



# 多言語 プログラミング

～プログラミングにおける骨格～



# 概要

～まえがき～

# 講義目的

- 多言語に渡るプログラミングを独学する術を身に着ける
- 基本的な用語や技術を学び、活かせる力を身に着ける
- マイコンやウェブなど、広い範囲に適応できる人材育成

# 進め方

- 重要な章ごとにまとめて講義
- 1 日（2時間） 1～2章分を進めたい
- 各章ごとに演習がある

# 進め方

- プログラミングだけではなく、**設計**等の講義も行う
- 著者の**サイト**を用いた説明もある
- かなり分かりやすく**簡略化**しています。頑張ってください^^



# プログラミングとは

～基本のキ～

# プログラミングどんなもの？

- プログラミングにはさまざまな種類が…  
ウェブ、マイコン、ゲーム…etc
- 大まかには2種類  
オブジェクト指向と構造化プログラミング（後述）
- 骨格は基本的に共通  
関数や変数など、基本的な骨格がある

# プログラミング言語

- 言語は多数ある

C, C++, C#, JavaScript, VisualBasic, Python...etc

- 本教材ではJavaScriptとArduinoによるC++を使用

※その他にも様々な言語での文法や解析法を紹介



# 基本的な骨格とは？

- 冒頭でライブラリの宣言
- メインループで処理、UI等の場合ではイベントを処理
- サブルーチン（関数）にて繰り返し処理

…etc

```
#include <stdio.h>

int main(void){
    printf('Hello World');
    return 0;
}
```

```
void setup(){
    pinMode(2, OUTPUT);
}

void loop(){
    digitalWrite(2, HIGH);
    delay(200);
    digitalWrite(2, LOW);
    delay(200);
}
```

# 演習 1 “プログラミングとは”

- JavaScript動作確認

1. VSCodeの作業フォルダを作成

2. スクリプトで“Hello World”を表示させる。

# 演習 1 “プログラミングとは”


- JavaScriptの書き方
  - htmlファイルの<script>タグ内に記述する
- ディスプレイさせる関数は、document.writeln()を使用
- 文末には;(セミコロン)を付ける

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
<title>Hello, World! </title>
</head>
<body>
<pre>
<script type="text/javascript">
//ここにスクリプトを記述
</script>
<noscript>JavaScriptが利用できません。</noscript>
</pre>
</body>
</html>
```

# 演習 1 “プログラミングとは” 回答

```
document.writeln('Hello, World !');
```





# 変数とデータ型

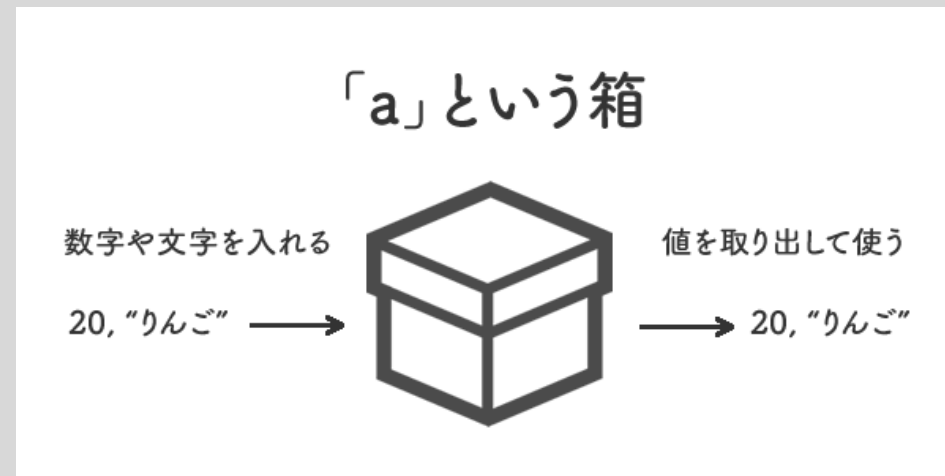
～プログラミングで使用する箱～

# 変数

- データを入れる箱

- 入れるデータによって種類がある

- 箱をひとまとめにしたもの → 配列



# 基本要素

- 宣言

変数を**作成**すること

- 初期化

変数の作成とともに**数値を割り当てる**こと

- 代入

変数の値を**上書き**すること

# 宣言

```
var x;
```

- どのような言語でも**基本同じ**  
(python等では宣言が暗黙)

```
int x;
```

- **(修飾子) + 型 + 変数名**で宣言する
- 変数名は**予約語**以外なら原則何でも可



# 初期化

```
var x = 100;
```

```
int x = 100;
```

- 宣言とともに値を割り当てる  
(Pythonでは宣言と初期化がセット)

```
int x = 'a';
```

- 型 + 変数名 = 初期値で宣言 + 初期化をセットで
- 初期値の型が一致している必要がある (詳細はp13にて)

# 初期化



- 初期化をしないと...

ランダムに適当な値“**不定値**”が割り当てられる

javascriptでは“**undefined**”と出る

**予測していない結果**になる可能性がある



- 基本的に**宣言**と**初期化**はセットでする

→初期化しないで使用するのは×

# 代入

- 変数を上書きして内部の値を更新する

```
var x = 100;
```

```
x = 200;
```

- 宣言後、任意の場所で 変数名 = 値
- = は実は演算子（詳細は演算子のところで）

# データ型とは？

- 変数に入るデータは予め決めておかなければいけない
  - 整数（`int`）型、文字（`char`）型など多数ある
- ※`var`型や`let`型は直接的なデータ型ではない

# データ型とは？

## 。様々なデータ型一覧

**言語**によってばらつきがあるが、基本的には以下の表

分類	名前	例
論理型	bool	True, false
文字型	char	a, b, c
文字列型	string	"abc"
整数型	int	1, 333
浮動小数点型	float, double	0.5, 0.0093
配列型	array	[1, 2, 3]
オブジェクト型	object	{x:1, y:2, z:3}

# var型とlet型は？

- **variety**（多様）からきている  
JavaScriptでは**var**や**let**を用いて変数を使用
- **自動**でデータ型が判定される  
→ **直接**のデータ型ではない
- varは**宣言の重複**が可(上書きされる)、letは不可

# データ型の使い分け

- **サイズ**による使い分け

変数の**データ型**には**サイズ**がある

ex) int : 32bit、char : 16bit..etc 言語によってばらつきがある

- **オーバーフロー**

サイズを超えると**エラー**が起きたり、**予想しない結果**が...

※使う言語によるサイズの確認は**必須**

# スコープとは？

変数には使える**範囲**がある

- **グローバル変数**  
ファイル全体が有効範囲。  
使いすぎるとメモリが..
- **ローカル変数**  
その関数の内部のみ。  
保持するにはstaticなどの修飾子を使用

グローバルスコープ：スクリプト全体で有効

```
var global_Data = 'hogehoge' ;
```

グローバル変数

ローカルスコープ：関数の中でだけ有効

```
function foobar() {  
    var local_Data = 'ほげほげ' ;  
}
```

ローカル変数

...

※詳細は関数の章で説明



# 修飾子とは？

- **オブジェクト**やスコープに関わる変数の**権限**のようなもの  
詳しくは**オブジェクト**の章で説明
- クラス間の変数にアクセスできないように**スコープ（アクセス）**  
を調節できたりする

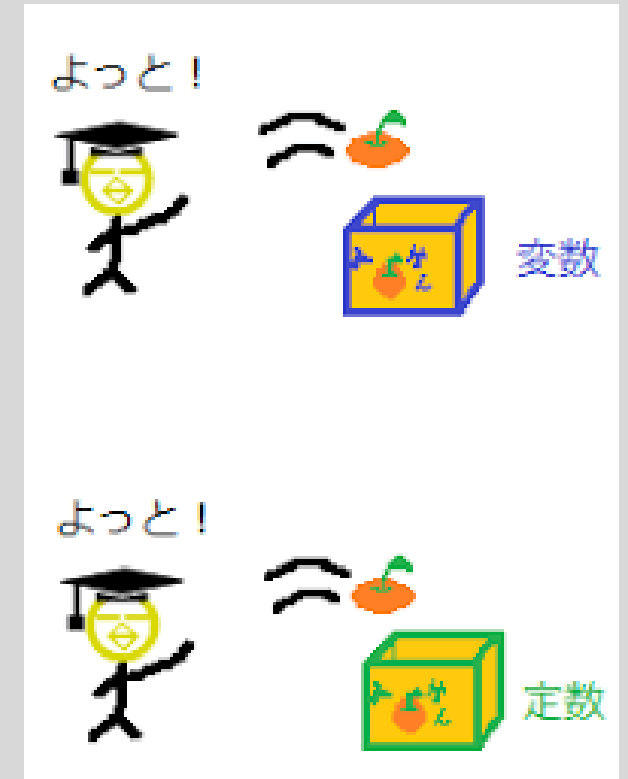
※言語によっては若干のばらつきがあるので注意が必要

# 修飾子の種類

- 変数のアクセス権限を宣言できる  
`private` や `public` など...
- 変数の保持に使用  
`static` など...
- ほかにも様々なものがある

# 定数

- 変数とは別に、数値に命名できる
- `const`修飾子を使用し、`const 定数名 = 値`
- 定数は変更することができない



## 演習 2 “変数とデータ型”

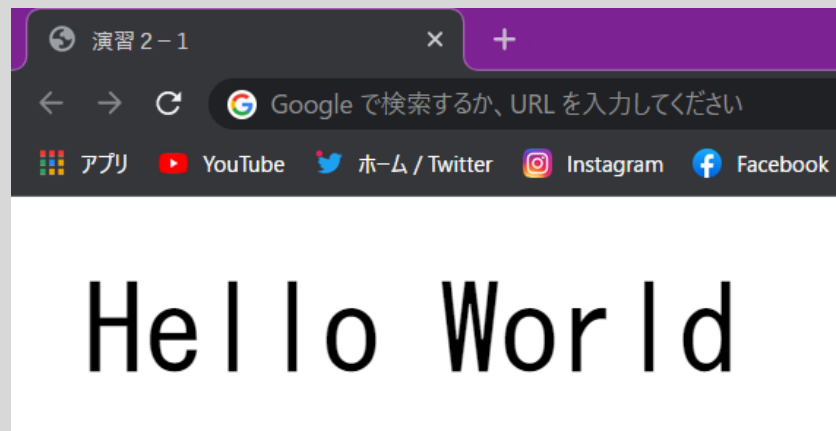
1. `var`型の変数を用いてHello Worldを表示させる
2. `var`型と`let`型の違いを確認する
  - 変数のメモリとスコープについて確認する  
→関数の章で確認するため、回答を見て確認

## 演習 2 “変数とデータ型”① 回答

- 宣言と初期化を使う方法と、宣言しその後代入という方法がある

```
var str = 'Hello World';  
document.writeln(str);
```

```
var str;  
str = 'Hello World';  
document.writeln(str);
```

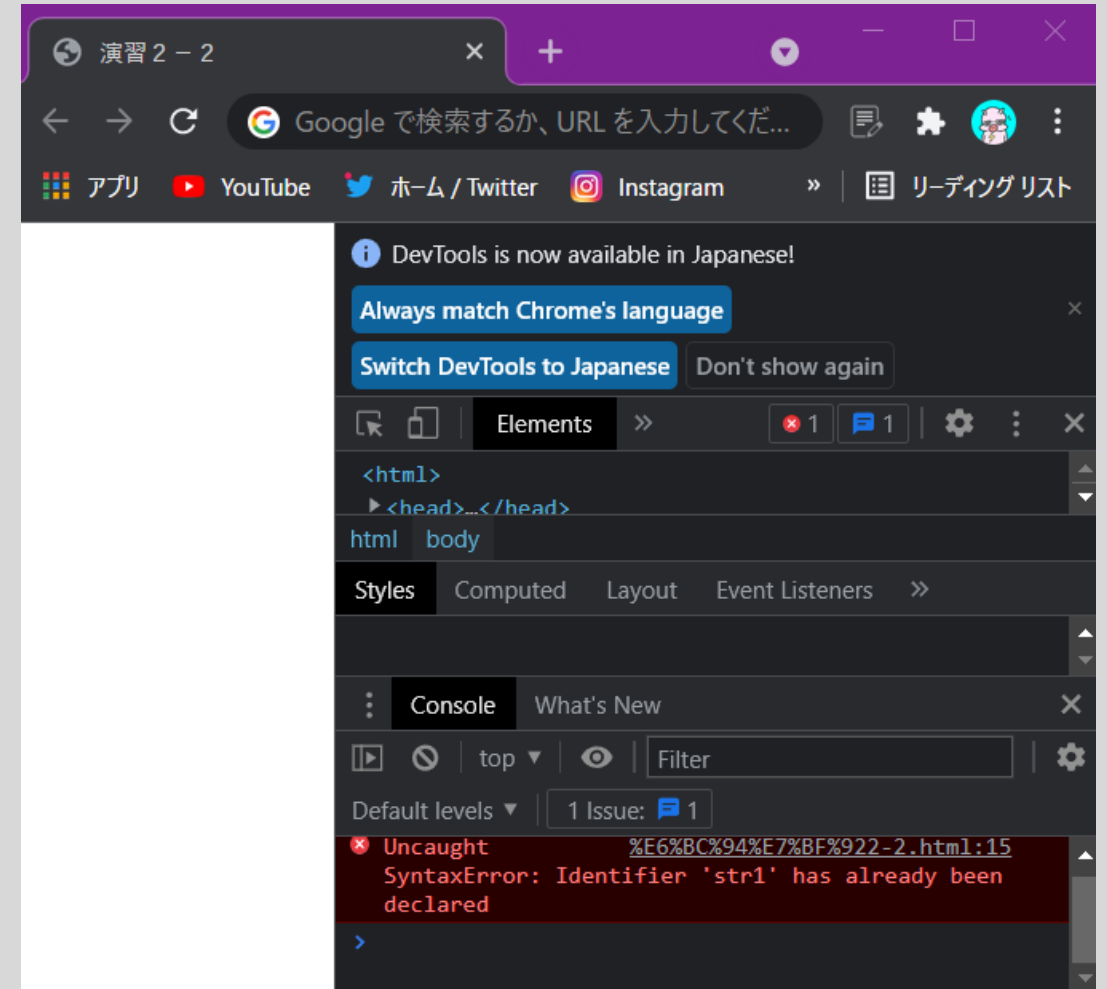


## 演習 2 “変数とデータ型”②

- **var**型と**let**型をそれぞれ再宣言するテストコードを実行し、  
両方を実行して確認する

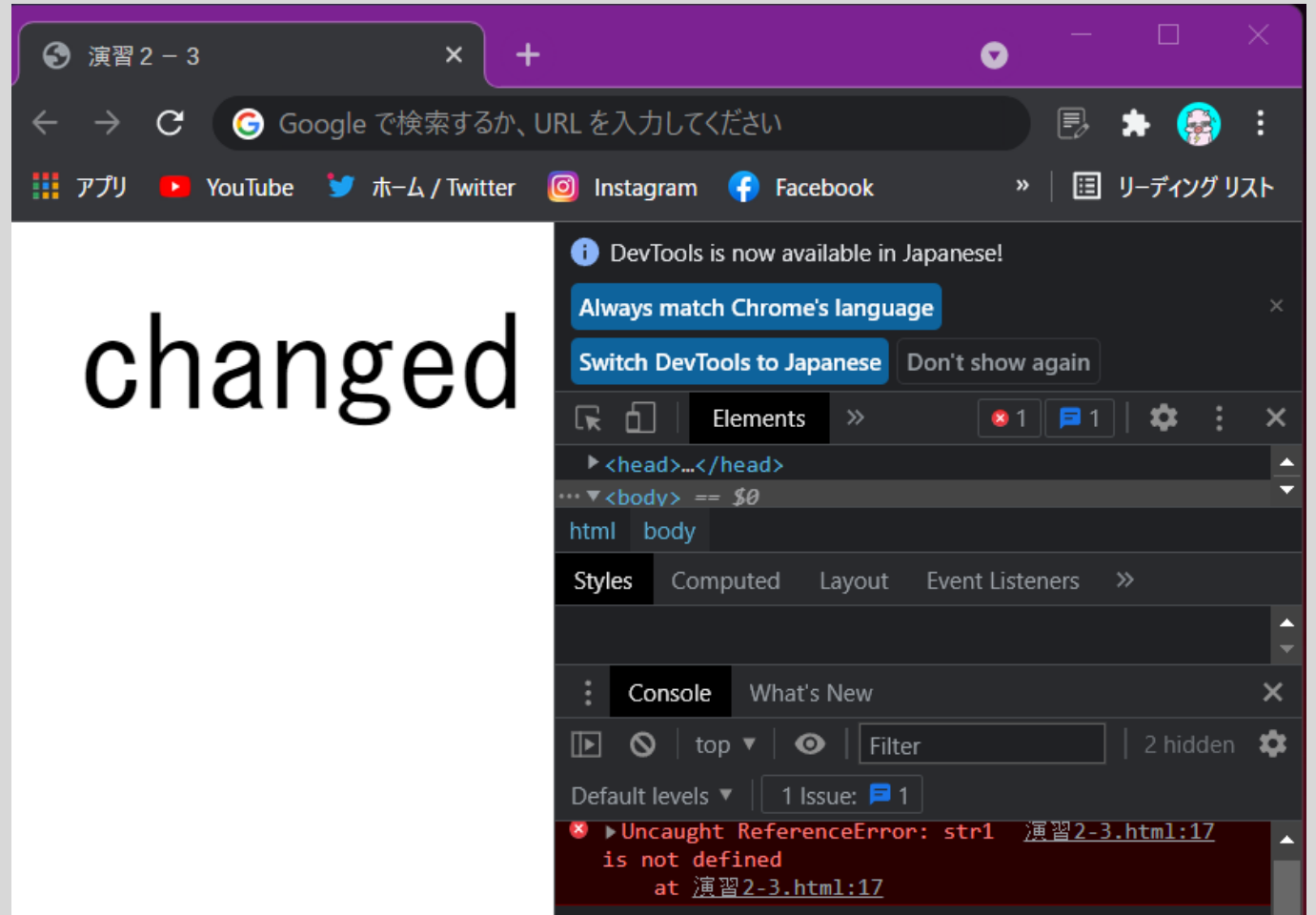
## 演習 2 “変数とデータ型”② 回答

```
var str = 'Hello World';  
var str = 'Hello JS';  
document.writeln(str);  
  
let str1 = 'Hello World';  
let str1 = 'Hello JS';  
document.writeln(str1);
```



## 演習 2 “変数とデータ型”③ 回答

```
var str = 'Hello World';
function test(){
  str = 'changed';
  var str1 = 'Hello Java';
}
test();
document.writeln(str);
document.writeln(str1);
```





## ここまでのまとめ

- プログラミングは**基本的な骨格**がある
- 変数はデータを入れる箱であり、**宣言**、**初期化**、**代入**などで操作する
- 変数は**スコープ**や**データ型**を確認する必要がある

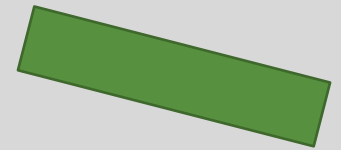


# 演算

～プログラムの演算～

# 演算の方法

- プログラミングは、**演算の繰り返し**で処理をする
- 演算には、**演算子**というものをを用いる
- 演算子の**優先順位**によって、多様な演算を実現する



# 演算子の種類

- 演算子にはさまざまな種類がある
- 算術演算子、代入演算子、比較演算子、  
論理演算子、ビット演算子
- ↑が基本の演算子で、どの言語にもほとんど共通している

# 算術演算子

- 数学的な演算を行う演算子

```
var x = 5;  
var y = 3;  
x = x + 3;    //x : 8  
x = x * y;    //x : 24
```

- 加減算や乗除、剰余を求めるものがある

- 記法は数学と同じように、値 演算子 値 として計算を行う（2項）

1    +    2

# 基本的な算術演算子

- 数学的な演算を行う演算子

- べき乗演算子については  
言語によりけり

演算子	概要	例
+	加算	$3 + 5$ (8)
-	減算	$5 - 2$ (3)
*	乗算	$5 * 2$ (10)
/	除算	$5 / 2$ (2.5)
%	剰余	$5 \% 2$ (1)
**	べき乗	$5 \wedge 2$ (25)

- 演算結果は基本的に変数のデータ型へ型変換（後述）される

※ 言語によって違うので注意

# 算術演算子（文字列の結合）

- 加算演算子を用いた文字列の結合が可能な言語が存在
- “文字列” + “文字列” として文字列を結合  
※PHPの場合、“文字列” . “文字列” として結合が可能
- 言語によって型の制約があるため注意  
※C#などの場合、文字列と整数型の加算は不可…etc

```
<?php
$str1 = 'Hello';
$str2 = 'World';

//HelloWorld
$str = $str1 . $str2;
?>
```

```
var str1 = 'Hello';
var str2 = 'World';
var str = str1 + str2; //HelloWorld
```

# インクリメント/デクリメント演算子

- 1 を加算/減算するときに便利な演算子

```
var x = 2;  
x++;    //x : 3  
++x;    //x : 4
```

- インクリメントは'++'、デクリメントは'--'で記述  
++は "+ 1"と同義、--は"- 1"と同義

- 記法としては、変数名 演算子 又は 演算子 変数名 とするだけ  
※演算子の位置により演算の順序が異なる（次頁）



# インクリメント/デクリメント演算子の順序

- 後置演算

演算対象の変数を処理した後、演算を行う

```
var x = 2;  
var y = x++;  
//yに代入 → xが3になる  
document.writeln(x); //3  
document.writeln(y); //2
```

- 前置演算

演算対象の変数を演算した後、処理を行う

```
var x = 2;  
var y = ++x;  
//xを加算 → yに代入  
document.writeln(x); //3  
document.writeln(y); //3
```

# 単項算術演算子

- 1 つの変数に対し演算を行う演算子
- 符号や論理の反転を行う

演算子	概要	例
+	変数(整数)の値を出力	+a (aがそのまま)
-	変数(整数)の符号を反転	-a (aの符号が反転)
!	変数(論理)の否定(not)	!a (aが0ならば1, 1ならば0)
~	変数(ビット)の1の補数	~a (aのすべてのビットを反転)

# 代入演算子（単純代入演算子）

- 演算した結果や値を、変数に設定（**代入**）するための**演算子**  
→ "**=**" : 代入については前章参照
- 代入するには**型が一致**してる必要がある
- **参照による代入**と**値による代入**、**分割代入**がある  
※ 参照による代入についてはオブジェクトの章で後述



# 代入演算子（複合代入演算子）

- 算術演算子やビット演算子と代入演算子が連動した演算子
- 言語によってはばらつきが...

# 複合代入演算子の種類

演算子	概要	例
<code>+=</code>	加算したものを代入	<code>x += 2 (x = x + 2)</code>
<code>-=</code>	減算したものを代入	<code>x -= 3 (x = x - 3)</code>
<code>*=</code>	乗算したものを代入	<code>x *= 2 (x = x * 2)</code>
<code>/=</code>	除算したものを代入	<code>x /= 5 (x = x / 5)</code>
<code>%=</code>	剰余を計算し代入	<code>x %= 3 (x = x % 3)</code>
<code>&amp;=</code>	論理積を計算し代入	<code>x &amp;= 4 (x = x &amp; 4)</code>
<code> =</code>	論理和を計算し代入	<code>x  = 5 (x = x   5)</code>
<code>^=</code>	排他的論理和を代入	<code>x ^= 3 (x = x ^ 3)</code>
<code>&lt;&lt;=</code>	左にシフト演算し代入	<code>x &lt;&lt;= 2 (x = x &lt;&lt; 2)</code>
<code>&gt;&gt;=</code>	右にシフト演算し代入	<code>x &gt;&gt;= 4 (x = x &gt;&gt; 4)</code>
<code>.=</code>	文字列を結合(PHP)	<code>x .= 'end' (x = x . 'end')</code>

# 比較演算子

- 条件分岐などで使用される、**論理の真偽**を評価する**演算子**
- 左辺と右辺の値を比較して、**True(1)**または**False(0)**を返す
- 言語によってばらつきがあるが、基本は同じ

# 比較演算子の種類

演算子	概要	例
==	左辺と右辺が等しいか	5 == 5(True, 1)
!=	左辺と右辺が等しくないか	5 != 5(False, 0)
<	左辺より右辺が大きいのか	5 < 5(False, 0)
<=	右辺が左辺以上か	5 <= 5(True, 1)
>	右辺より左辺が大きいのか	5 > 3(True, 1)
>=	左辺が右辺以上か	5 >= 3(True, 1)
===	左辺と右辺がデータ型まで等しいか	5 === 5(True, 1)
!==	左辺と右辺がデータ型まで等しくないか	5 !== 5(False, 0)
? :	三項演算子	(x == y) ? y + 1 : y

# 論理演算子

- 論理演算子を用いる条件式を組み合わせて全体を評価する演算子
- 条件分岐等の制御構文で用いられることが殆ど  
※詳細は制御構文の章で
- 式全体の評価を、ショートカットする演算も存在（後述）



# 論理演算子の種類

- 。言語によってほとんど同じ、以下のような**演算子**がある

演算子	概要	例
&&	左右の式が両方True(1)か	5 == 5 && 20 == 20(True, 1)
	左右どちらかの式がTrue(1)か	5 == 5    20 == 10(True, 1)
!	式全体の評価を反転	!(5 < 5)(True, 1)

# 論理演算子のショートカット演算

- 式を評価するとき、**左式の真偽のみ**で評価が決定する演算  
→ 右式の評価は行われない
- **&&**でのショートカット演算  
→ 左式がfalse(0)の場合、右式がどうであれ**false**になる
- **||**でのショートカット演算  
→ 左式がtrue(1)の場合、右式がどうであれ**true**になる

## 演習 3 “演算子”①

1. 算術演算子を用い、半径3[m]の球面の表面積を求める  
→円周率はMath.PIを用いる
2. 複合代入演算子を用い、任意の定数を4乗する

※論理演算子に関しては制御構文で演習

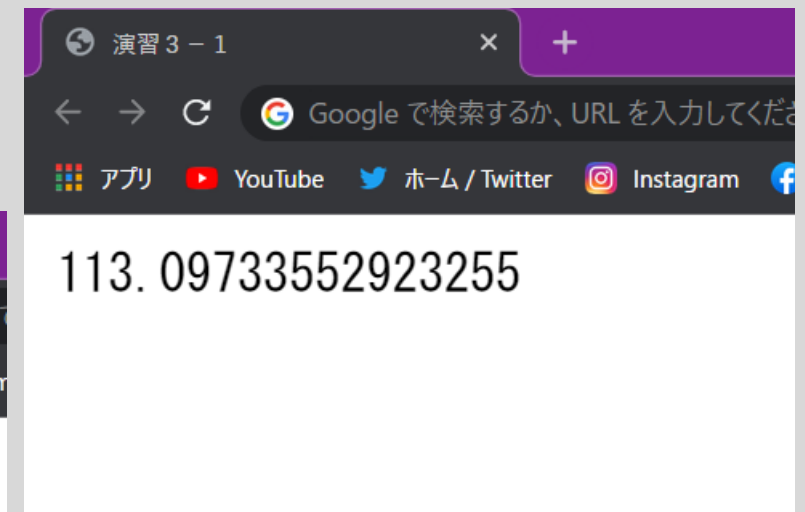
# 演習 3 “演算子”① 回答

1. 表面積 $4\pi r^2$ の公式を用いる

```
var r = 3;  
var area = 4 * Math.PI * r ** 2;  
  
document.writeln(area);
```

2. 演算子 $**=$ を用いる

```
var n = 5;  
var val = n **= 4;  
  
document.writeln(n);
```



# ビット演算子

- 整数値を2進数で表し、2進数のビットに対して演算を行う
- マイコンにおいてシリアル通信やI2C, SPI通信で必要
- 2進数で情報をやり取りすることは、非常に重要

# ビット演算とは？ ～ビット論理演算子～

```
101
|001
-----
101
```

- 。論理和 (OR) → "|"

日本語では「A又はB」と表現される

演算の左右どちらかが1ならば、結果が1になる (A+Bと同じ)

```
101
&001
-----
001
```

- 。論理積 (AND) → "&"

日本語では「AかつB」と表現される

演算の左右どちらかが0ならば、結果が0になる (A×Bと同じ)

# ビット演算とは？ ～ビットシフト演算子～

- 2進数のビットを右or左に、指定したビット数だけずらす演算
- ずらした後、端は0埋めに、はみ出しは切り捨てる  
→ ビット数は変わらない
- 右シフト">>", 左シフト"<<"

101  
----->>2  
001

101  
-----<<1  
010

## 演習 3 “演算子”②

1. ビット論理演算子を用いて、ドモルガンの法則を確認する



$$\sim A \ \& \ \sim B = \sim(A \mid B)$$

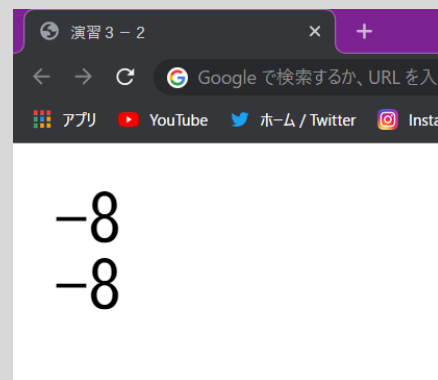
$$\sim A \mid \sim B = \sim(A \ \& \ B)$$

2. ビットシフト演算子を用いて、シフト演算が2の乗算になることを  
確認する



## 演習 3 “演算子”② 回答

```
var x = 5;  
var y = 7;  
  
var result1 = (~x) & (~y);  
var result2 = ~(x | y);  
  
document.writeln(result1);  
document.writeln(result2);
```



```
var x = 5;  
var y = 7;  
  
var result = 5 << 3;  
  
document.writeln(result);
```



# 型変換とキャスト（Cなど）

```
int x = 2;  
int y = 3;  
// int/doubleの計算  
// 暗黙的に型が変換される  
double ave = (x + y) / 2.0;
```

## 暗黙的な型変換

→違う型同士で演算したとき、どちらかに型変換されて演算

- 暗黙的なため、随時どの型に変換されているか**把握しにくい**  
→**可読性**も下がってしまう…

# 型変換とキャスト（Cなど）

```
int x = 2;  
int y = 3;  
// (double) + (式)  
// 明示的にdouble型として演算される  
double ave = (double)(x + y) / 2;
```

## 明示的な型変換（キャスト演算）

→(型) + 式 or 値 とすることで明示的に型変換が可能

### ◦キャスト演算子

→キャスト演算において (型) をキャスト演算子と呼ぶ

# 演算子の優先順位

- 演算には優先順位が存在
- 複雑な演算で意識が必要
- 右図はJavaScriptの例

優先順位	演算子
高       ↑	かっこ (())、配列 ([])
	インクリ/デクリメント、単項算術演算子
	乗算 (*)、除算 (/)、剰余 (%)
	加算 (+)、減算 (-)、文字列結合 (+)
	シフト演算子
	比較演算子 (<、<=、>、>=)
	比較演算子 (==、!=、===、!==)
	AND(&)
	OR( )
	論理積 (&&)
低	論理和 (  )
	三項演算子 (?:)
	代入演算子
	カンマ (,) : クラスで使用

# 演算子の結合則

- 演算子を左か右で  
**結合するか**を決定する
- 言語によって  
基本的に同じ

結合性	演算子の種類
左→右	算術演算子
	比較演算子
	論理演算子
	ビット演算子
	かっこや配列
右→左	インクリ/デクリメント
	代入演算子
	単項演算子
	三項演算子
	deleteやtypeof等

# 変換指定（Cなど）

- printfやscanf関数などで、**文字列**に**数値**を埋め込むときに使用
- %ab.cd（abcは定数）のように使用する
  - a: **0フラグ**、数値の前の余白を**0**で埋める（オプション）
  - b: **最小フィールド値**、最低限の**表示文字数**（オプション）
  - c: **精度**、表示する**最小の桁数**（オプション）
  - d: **変換指定子**、データ型に対応した文字が必要

# テンプレート文字列 (JavaScriptなど)

- 文字列へ変数を埋め込むときに使用する
- 文字列結合演算子(+)を使わずに、簡潔になる
- `${変数名}`として文字列に埋め込む  
→ `` (バッククォート) で囲む

```
let name = 'Jin';  
let str = `こんにちは、  
${name}さん。`;
```

## 演習 3 “演算子”③

1. 上底3、下底4、高さ8の台形の面積を求め、  
優先順位と結合性確認する
2. テンプレート文字列を用いて、1 の結果を文字列に埋め込む



## 演習 3 “演算子”③ 回答

```
var jotei = 3;  
var katei = 4;  
var takasa = 8;  
  
var result =  
    (jotei + katei) * takasa / 2;  
document.writeln(result);
```



```
var jotei = 3;  
var katei = 4;  
var takasa = 8;  
  
var result = (jotei + katei) *  
takasa / 2;  
document.writeln(`台形の面積は、  
${result}です。`);
```



# 演算子のまとめ

- 演算子には算術やビット、論理などさまざまな種類がある
- 代入演算子には単純と複合がある
- 演算子には優先順位や結合性があり、複雑な計算をするときに必要
- 変数は文字列に埋め込める



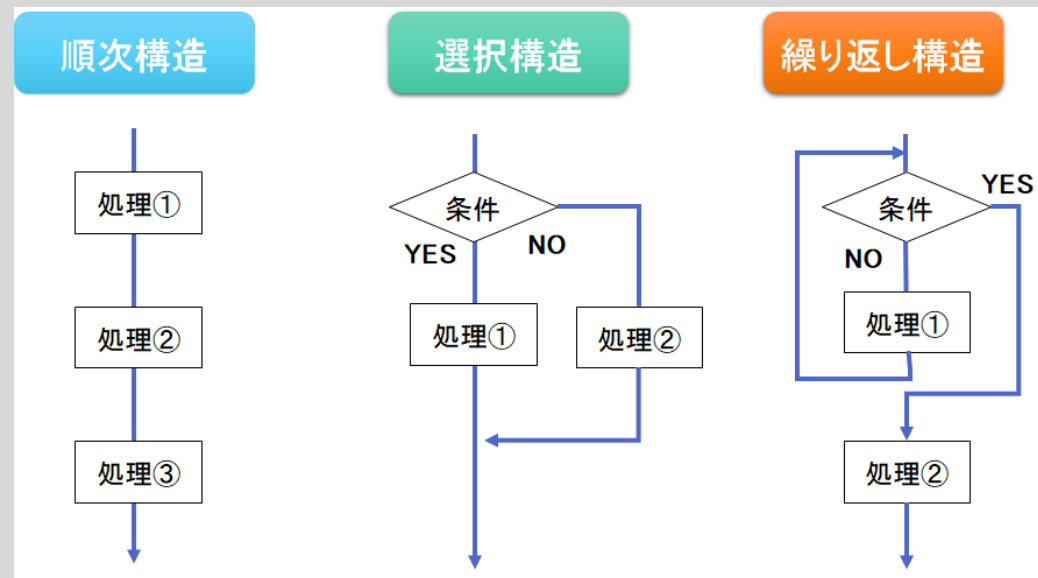
# 制御構文

～条件分岐～

# 制御構文

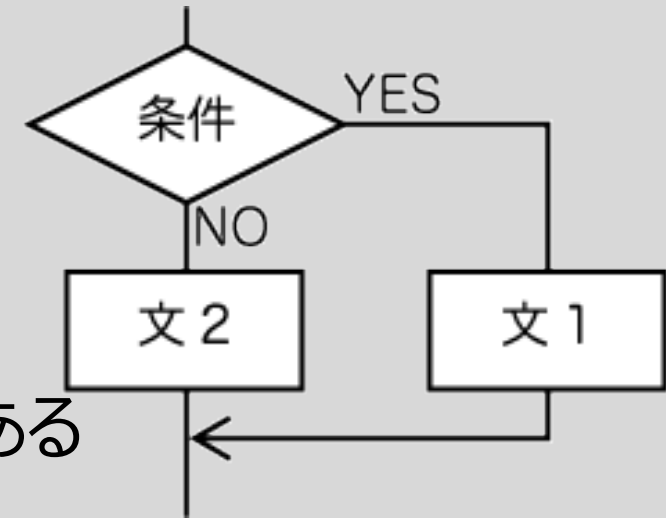
## 構造化プログラミングの手法

1. 記述された順番に処理を行う**順次**
2. 条件によって処理を分岐する**選択**
3. 特定の処理を繰り返し実行する**反復**



# if文～条件による処理の分岐～

- 時と場合に応じて、**処理の分岐**が必要  
→ **if文**や**switch文**がある
- if構文は**真**のとき実行する**if**と、  
**偽**のとき実行する**else**によって成立
- 条件式は**比較演算子**と**論理演算子**  
によって立てる



```
if(条件式){  
    //条件式がtrueの場合  
}else{  
    //条件式がfalseの場合  
}
```

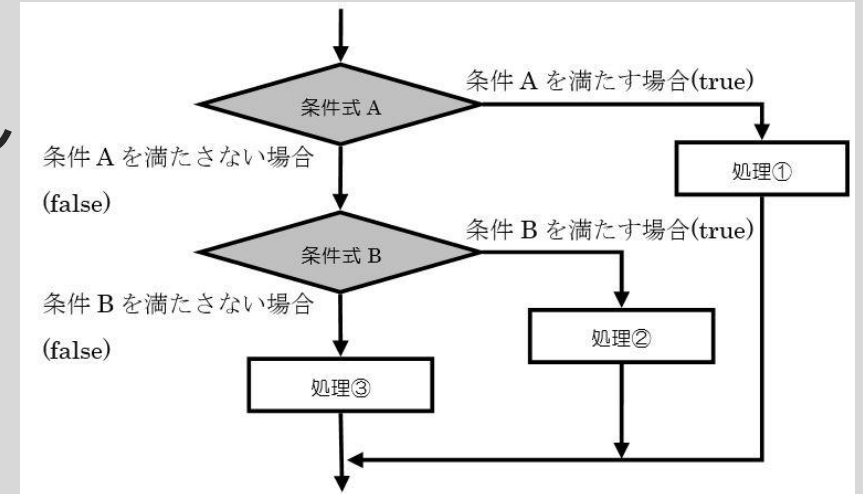
# if文～条件による処理の分岐～

## else-ifによる多岐分岐

- else if文を用いて多岐の分岐が可能

- 上から順に実行するため、評価は  
条件式1→条件式2の順番になる

- ※ switch文でも多岐分岐が可能  
可読性のために使い分けが必要



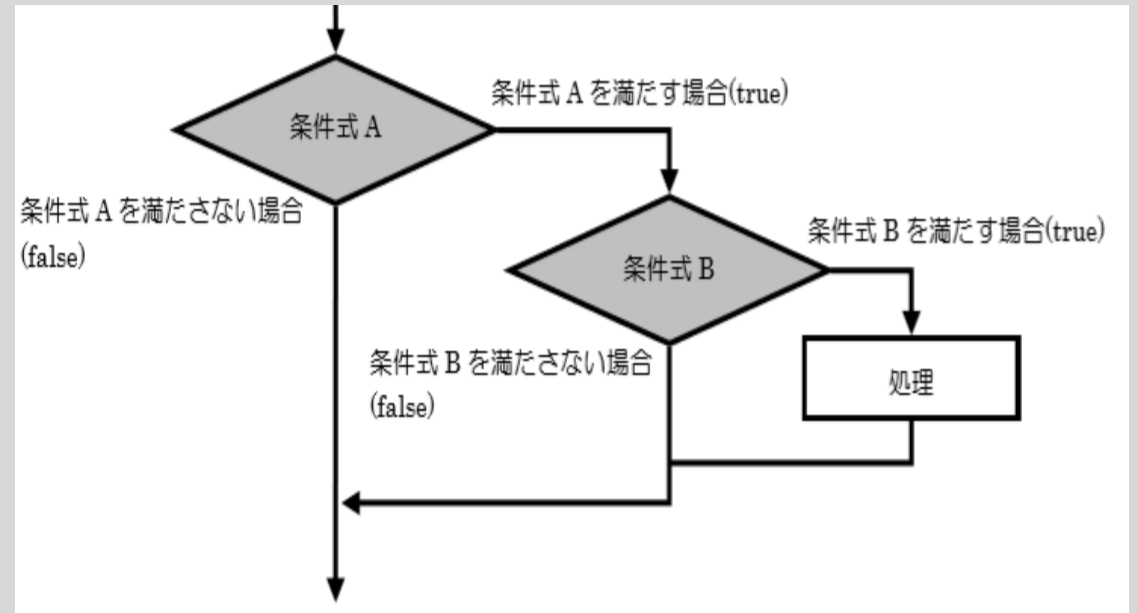
```
if(条件式 1){  
    //条件式1がtrueの場合  
}else if(条件式 2){  
    //条件式1がfalseで  
    //条件式2がtrueの場合  
}else{  
    //条件式が全てfalseの場合  
}
```

# if文～条件による処理の分岐～

## if文による入れ子構造（ネスト）

- if文を入れ子にすると、  
より複雑な分岐が可能に  
→ネストともいう

※ほかの制御構文でもネストは可能だが、可読性も考慮すべき…  
→“コードの可読性”の章で詳述



# switch文

- 変数の値によって多岐分岐できる便利な構文switch文  
→ 同値演算子(==)による多岐分岐
- 以下の手順で処理が分岐
  1. 先頭の式を評価
  2. 上から同値演算を行い、一致する  
case句を実行
  3. 2の手順で見つからない場合、  
default句を実行する

```
switch(式){  
    case 値1:  
        //式が値1の場合  
        break;  
    case 値2:  
        //式が値2の場合  
        break;  
    default:  
        //式が当てはまらない場合  
        break;  
}
```



# switch文

## break文の重要性

- break文は**処理の終わり**を表す  
→ breakがないと、**下のcase文**まで  
処理が続行する
- **フォールスルー**  
break文を**わざと**省略し、次のbreakまで  
処理を**貫通**させる

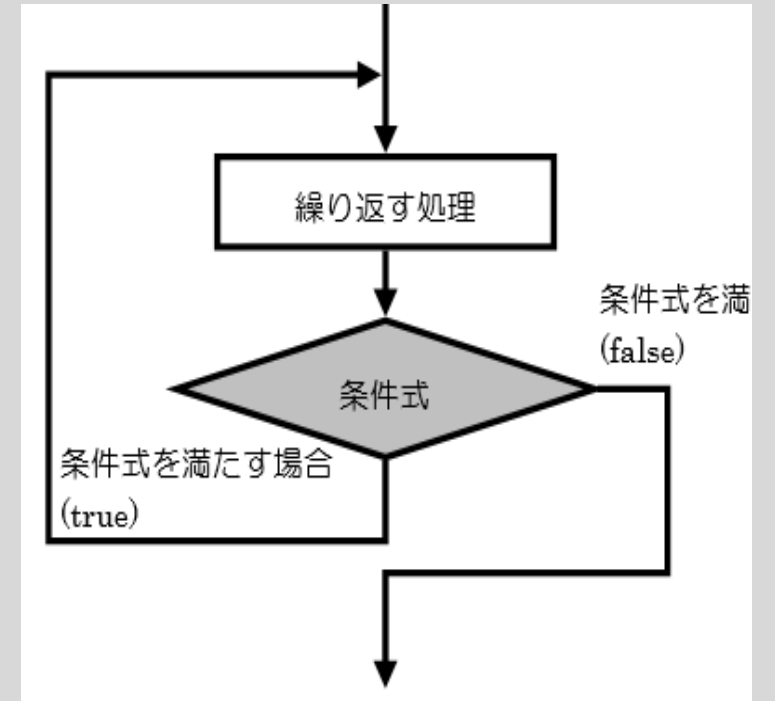
```
var rank = 'B'
var result;
//フォールスルーの例
//ランクによって場合分け
switch(rank){
  case 'A':
  case 'B':
    result = 'success';
    break;
  case 'C':
    result = 'false';
    break;
  default:
    result = '';
    break;
}
```

# 演習 4 “制御構文”①

# 演習 4 “制御構文”① 回答

# while文① do~while文

- 処理の分岐と並び、  
処理の反復をする構文も存在  
→ **while**文や**for**文



- **do~while**構文では、do文で処理を実行  
→while文で式を**判定**し、**繰り返し**実行

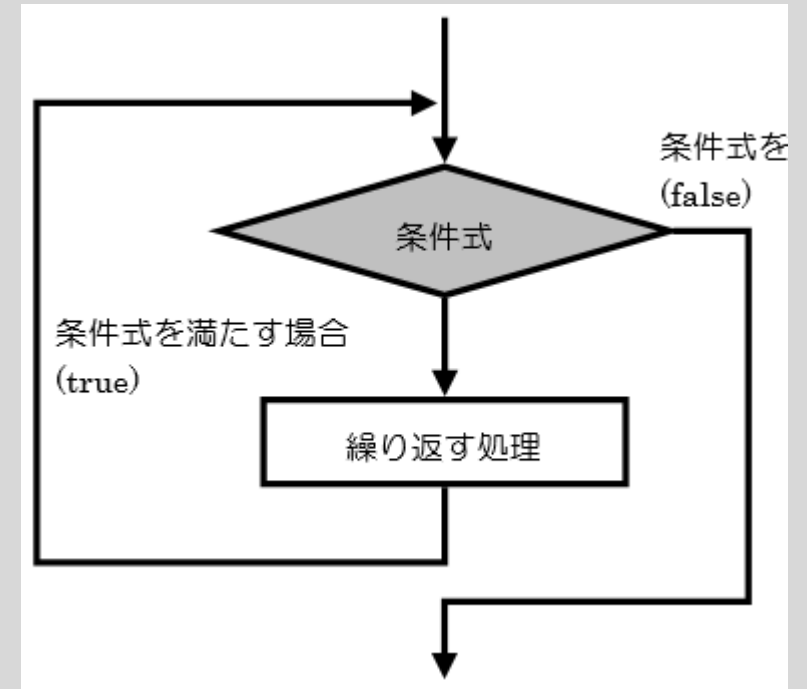
```
do{  
  //条件式が真の間反復  
}while(条件式);
```

## while文② while文

- while構文

do~whileと同じように  
条件式が真の間処理を繰り返す構文

- 条件式を意図的に真にすることで  
無限ループができる  
→マイコンの章で詳述



```
while(条件式){  
    //条件式が真の間反復  
}
```

# 判定の順序

whileとdo~whileでは判定の順序が違う

- 後置判定

ループの最後に条件式を判定する

→do~while

- 前置判定

ループの最初で条件式の判定をする

→whileやfor

## 演習 4 “制御構文”②

## 演習 4 “制御構文”② 回答



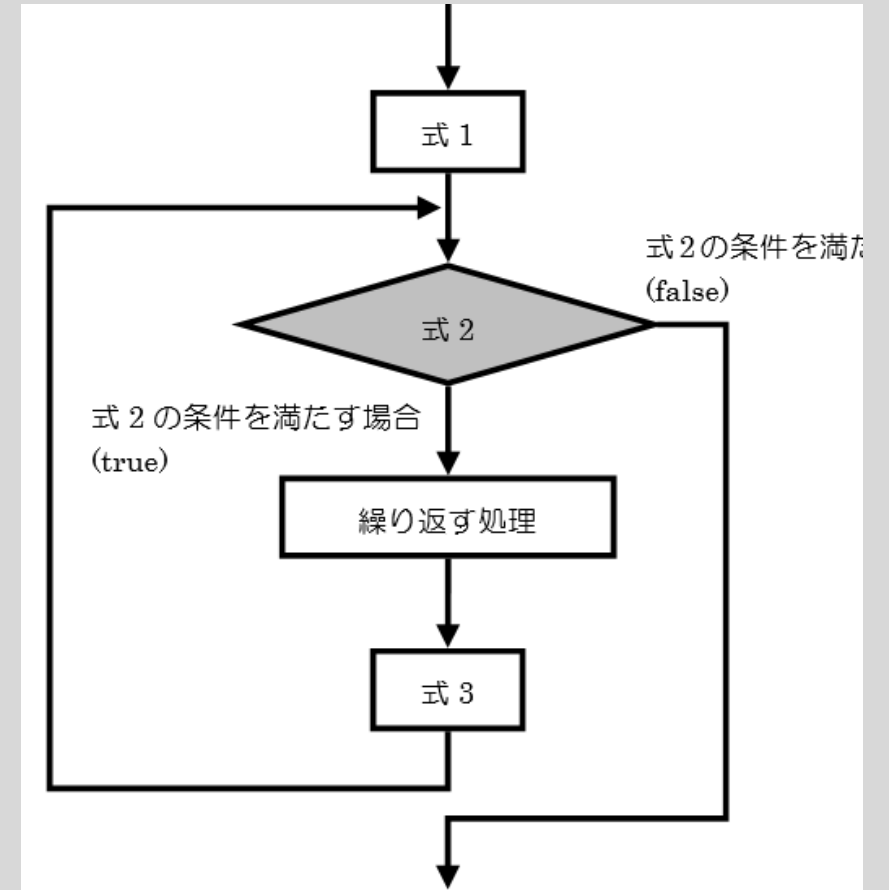
# for文

- for構文

指定された回数処理を繰り返す構文

- 前処理、条件式、後処理の3パートがある

- while文同様無限ループを生成することも可能



# for文

```
for(前処理;条件式;後処理){  
    //条件式が真の間反復  
}
```

- **前処理**（省略可）  
ループに入る直前に行う処理、変数の**初期化**を行うことが多い
- **条件式**（継続条件、省略可）  
ループの継続を**判定**する式。  
省略して**無限ループ**を生成することも可能
- **後処理**（省略可）  
ループを終えるたびに行う処理。変数の**増減**を行うことが多い

# 無限ループの生成

- 無限ループを用いることで継続的な処理が可能  
→マイコンの章で詳述
- **while**文での無限ループ  
カッコ内の条件式を真にする
- **for**文での無限ループ  
カッコ内の条件式を省略する

```
while(1){  
    //無限に処理が続く  
}
```

```
for(前処理;;後処理){  
    //無限に処理が続く  
}
```

# break文とcontinue文

- break文

- continue文

# ラベル構文

for文のネスト ~多重ループ~

## 演習 4 “制御構文”③

# 演習 4 “制御構文”③ 回答



# オブジェクト指向の様々な繰り返し構文

- for~in文

オブジェクトの各要素に対して繰り返し処理を行う構文

- 仮変数に一時的にオブジェクトのキーを格納

- オブジェクトなどはオブジェクトの章で詳述

```
for(仮引数 in オブジェクト){  
    //一つずつキーを取り出し反復  
}
```

# オブジェクト指向の様々な繰り返し構文

- for~of文

**配列**の各要素に対して繰り返し処理を行う構文

- 仮変数に一時的に配列の**要素**が格納される

→for~ofは**キー**に対し、for~inは**要素**

- 配列は**オブジェクト**の章で詳述

```
for(仮引数 in 配列){  
    //一つずつ要素を取り出し反復  
}
```

# オブジェクト指向の様々な繰り返し構文

- foreach文

C#などではfor~inをforeach~inで用いる

- オブジェクトやリスト（配列）に対して各要素ごとに反復処理

- オブジェクトの章で詳述

```
foreach(仮変数 in オブジェクト){  
    //一つずつ要素を取り出して反復  
}
```

# オブジェクト指向の様々な繰り返し構文

- phpのfor~as構文や Visual BasicのFor Each ~ Next構文など  
オブジェクト指向型の言語には様々な繰り返し構文が
- 基本はfor文でカバーできるが、知っておくと便利
- 参考

<https://ja.wikipedia.org/wiki/Foreach%E6%96%87>

try-catch-finally文

# 演習 4 “制御構文”④

# 演習 4 “制御構文”④ 回答

# 制御構文のまとめ





# オブジェクトと関数

～繰り返し処理をひとつに～



# コンピュータの基礎

～マイコンを使ってみよう～

# ノイマンコンピュータの大原則～ 5 大装置～

- 制御・演算装置

プログラムを実行する制御装置、また算術演算を行う制御装置


- メモリ

プログラムやデータを記憶する装置。

- 入力・出力装置


インタープリター方式とは？

コンパイル方式とは？



# マイコンプログラミング

～マイコンを使ってみよう～



# コードの解析と 言語習得の心得

～見て学び、知って使う～

# 自己紹介~Introduction~

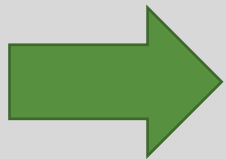


- 奈良高専電気工学科に所属
- 電気系情報発信サイト“JinProduction”の運営者 兼 ライター  
⇒ 現“JinProductionDiary”、来年よりリニューアル
- 電気技術研究会という同好会の部長  
奈良高専祭電気科展の代表者



# 講演目的

- 読む能力を身に付け、自分のプログラムをデバッグする力を身に付ける
- プログラムを書く+読むことで、より良い問題解決能力を得る
- 読む力を駆使し、言語の学習をより効率的に行えるようになる



読むことによって知識を付け、書くことによってそれを活かす力

※別スライド“知識と技術”参照

# 目的達成のために

- **解析**と**可読性**の2パートで読む + 書く力について解説
- **言語の学習法**を最後に組み入れ、理解を深める
- 筆者作成のアプリを用いた**具体的手順**を解説





# プログラミングの解析

～処理を見極める～

# プログラミングを**読む**目的

- **オープンソース**のプロジェクトを使用する機会が増えると…  
様々な場面で、ソースコードの**解析**が必須になる
- 必要最低限の情報を**抜き出し**、応用する**技術**が必要
- 自分の書いたプログラムを**デバッグ**するときなんかも…



# プログラミングを**読む**手順

読む対象の  
**プログラム**を決定

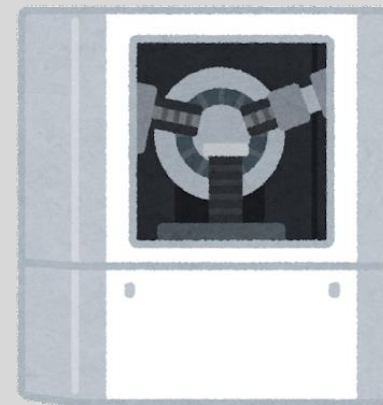


**プログラムの解析**で  
読む**ファイル**を決定



**コードの解析**で  
**改良**や**引き出し**を  
行う

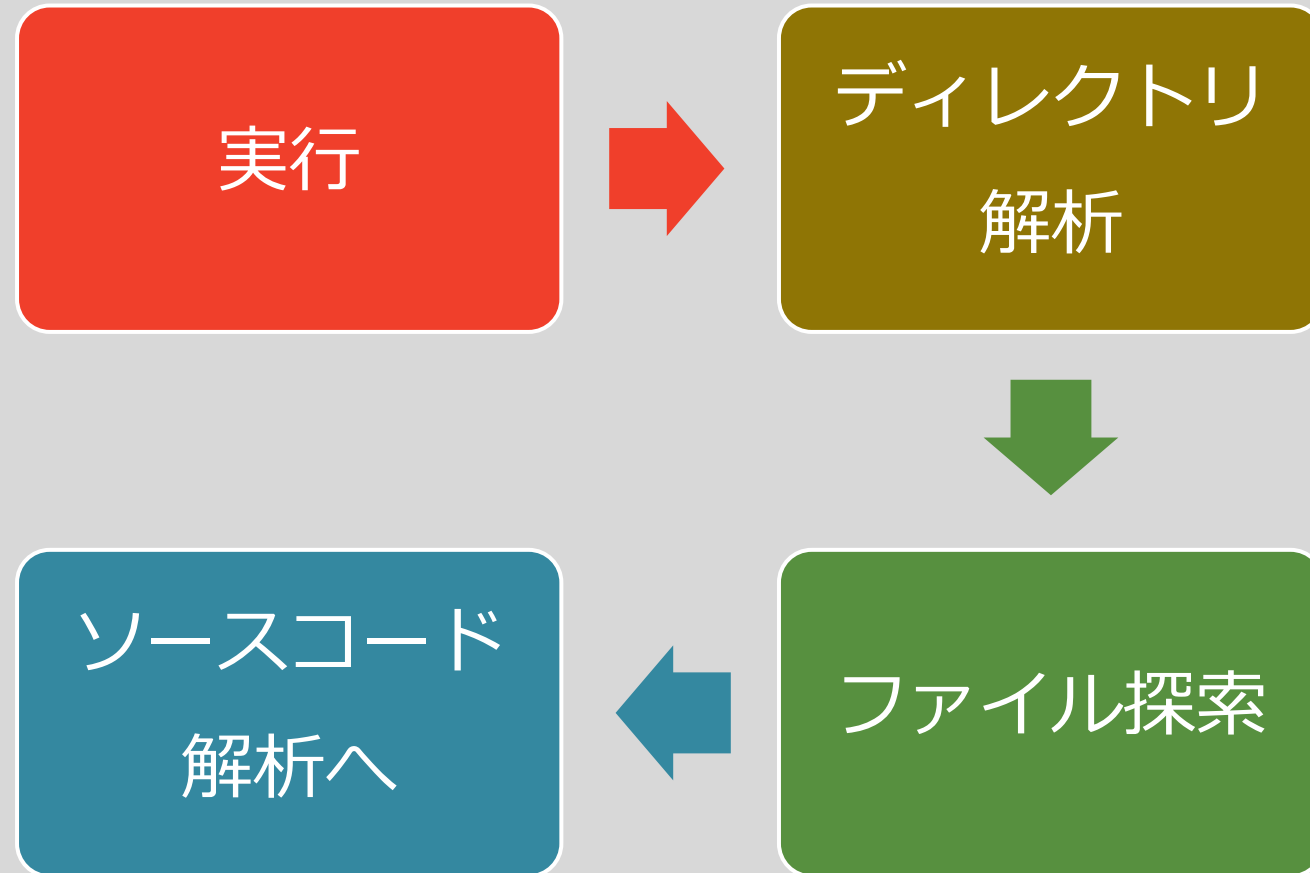
# プログラム解析



- プログラムとは  
ソースコードという言葉ファイルをひとまとめにしたもの
- ソースコードを解析するには、プログラムの構造を知っておく必要が
- ファイル名やディレクトリ名などから、ヒントを得る必要がある

# プログラム解析の手順

## ～目当てのファイルはどこに？～



# ディレクトリ解析



- 大きなプログラムは、多くのディレクトリが存在
- ディレクトリ名から大まかに予測  
例えば…典型的なウェブページではassetsフォルダやsrcフォルダなど
- ディレクトリの構造を読み取り、目的に合わせたファイルの探索に



# ファイル探索



- 解析したディレクトリに入り、ファイルを探っていく
- 拡張子を確認し、ファイルの種類を把握  
例えば…C#は.cs、実行ファイルは.exeなど
- ファイルを見つけたら、まずはデバッグを行う！

# ソースコード解析



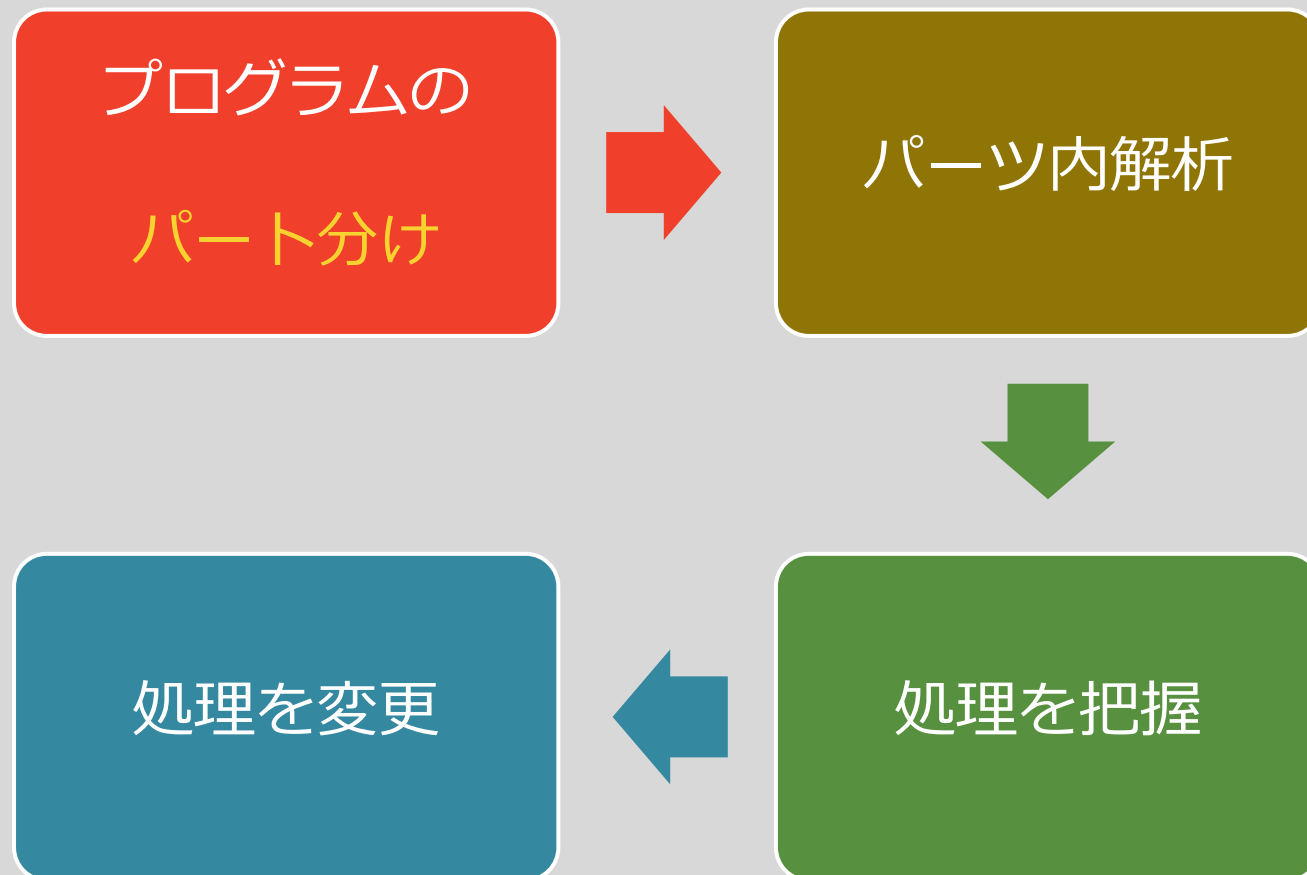
- プログラミングの解析における**醍醐味**
- ソースコードを解析するには、以下の手段が存在
  - **動的解析**  
**デバッグ**を用いる方法であり、変数の値などを随時**確認**できる
  - **静的解析**  
**手動**で読み解く方法であり、変数の値などは完全に**予測**する

# ソースコードの動的解析

## ～デバッグ～

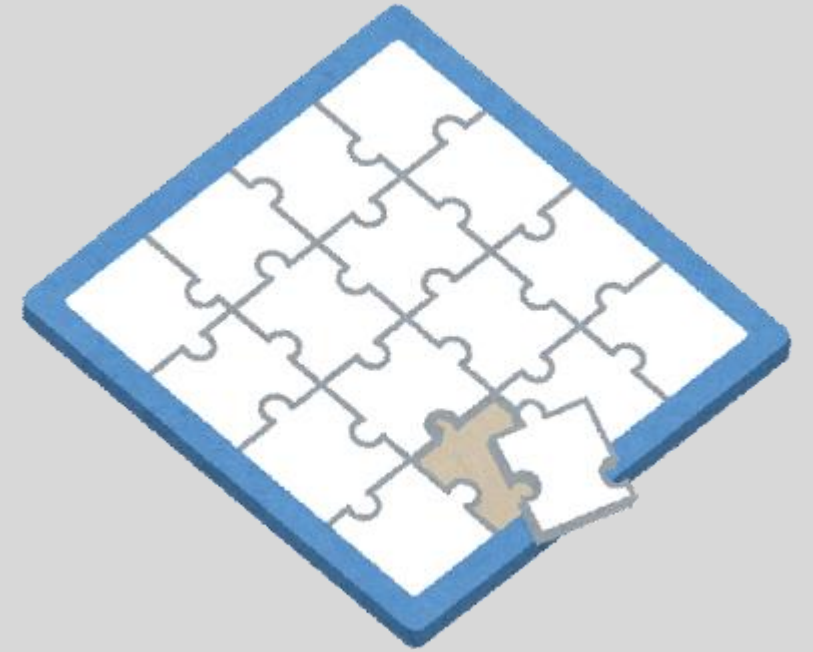
- コードを実行し、**任意の場所**を**観測**することで解析する方法
- デバッガーを使用する方法
  - VisualStudio**などの**デバッグツール**を用いて  
コードを**改変せず**に必要な情報を得る
- デバッガーを使用しない方法
  - printf**や**console**などの**標準関数**を確認する場所に**挿入**する

# ソースコード静的解析の手順



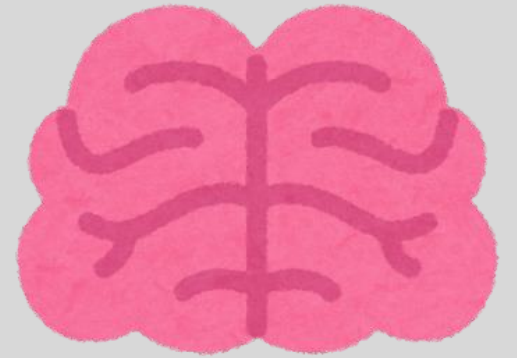
# パート分け

- プログラムとは大きな**処理の流れ**のようなもの
- 膨大なプログラムを読むには？？  
⇒ 書くとき同様、**論理的思考**を活用しよう！！
- ソースコードを解析するうえでも、**論理的思考**は**必須**スキル



# 論理的思考（ロジカルシンキング）とは？

- ものを体系的に整理し、道筋を立てて考える思考法
- 問題解決の際、原因特定や解決策の立案に役に立つ
- ソースコードを読む場合、処理の流れを論理的に読み解く必要がある



# 解析への活用

- ソースコード内の全要素を確認し、道筋を立てて大きな処理の流れを把握  
⇒ コメント文を参考にして予測したり  
必要に応じて動的解析を用いて確認する
- 処理の流れを確認したら  
パズルのように処理をまとめてパート分け
- 分けたパートごとに細かく解析を行う

# パーツ内解析

- 。分けたパーツを確認し、任意のパーツを絞り出す

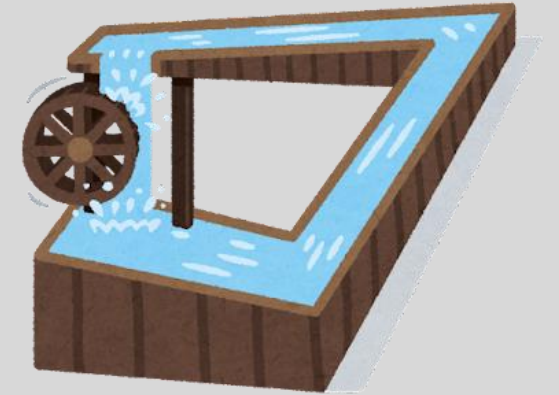


- 。絞り出したパーツで、変数や関数の役割を予測したり  
動的な解析を用いてひとつずつ動作を確認したりする
- 。一つ一つの変数や関数を、パーツの中のパーツとみる



# 処理の流れを把握

- 1つ目の手順でしたように  
次は絞ったパーツで小さな処理の流れを把握
- 確認した役割のパーツをつなぎ合わせ、パズルを完成させる
- コメント文を参考にして予測したり  
必要に応じて動的解析を行ったりして動作を確認する



# 処理改良の手順



。処理の改良にも、論理的思考を用いた手順が存在

## 1. 原因解明

区切ったパーツで、変更すべき構文や変数、処理を確認する

## 2. 解決策の立案

どう変更するか、また新しくどのような処理を挿入するか、考案する

# プログラミング解析のまとめ



- パーツ分け、パーツ内解析、処理の把握、改変を繰り返し、目的の動作へ近づけていく
- 論理的思考を用いた道筋に基づき処理の流れを理解する
- 動作や処理の予測と、実際に動かしたときの確認がキーポイント
- 大きなソースコードを、小さく区切って考えていく



# コードの可読性

～読みやすいコードのために～

# より良い可読性を求めて

- プログラミングを読みやすくするために**可読性**という概念が存在
- 可読性は大まかに以下の2つに分かれる
  - ソースコード内の**記述**
  - 論理的思考を用いた**処理の流れ**



# 命名法



- 変数や関数を適切に命名すると、誰にでも読みやすくなる
- 変数名は最初を小文字、関数名は大文字にする…  
など、ルールを決めて記述しているとより分かりやすく
- 名前自体は変数の役割を、より分かりやすく命名する必要がある  
例えば…カウントする変数：`cnt`、一時的な変数：`tmp`  
以前の値を格納：`pre~`、など…

# 演算子の選択

- 演算において、**右辺に変数**が来る場合  
**複合代入演算子**を用いると、よりコードが見やすくなる  
⇒演算の章を参照
- **if構文**を使わなくても、**3項演算子**で解決する場合  
積極的に用いると、**簡潔なコード**で見やすくなる
- **ifネスト**を避けるときには、**論理演算子**を用いると簡潔に

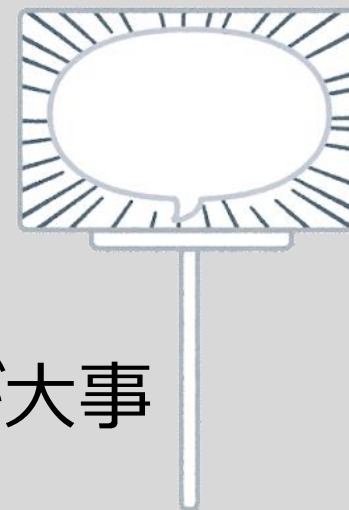


# インデント

- プログラムにおいて、**字下げ**（**インデント**）は非常に重要な要素
- 基本的には、**中括弧**は**4字**or**2字**下げる  
**ネスト**が深くなると、字下げが多く**読みにくく**なる場合もある
- インデントで関数の**区切り**などを決めている言語もある  
(Pythonなど)



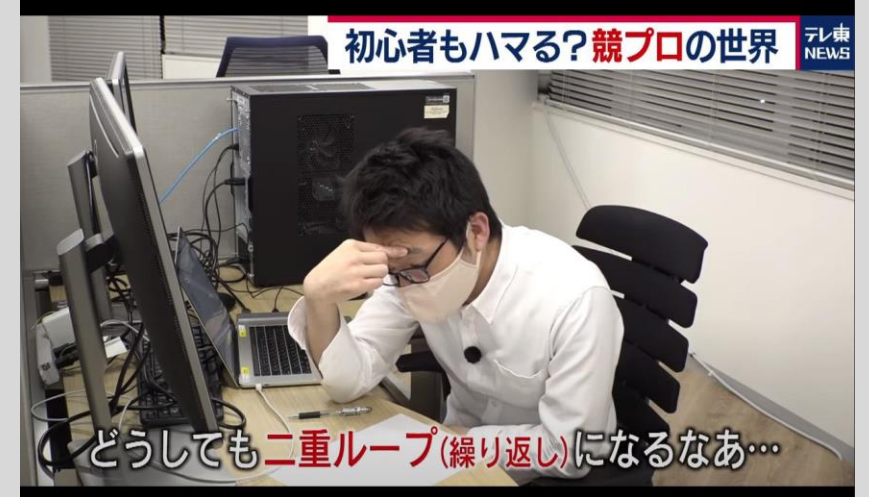
# コメント文



- プログラムを書くと、随時その処理の**役割**を  
**コメント**で**注釈**することが大事
- コメントは**完結**に要点だけを書く、**抽象的**なもので可  
**具体的**に書きすぎると長くなり、かえって可読性が**悪く**なってしまう…
- 注釈がないと、**可読性**だけではなく、自分でも**忘れてしまう**ことが…

# ネストの回避

- ネストはなぜ使う？  
ifのネストでは複雑な条件分岐が可能  
多重ループでは多次元配列の処理が楽に
- ネストの問題  
複雑すぎると、記述と処理両方の可読性が悪くなる…
- 解決策  
ifネストでは条件文に論理演算子を用いたり、  
多重ループを回避するには、配列の次元を減らすなど



# 関数やクラス分け



処理をメインルーチンに詰め込むと…

どこからどこまでが、何の処理の範囲か不明になる

そこで…

⇒適切に関数やクラスを分けることが重要に

関数に処理をまとめると、ほかの関数からも参照が可能になる

⇒構造化プログラミングの反復処理にあたり、可読性が改善

# 配列やオブジェクト



同じ系統の変数を大量に作成すると…

多すぎる変数に対し、処理の役割が見えにくい！

そこで…

処理を関数にまとめるように

変数も配列やオブジェクトにまとめることが重要！！

# ファイル分け



ファイル名と**全く別の処理**が記述されていると…

**処理**だけではなく、**ファイルの解析**も困難になる

そこで…

**別ファイル**や**ライブラリの自作**によって

**処理の混同**を避けることを意識しよう！！

# ライブラリの多用

「ちょっとねえ、僕はちょっと怒ってます」おぢさんは間違っている！！

複雑な処理を頑張って自作して追加しても…

重要な処理に注目できず、手間や時間が取られる  
そこで…

先人に感謝し、ライブラリを使わせてもらおう！！！！

⇒ライブラリ自体の理解は最低限必要



# コードの可読性のまとめ



- コードを読みやすくするためには、**可読性**を意識する必要がある
- 記述**の可読性
  - 命名**や**演算子**、**インデント**、**コメント**、**ネスト**などに  
注意して記述する必要がある
- 処理**の可読性
  - 関数**や**クラス**、**配列**、**オブジェクト**、**ファイル分け**や**ライブラリ**などに  
注意して記述する必要がある



# 言語の学習法

～いろんなところに適応しよう～



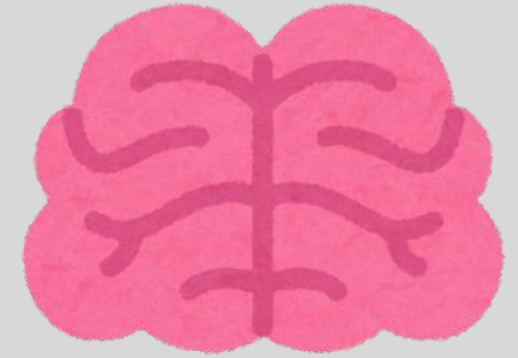
# 言語習得の手順

新しく言語を学ぶときの手順



1. 記述方法（インデントの決まりや文末のセミコロンなど）を知る
2. 基本の骨格（制御構文やデータ型など）について調べ、理解する
3. 既存のコードを解析したり、独自でコードを記述することにより習得

# 論理的思考と問題解決能力



- 言語を学ぶときに最も重要なのは…  
論理的思考である
- 言語の文法を身に付けるのは非常に簡単だが…  
初めての言語では、慣れないエラーにてこずる可能性も

# 論理的思考と問題解決能力

- エラーに出くわした時、**論理的思考**を最大限に活用しよう
- **対象のプログラム**に対し、**プログラミング解析**をかける
- 解析した結果、発見した**問題**に対し、**解決策**を立案
- **可読性**を意識したうえで、コードを再び**記述**、**デバッグ**の繰り返し

# 論理的思考と問題解決能力

- **エラー対処**だけではなく…

ものづくりをする際に非常に**重要**になるのは**エラー処理**… ???

⇒ **間違い**

- もしエラー無く**コードが実現**しても

**本質の課題**を解決できてなければ意味がない！！

つまり…

**課題の本質理解**と、それに沿った**真の課題解決**が必要！

# 言語を習得するには

基本の知識を身に付けたうえで、プログラミング解析を繰り返す  
⇒動的なデバッグで処理の確認をすることで  
だんだんと予測が可能になる



課題解決の予測を立て、コーディングして確認して言語習得が完成

これは独学による、課題解決能力を駆使した  
最も効率的な言語習得法であり、本質理解につながる

# 言語の習得法のまとめ



- 記述方法、基本の骨格を知り  
それを活かしてプログラミング解析を重ねて言語習得をする
- モノづくりではエラーを対処するだけでなく  
課題の本質理解から真の問題解決を実現することが重要
- 問題解決を繰り返すことで予測と確認の手順が身に付き  
最も効率の良い言語習得法が完成する



# 最後に

～全体のまとめ～

# プログラミングを読むこと



- **プログラミング解析**の手順  
プログラムの解析⇒ソースコードの解析
- **読み、処理の流れ**を理解することで**改良**し、**問題解決**につなげる  
⇒**論理的思考**の最大活用によって**効率**が上がる



# プログラミングを書くこと

- コードを書くとき、**可読性を意識した**コーディングが重要に
- 記述**の面では**見やすい**コードを、  
**処理**の面では**読みやすい**コードを心がける



# プログラミングを学ぶこと

- 基本的な知識を活用し、プログラミング解析を用いて学習する
- エラーの対処だけでなく、問題自体の解決を心がける
- 予測と確認の手順で、言語を本質から身に付けよう！！



# 問題解決能力を駆使したものづくり

- 読んで学び、知って使う手順はプログラミングだけか…

⇒全くの誤解です！



- 電気や数学などの理数分野や  
環境問題や政治問題などの社会など  
広くにわたって応用が可能！

- ぜひ、論理的思考を身に付け、課題解決に取り組んでください！