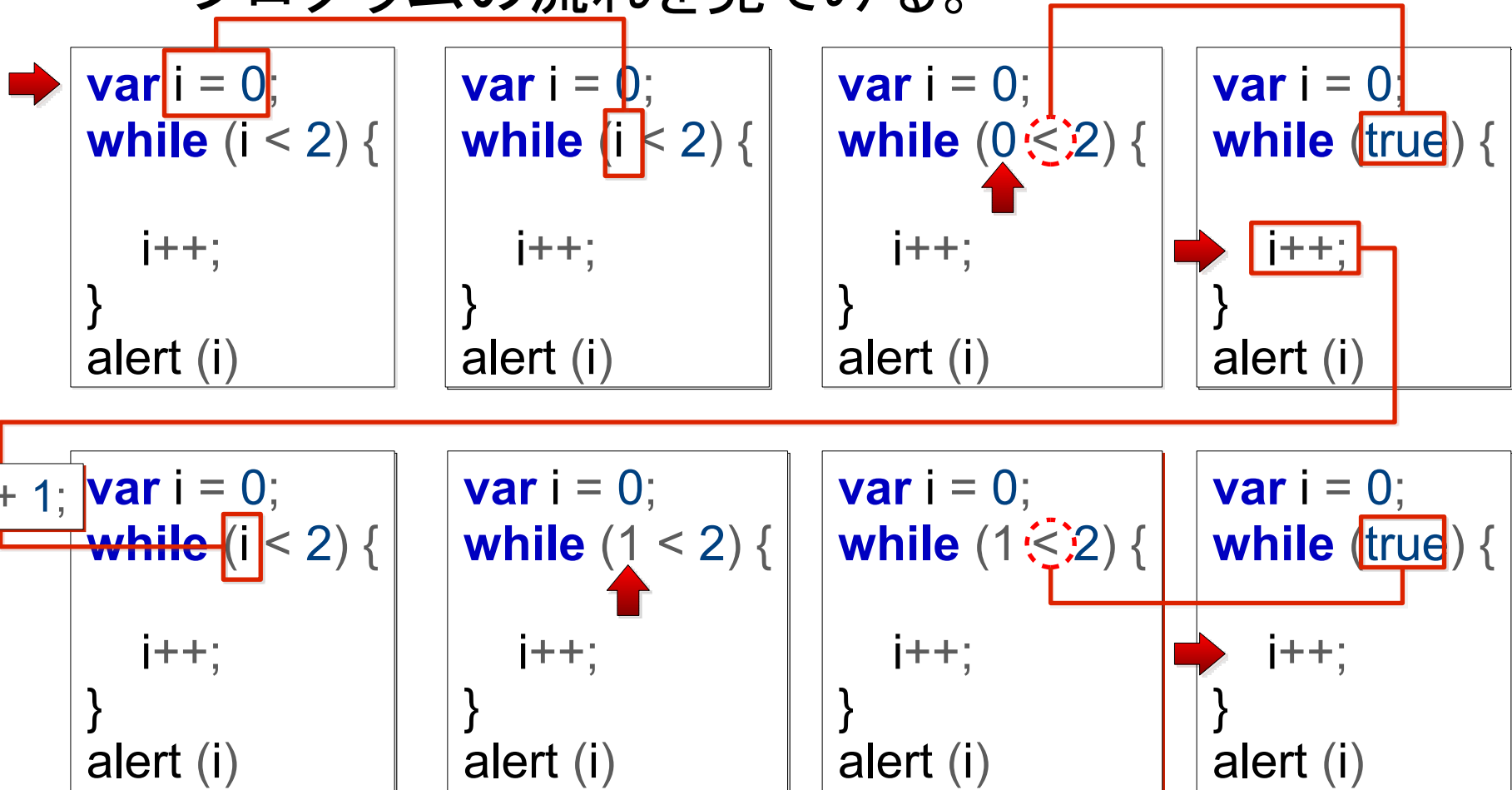
- 注意して欲しいのは、()の中。
- while文も、()の中が「true/false」のどちらであるかにしか興味がない。
- 「 $i < 3$ 」という状態には、興味がない。
- 今回も演算子を処理して、一つだけの答えを出しましょう。

```
var i = 0;
while (i < 2) {
    i++;
}
alert (i)
```

制御構造

- while文

- プログラムの流れをしてみる。



制御構造

- while文
 - プログラムの流れを見してみる。

```
var i = 0;  
while (true) {  
    i++;  
}  
alert (i)
```

```
1 + 1; var i = 0;  
while (i < 2) {  
    i++;  
}  
alert (i)
```

```
var i = 0;  
while (2 < 2) {  
    i++;  
}  
alert (i)
```

```
var i = 0;  
while (1 < 2) {  
    i++;  
}  
alert (i)
```

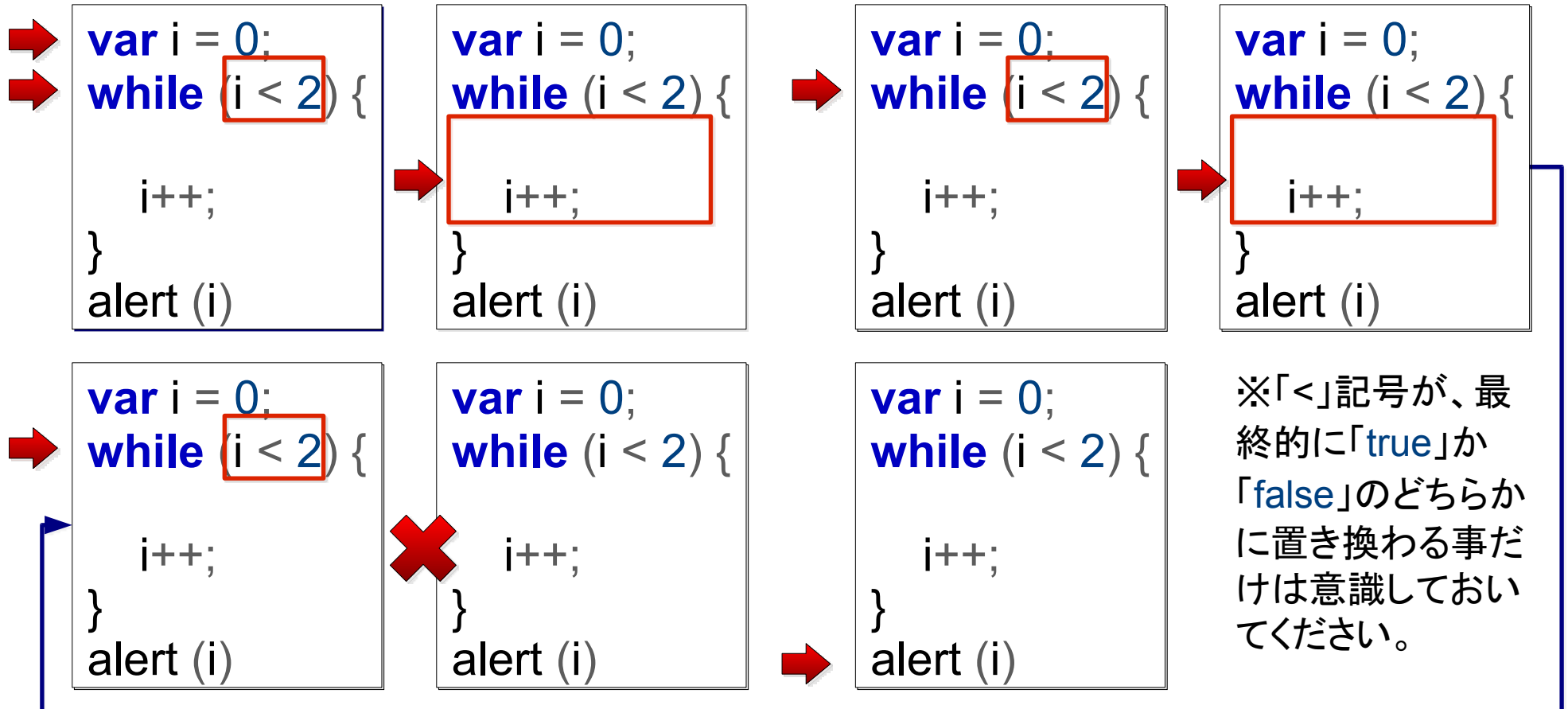
```
var i = 0;  
while (false) {  
    i;  
}  
alert (i)
```

```
var i = 0;  
while (i < 2) {  
    i++;  
}  
alert (i)
```



制御構造

- while文
 - はやい話が。



制御構造

- for文
 - for文の大きな特徴は、if文やfor文と違って、エリアが4つもあること。
 - 初期化するところ
 - 処理を行うかどうか決めるところ
 - 処理を書くところ
 - 処理が終わった後に次のステップの準備をするところ
 - 色分けしてwhile文に例えるとこんな感じ。

```
for ( □; □; □ ) {  
    □  
}
```

```
var i = 0;  
while ( i < 2 ) {  
    □  
    i++;  
}
```

制御構造

- for文
 - while文とは違い、構文の中に**前処理**と**後処理**を書く場所が、エリアとして予めが用意されているのが特徴。
 - 条件によっては、一度も処理が行われないこともあることに注意。

```
for ( □; □; □ ) {  
    □  
}
```

```
var i = 0;  
while ( i < 2 ) {  
    □  
    i++;  
}
```

制御構造

※「<」記号が、最終的に「true」か「false」のどちらかに置き換わる事だけは意識しておいてください。

```
for (var i = 0; i < 2; i++) {  
}  
alert(i)
```

```
for (var i = 0; i < 2; i++) {  
}  
alert(i)
```

```
for (var i = 0; i < 2; i++) {  
      
}  
alert(i)
```

```
for (var i = 0; i < 2; i++) {  
}  
alert(i)
```

```
for (var i = 0; i < 2; i++) {  
}  
alert(i)
```

```
for (var i = 0; i < 2; i++) {  
      
}  
alert(i)
```

```
for (var i = 0; i < 2; i++) {  
}  
alert(i)
```

```
for (var i = 0; i < 2; i++) {  
}  
alert(i)
```

```
for (var i = 0; i < 2; i++) {  
      
}  
alert(i)
```

```
for (var i = 0; i < 2; i++) {  
}  
alert(i)
```

