

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～

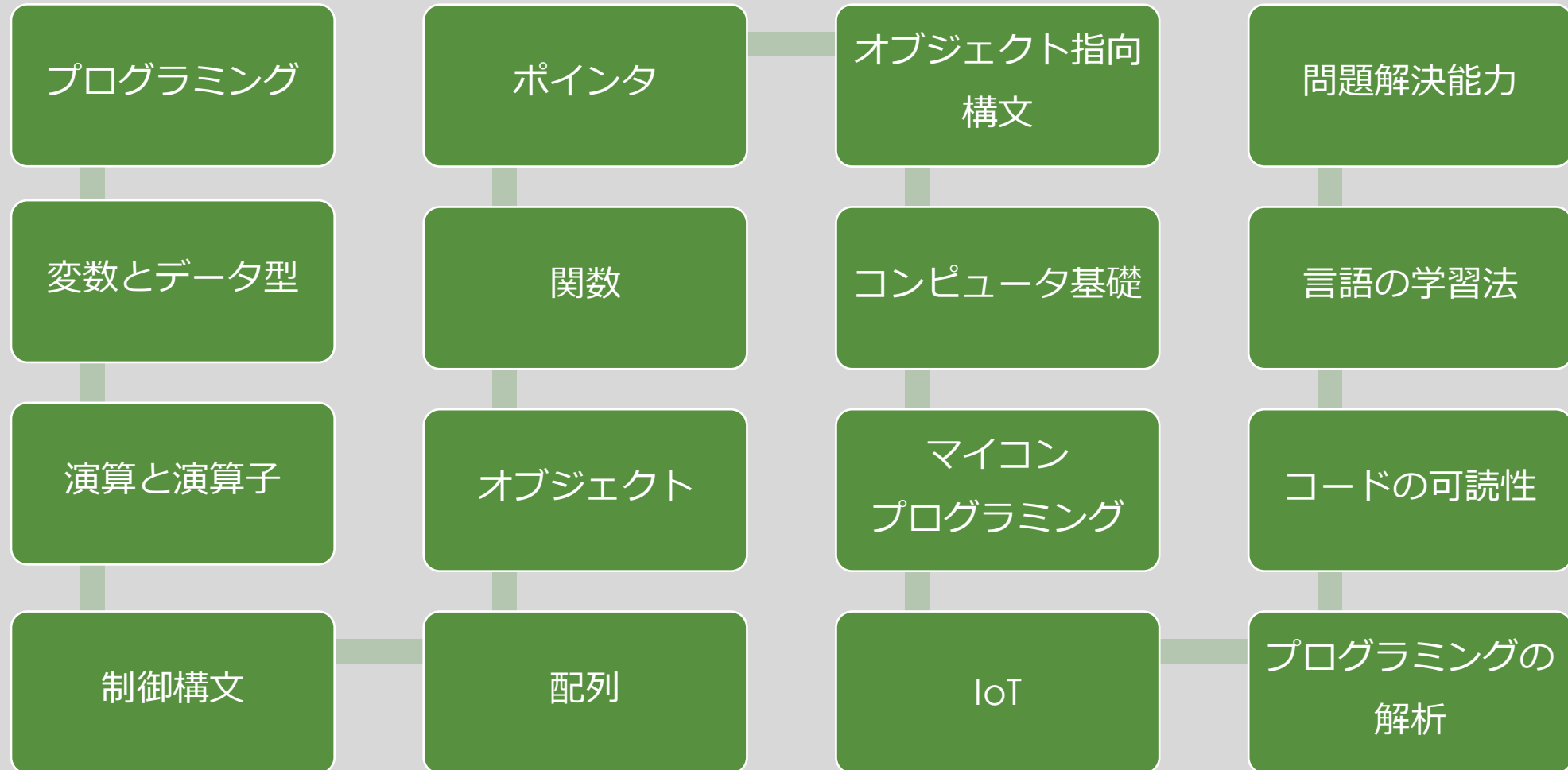
講義目的

- 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

カリキュラム



進め方

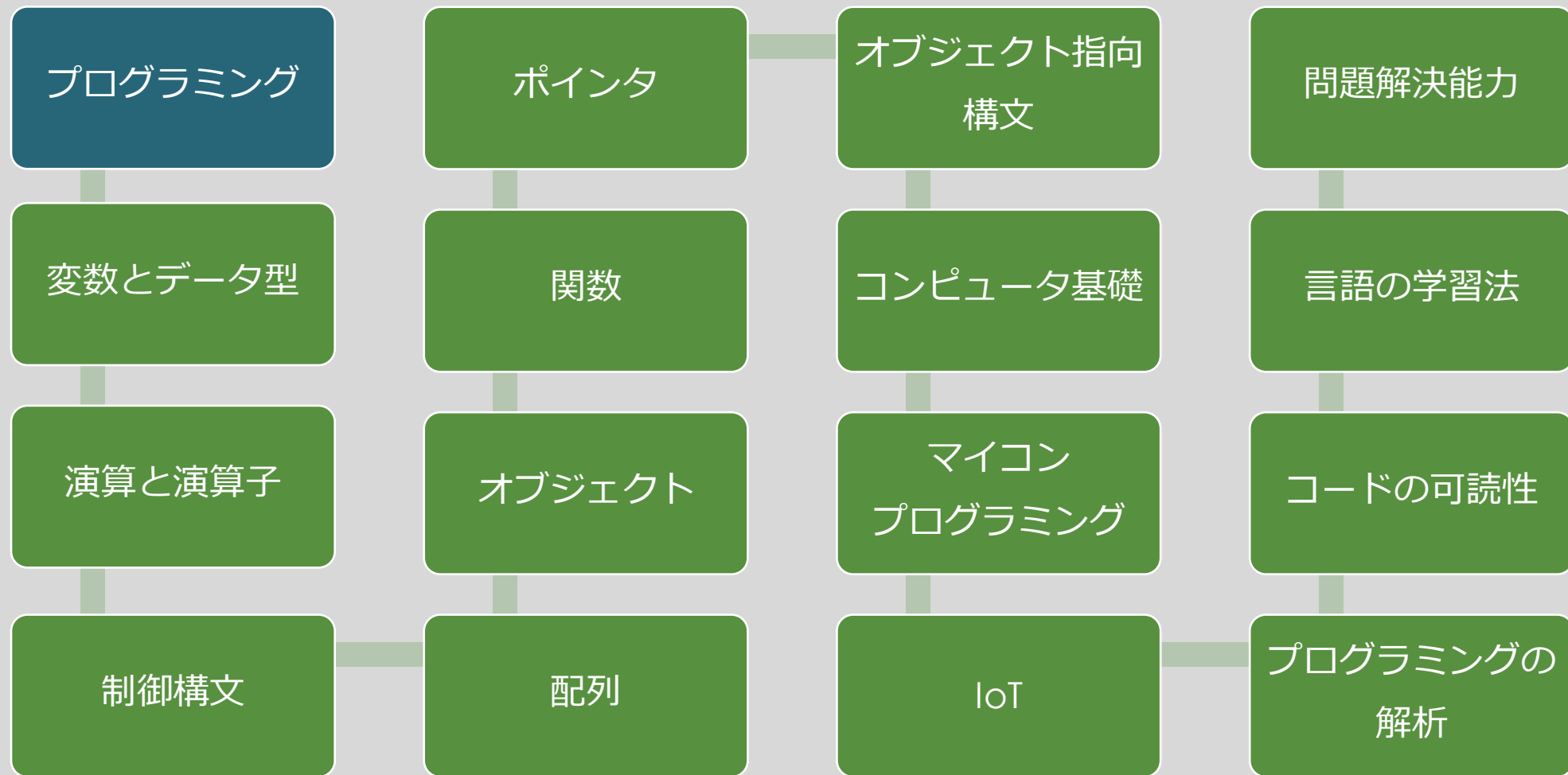
- プログラミングだけではなく、**設計**等の講義も行う
- 著者作成の**アプリ**を用いた説明もある
- かなり分かりやすく**簡略化**しています。頑張ってください^^



プログラミングとは

～基本のキ～

進度



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);
}

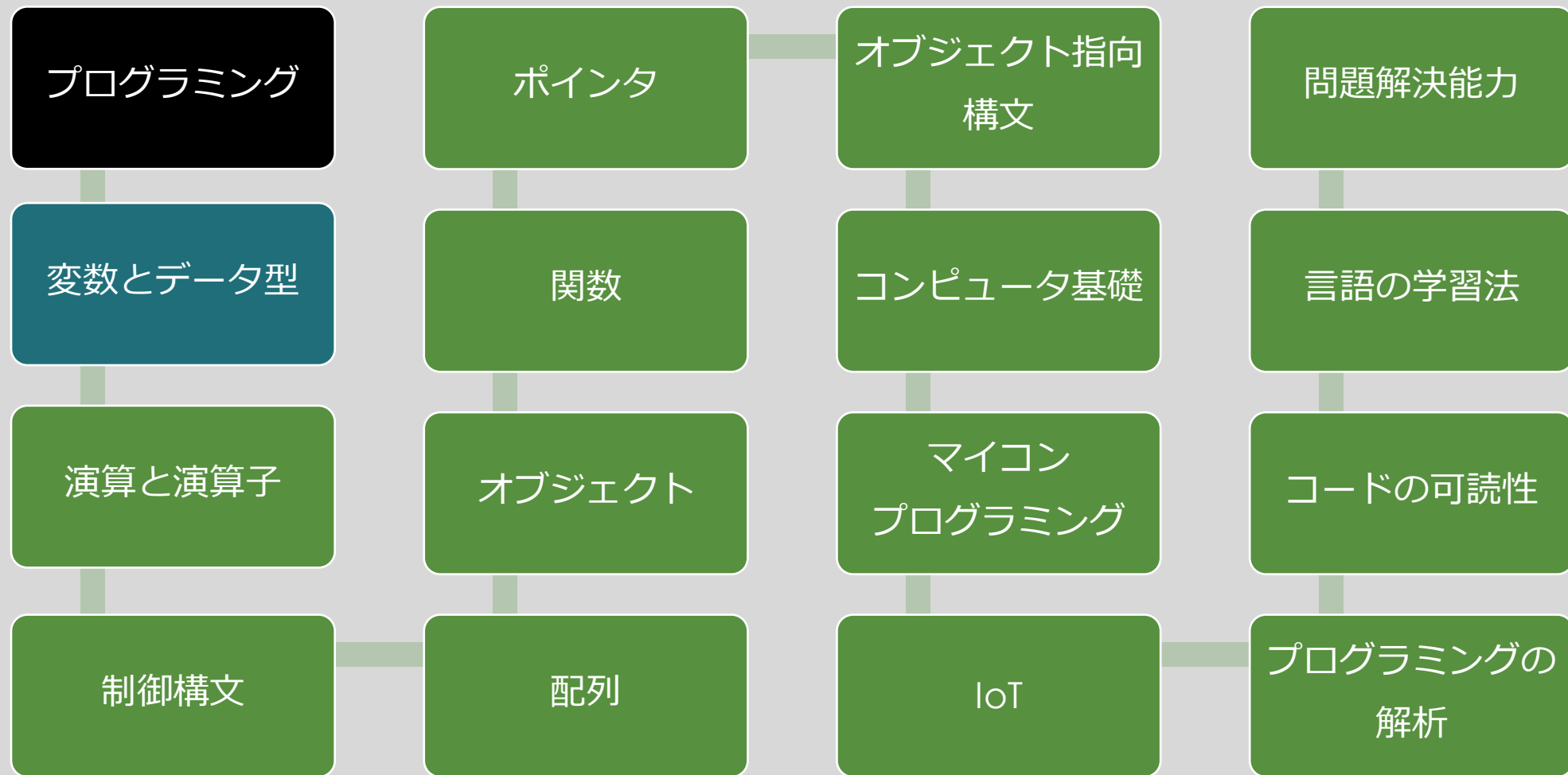
void loop(){
    digitalWrite(2, HIGH);
    delay(200);
    digitalWrite(2, LOW);
    delay(200);
}
```



変数とデータ型

～プログラミングで使用する箱～

進度

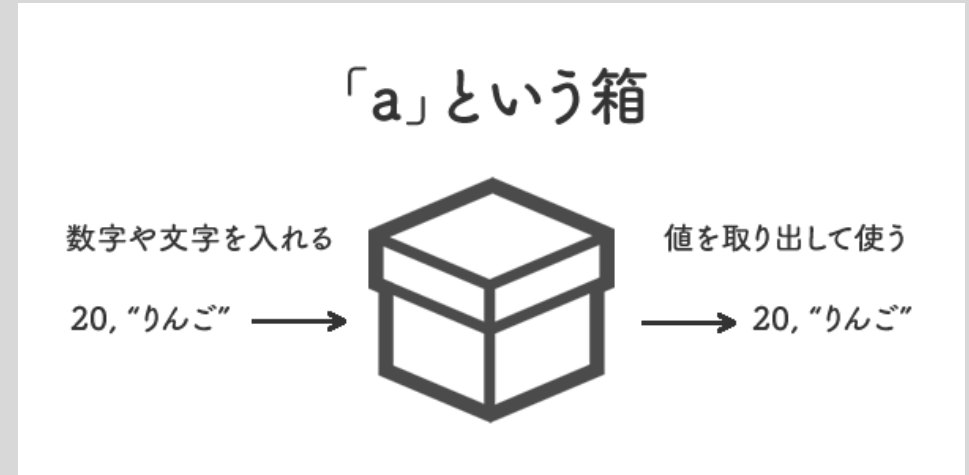


変数

- 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

```
let 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

```
let 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

```
let 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にて）

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ならない
  予め決める言語を静的型付け言語という
 - 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？～動的型付け～

- **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
 自動で割り当てられる言語を**動的型付け**言語という
- varは**宣言の重複**が可(上書きされる)、letは不可
- var型は後述の**スコープ**が特殊なので基本は**let**を使用

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

※詳細は関数の章で説明

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

...

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**オブジェクト指向**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

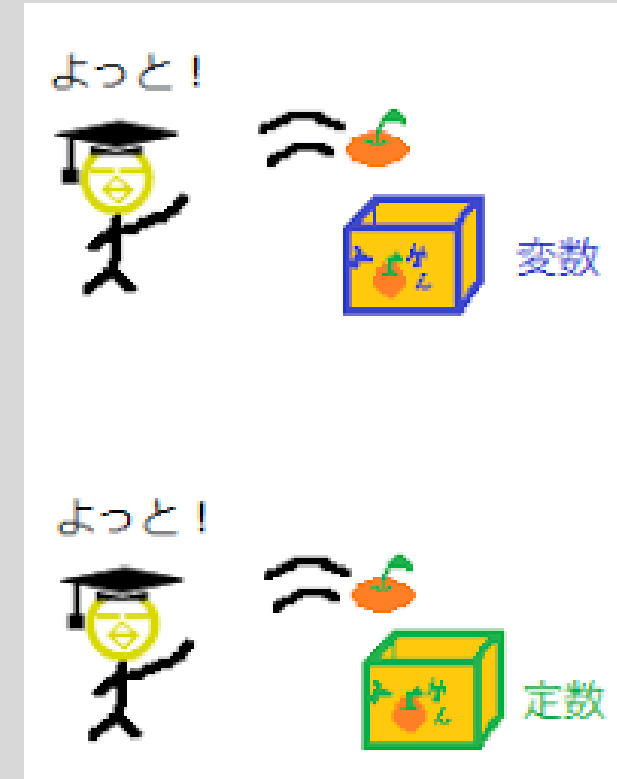
※言語によっては若干のばらつきがあるので注意が必要

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`
- 静的型付けでは、`const 型名 定数名 = 値`
とすることが多い
- 定数は変更（再代入）することができない



ここまでのまとめ

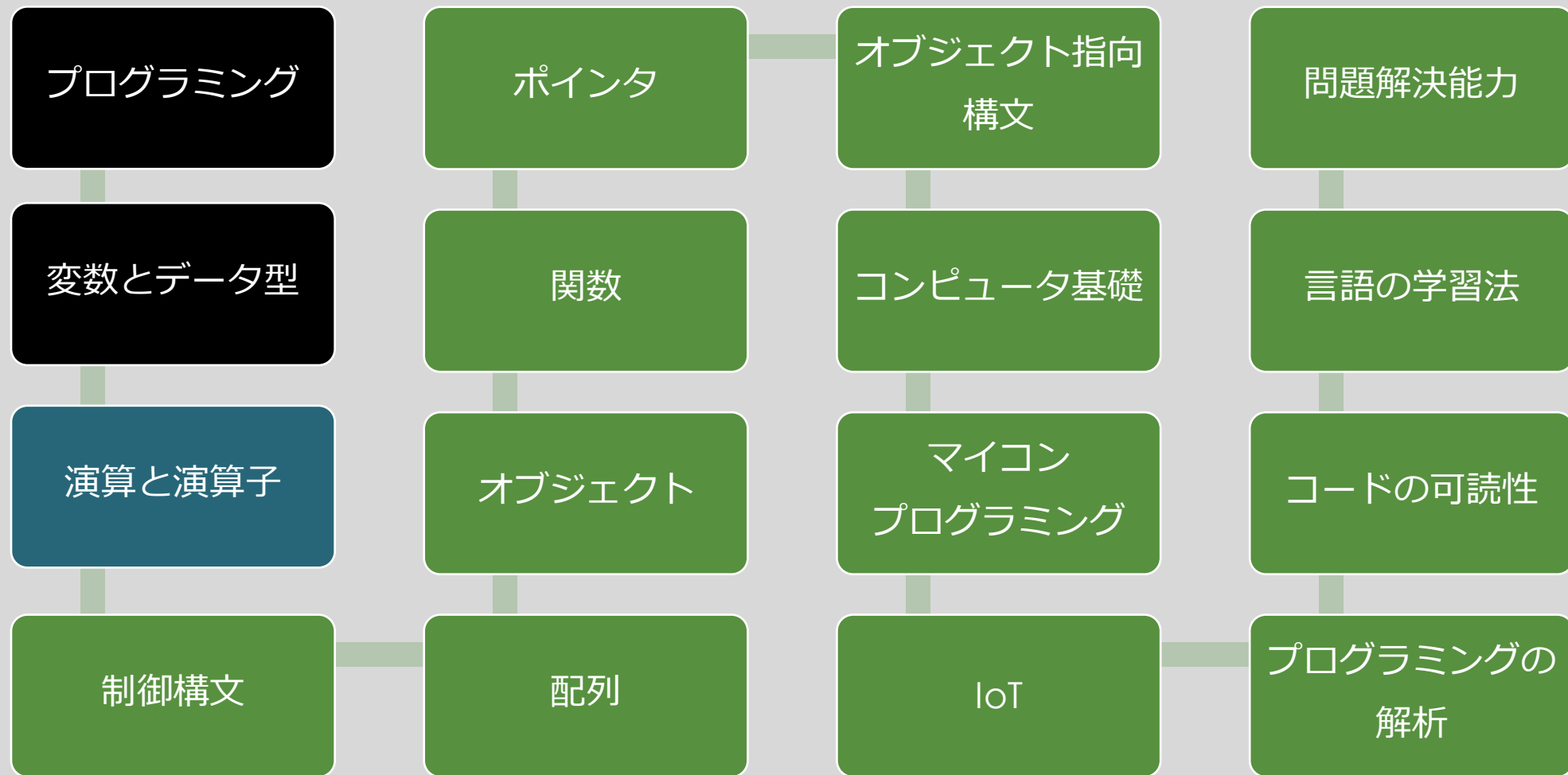
- プログラミングは**基本的な骨格**がある
- 変数はデータを入れる箱であり、**宣言**、**初期化**、**代入**などで操作する
- 変数は**スコープ**や**データ型**を確認する必要がある



演算と演算子

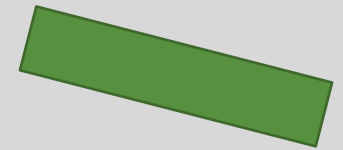
～プログラムの演算～

進度



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

```
let x = 5;  
let 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;
?>
```

```
let str1 = 'Hello';
let str2 = 'World';
let 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

```
let 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

```
let x = 2;  
let 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

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

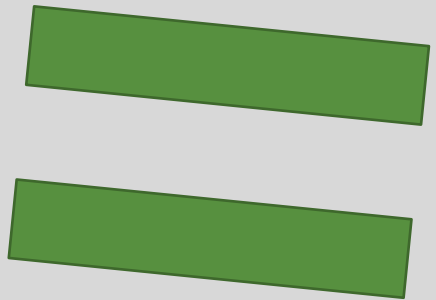
単項算術演算子

- 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>&=</code>	論理積を計算し代入	<code>x &= 4 (x = x & 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><<=</code>	左にシフト演算し代入	<code>x <<= 2 (x = x << 2)</code>
<code>>>=</code>	右にシフト演算し代入	<code>x >>= 4 (x = x >> 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

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

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

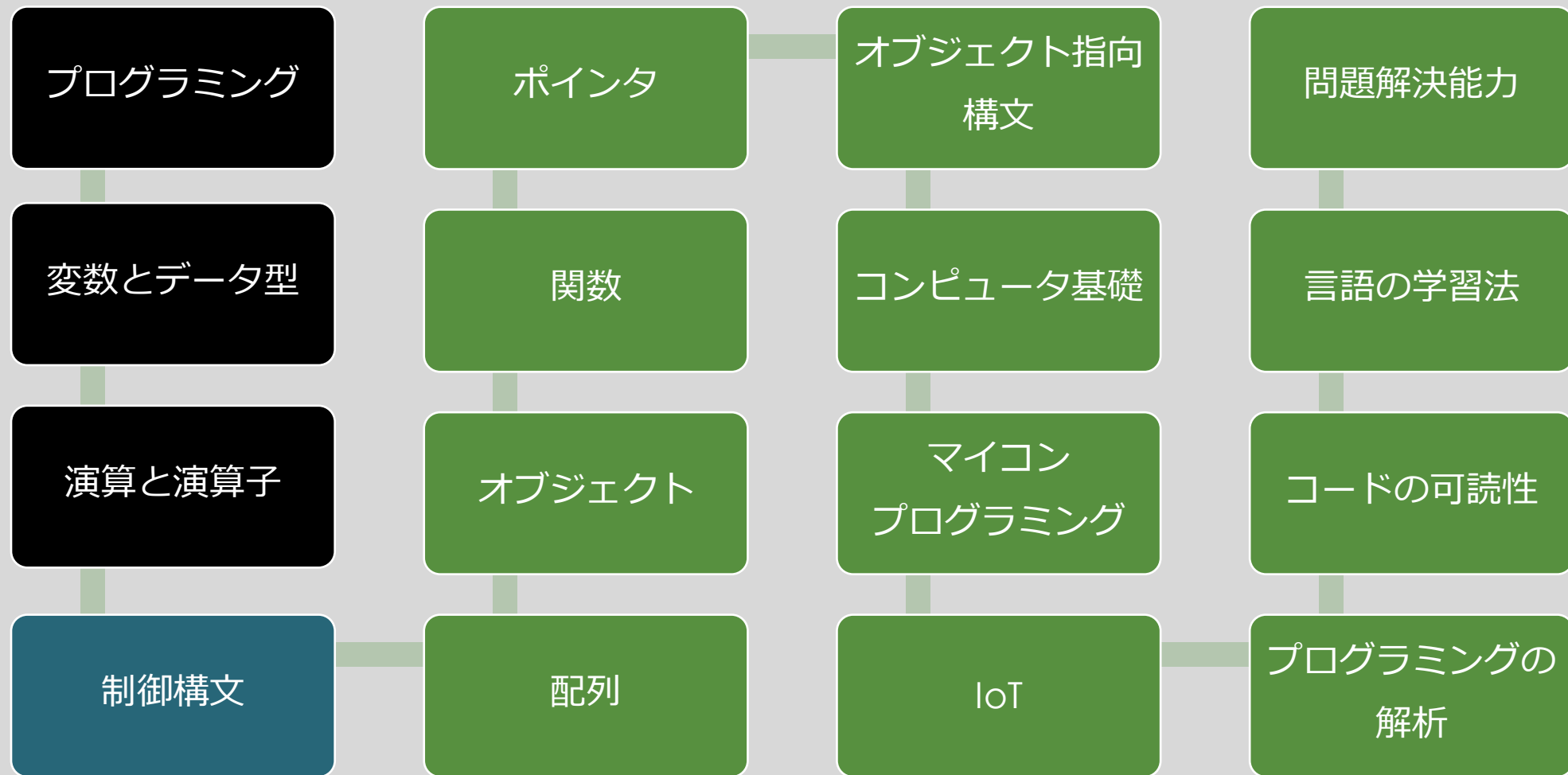
- 演算子には算術やビット、論理などさまざまな種類がある
- 代入演算子には単純と複合がある
- 演算子には優先順位や結合性があり、複雑な計算をするときに必要
- 変数は文字列に埋め込める



制御構文

～条件分岐と繰り返し～

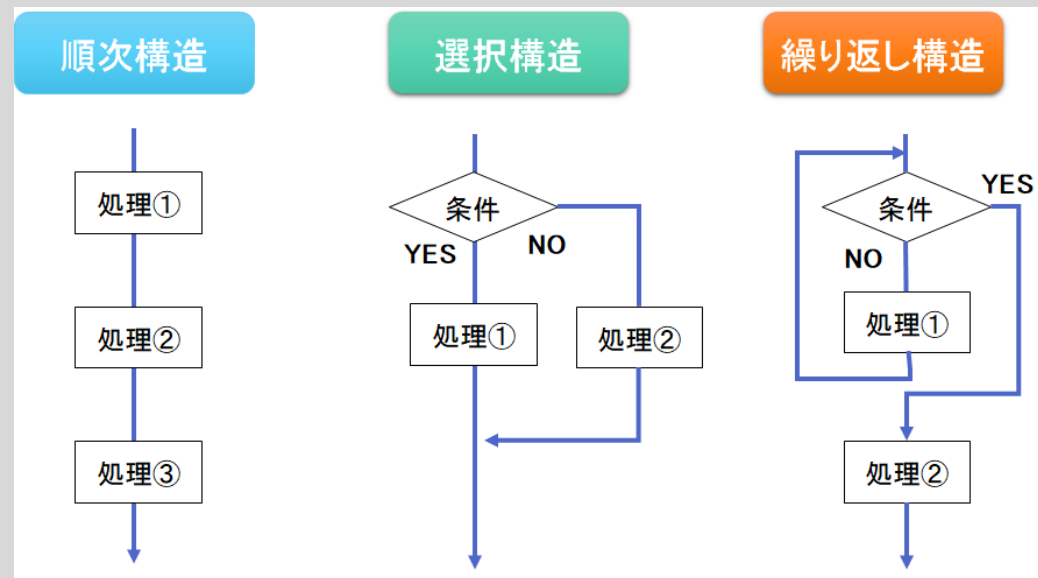
進度



制御構文

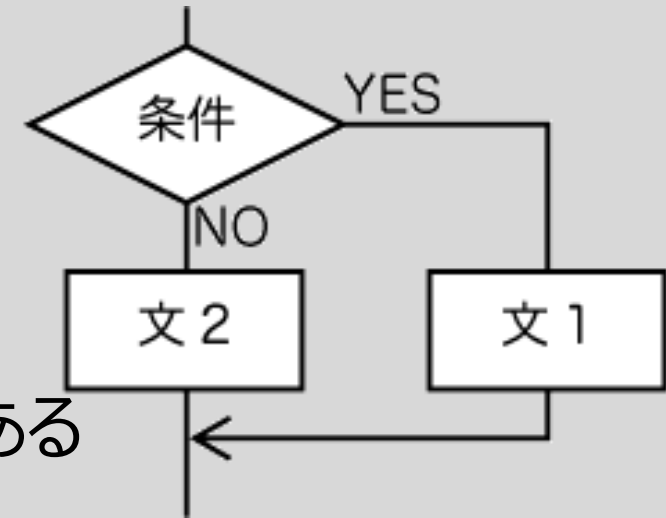
構造化プログラミングの手法

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  
}
```

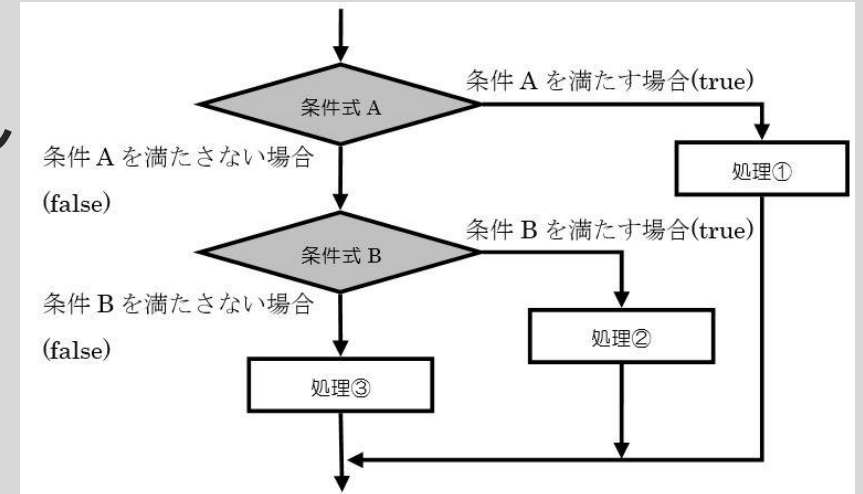
if文～条件による処理の分岐～

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



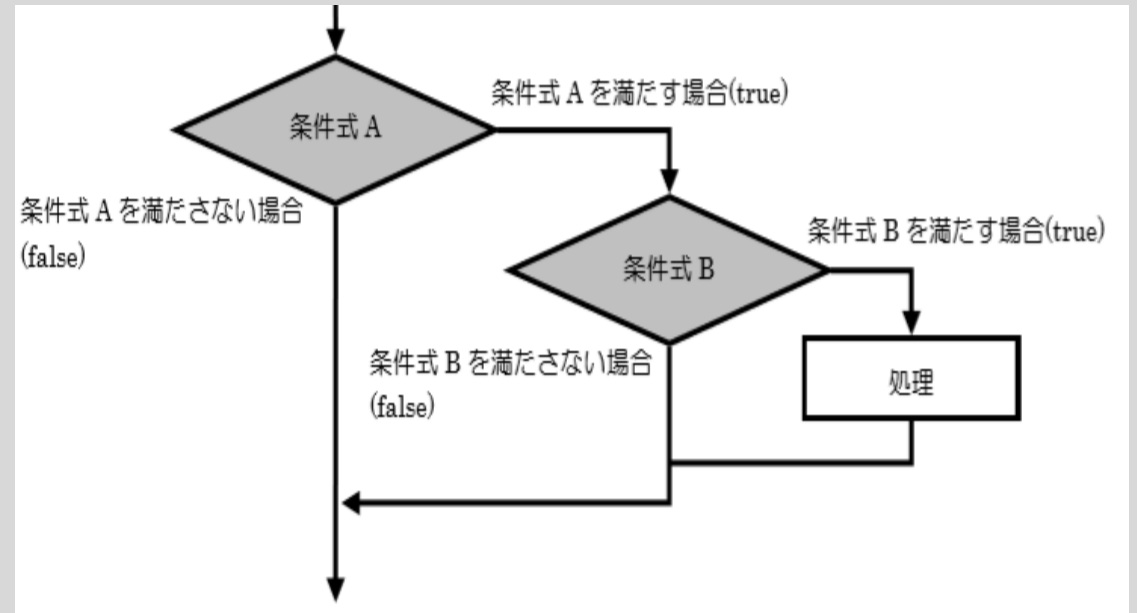
```
if(条件式 1){  
    //条件式1がtrueの場合  
}else if(条件式 2){  
    //条件式1がfalseで  
    //条件式2がtrueの場合  
}else{  
    //条件式が全てfalseの場合  
}
```

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

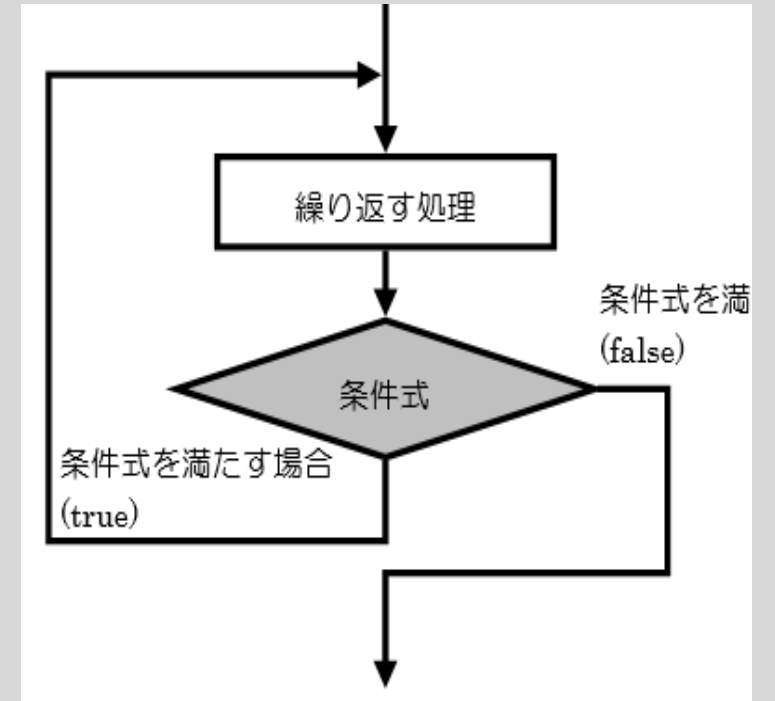
break文の重要性

- 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

```
let rank = 'B'
let 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;
}
```


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

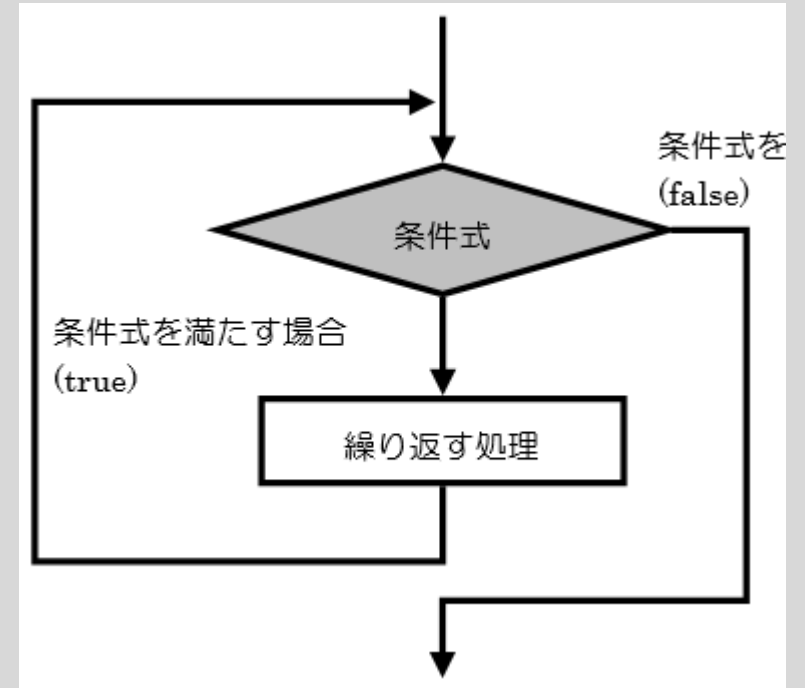
```
do{  
  //条件式が真の間反復  
}while(条件式);
```

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

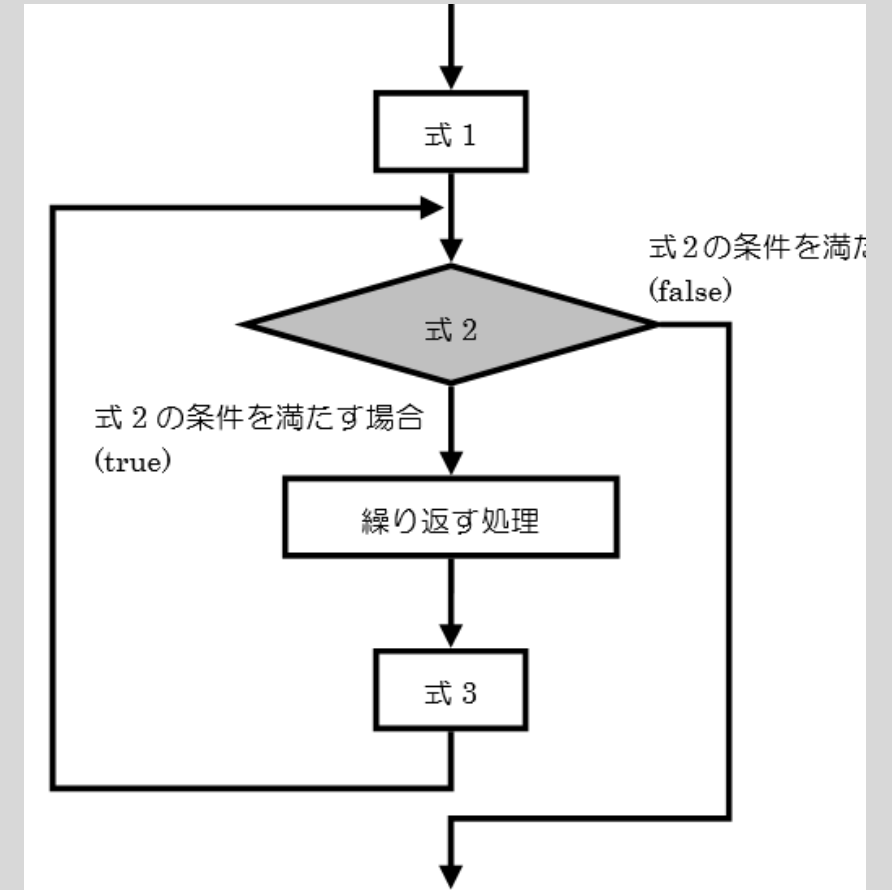
for文

- 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

forループを**強制的**に脱出することができる

if文と組み合わせて条件で抜けることが多い

```
for(let i = 0; i < 5; i++){  
    if(i > 3){  
        break;  
    }  
}
```

- continue文

forループを1度**スキップ**して次のループに移ることができる

breakはfor文**すべての**処理をスキップするのに対し

continueは**一度だけ**スキップする

```
for(let i = 0; i < 5; i++){  
    if(i < 2){  
        continue;  
    }  
}
```

ラベル構文

```
label :  
for(let i = 0; i < 5; i++){  
    if(i > 2){  
        break label;  
    }  
}
```

- ラベル名 :

とすることでプログラムの任意の行に移行するラベルを付けられる

- ラベル付きbreak(continue)文⇐Cはない…

break ラベル名として任意の行へループから抜けることが可能

- goto構文⇐Cの代替

go to ラベル名をラベル付きbreak文の代わりとして使用できるが…

➡ 非推奨

for文のネスト ~多重ループ~

```
for(let i = 0; i < 5; i++){  
  for(let j = 0; j < 3; j++){  
    console.log(i * j);  
  }  
}
```

- 行列的な処理をするために多重ループというものが存在

for文を**入れ子**構造にして中のforループから順に処理

- 右上の例では…

$0 \times 0 \Rightarrow 0 \times 1 \Rightarrow 0 \times 2 \Rightarrow 1 \times 0 \Rightarrow 1 \times 1 \Rightarrow 1 \times 2 \Rightarrow 2 \times 0 \dots$

のように**j**からインクリメントされ、

iの条件が満たされるまで繰り返しになる

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**

```
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

```
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

```
try{  
    //処理  
}catch(/*変数に例外を受け取る*/){  
    //例外処理  
}finally{  
    //最終処理  
}
```

- 開発時に想定外のエラーに遭遇することも

➡ 例外処理が有効

- tryで処理

catchでtryで発生した**例外の処理**

finallyで例外に関わらず**最終的**に実行される処理

- 例外のデバッグにも使用できる

制御構文のまとめ

- 制御構文は構造化プログラミングの基本
- 順次、選択、反復の3つの種類がある
- 選択・・・if文やswitch文
- 反復・・・for文やwhile文

参考文献

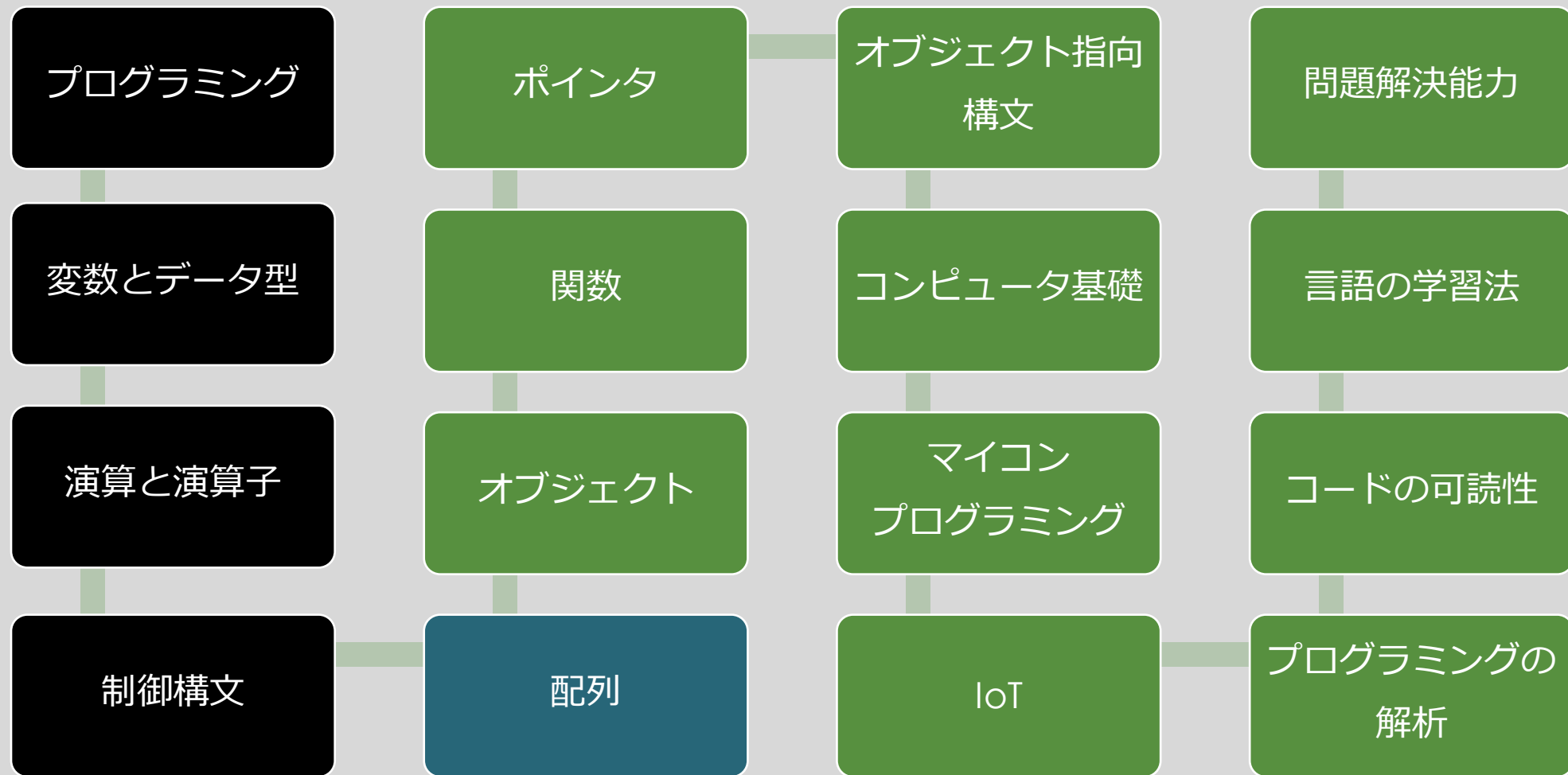
- <https://eng-entrance.com/linux-shellscript-variable>
- <https://xtech.nikkei.com/it/atcl/column/14/091700069/091700002/>
- <https://wa3.i-3-i.info/diff446programming.html>
- <http://www.b.s.osakafu-u.ac.jp/~hezoe/pro/chapter5.html>
- https://kanda-it-school-kensyu.com/php-super-intro-contents/psi_ch09/psi_0905/
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch06/jbi_0606/
- <https://itmanabi.com/structured-objectoriented-prog/>
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch07/jbi_0704/
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch07/jbi_0703/
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch07/jbi_0702/



配列

～変数をまとめる～

進度



配列

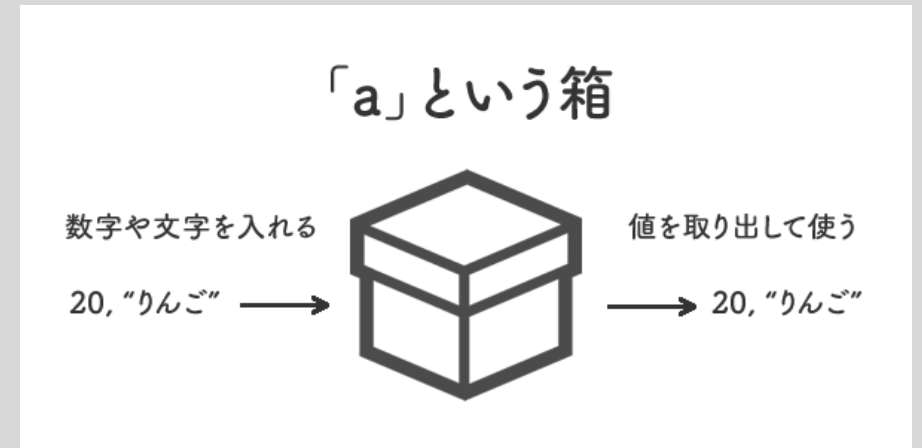
- 同じデータ型の変数をひとつに**まとめたもの**

- オブジェクト指向言語では配列を**オブジェクト**として扱う

➡ 後述

- 文字列はchar型の配列として扱う

- Cでは**静的**な配列、オブジェクト指向では**動的**な配列



基本要素

- 宣言

配列を**作成**すること

- 初期化

配列の作成とともに**数値を割り当てる**こと

- 代入

配列の値を**上書き**すること

宣言

```
int x[5];
```

```
let x;
```

- 。配列の宣言は言語によって様々（静的はC、動的はJSの例）

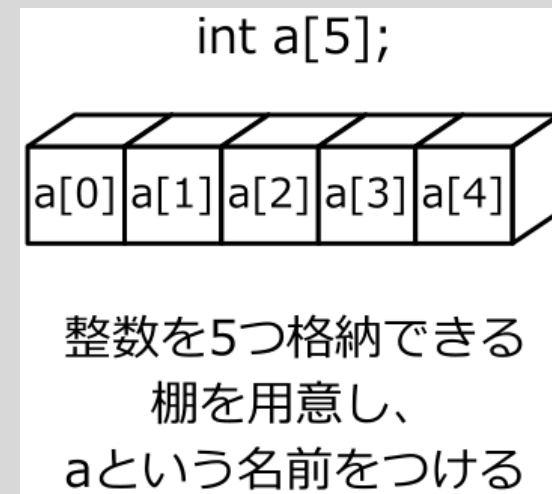
要素型 配列名[要素数]（静的）

let 配列名（動的）

- 。要素数は定数、要素型は要素に入れるデータ型
- 。動的配列は後述のインスタンス化によっても宣言可能
- 。添字演算子

配列名 + [先頭から何個後ろか] として配列にアクセス

[]を添字演算子と呼ぶ



初期化

```
int y[5] = {1, 2};  
//先頭から順に1, 2, 0, 0, 0
```


```
let x = [0, 1, "hoge"];  
//動的では個々に型が分かれる
```

- 宣言と共に値を割り当てる
 - 要素型 配列名[要素数] = { 初期値 } (静 的)
 - let 配列名 = [初期値] (動的) 静的はC、動的はJSの例
- 静的配列：要素数より少ない数を割り当てると、それ以外は0になる
 要素数を超えて割り当てると、エラーになる
- 動的配列：要素数も変動する

またJSの配列(arrayオブジェクト)では、複数のデータ型をまとめることが出来る

代入

```
x[0] = 3;  
//先頭から0番目(先頭含め1つ目)に3を代入  
x[4] = 2;  
//先頭から4番目(先頭含め5つ目)に2を代入
```

- 配列に配列を代入することは不可
  後述のオブジェクトで代入することは可能
- 配列名[先頭からのカウント] = 値
 として添字演算子で配列の要素にアクセスし、代入する
- 配列を丸ごとコピーするには、制御構文を用いた工夫が必要

配列と制御構文

- 列は制御構文を用いて操作をする
➡ 構造化プログラミングの手法
- 代表的なコピーと反転について紹介
- 大抵はfor文の繰り返し処理と組み合わせることが重要

配列のコピー

- for文で要素数分インクリメントして繰り返す
- コピーする側とされる側両方に 同じ要素を指定して代入
- 0からインクリメントすると先頭から順番にコピーされていく

```
int x[5] = {1,2,3,4,5};  
int y[5] = {0}  
  
for(int i = 0, i < 5; i++){  
    y[i] = x[i];  
}
```

配列の反転

- 要素数の半分だけ繰り返す
- 入れ替えるときに塗り替えられる値は
あらかじめ適当な変数に格納して保存しておく

```
int x[5] = {1,2,3,4,5};  
  
for(int i = 0; i < 2; i++){  
    int temp = x[i];  
    x[i] = x[4-i];  
    x[4-i] = temp;  
}
```

x[0]をtempに保存 ➡ x[0]にx[4]代入 ➡ x[4]にtemp代入

x[1]をtempに保存 ➡ x[1]にx[3]代入 ➡ x[3]にtemp代入

多次元配列

- 配列を配列として縦に並べると...

次元の多い配列が出来る



- 配列を要素とした配列と捉える ← 多次元配列

- 添字演算子が1つ増えるだけの違い

1次元配列

0	12
1	24
2	18

要素数 3

2次元配列

0	0	12
	1	24
	2	18
1	0	31
	1	9
	2	27

要素数 2

3次元配列

0	0	0	12
	1	2	18
1	0	1	31
	1	2	27
1	0	0	4
		1	30
	1	0	12
		1	15

要素数 2

多次元配列基本要素

- 要素型 配列名[行要素][列要素] =
 { { 初期値} , { 初期値} ... }
- let 配列名 = [[初期値],[初期値],...]
静的はC、動的はJSの例
- 配列名[行][列]
で配列の要素にアクセスでき、代入が可能
- 多次元配列は行列的なので2重ループを使って処理をすることが殆ど

配列のまとめ

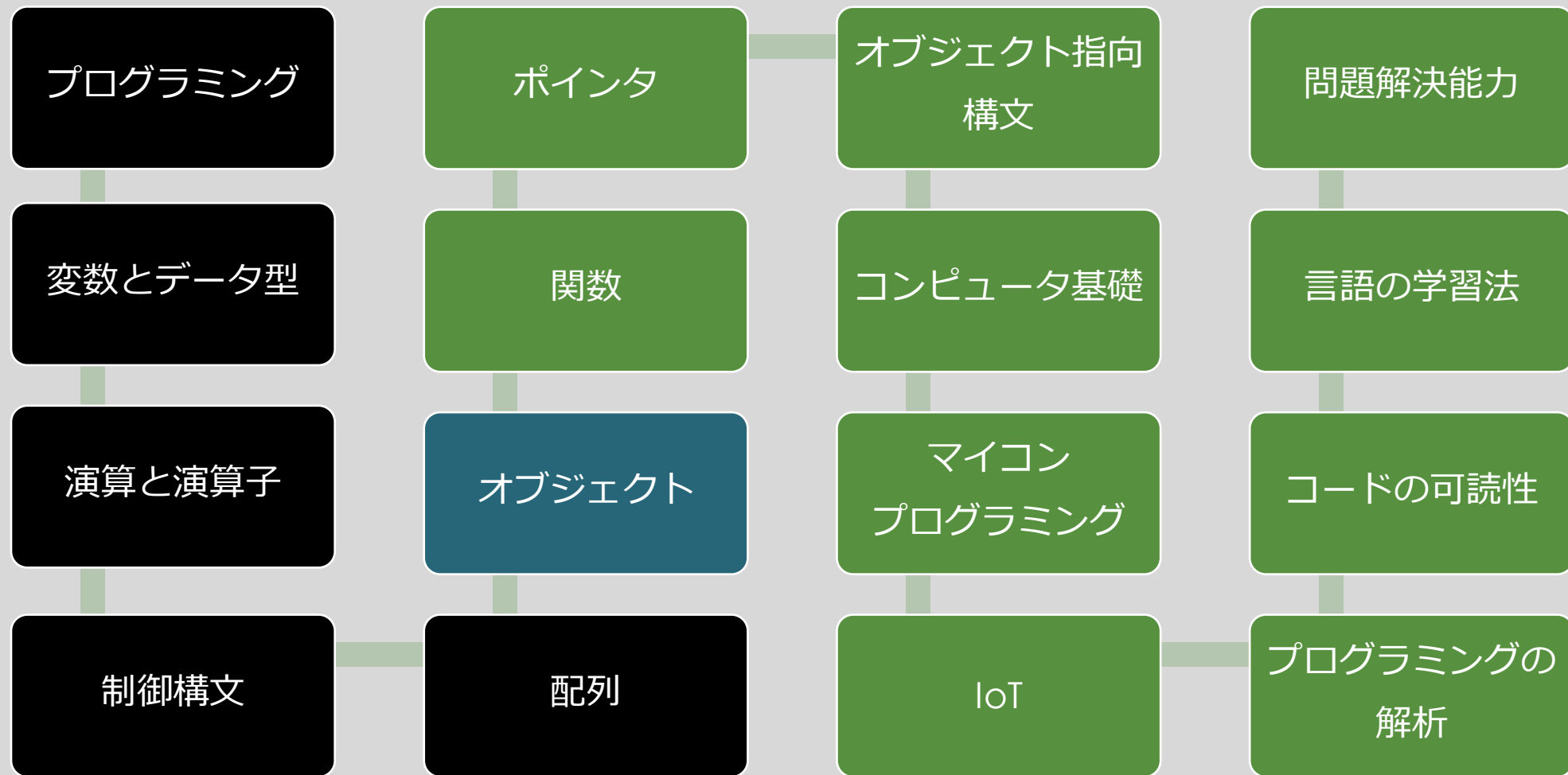
- 。配列にも変数と同じように宣言、初期化、代入の操作が存在
- 。配列丸ごと操作することは出来ず、制御構文を用いて操作する
- 。配列を要素とすることで多次元配列を生成可能に



オブジェクト

～より複雑なデータをより扱いやすく～

進度



オブジェクトとは

- 配列

要素番号(**index**)を指定してアクセスする**単一属性**のデータ群

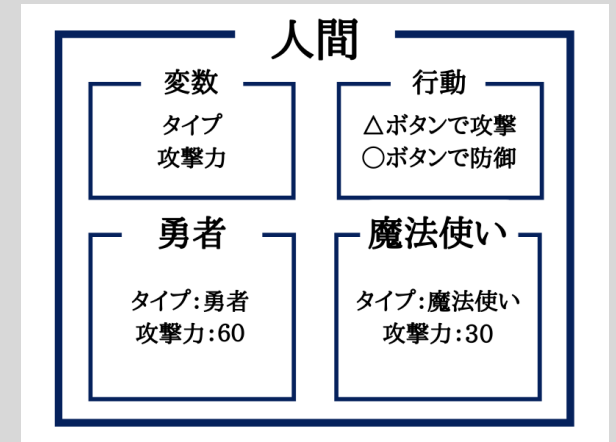
- **連想配列**

設定した名前(**key**)を指定してアクセスする**単一属性**のデータ群

- **オブジェクト**

複数属性で構成され、

それぞれの**名前**を指定してアクセスするひとつの**モノ**



連想配列

- 。名前をキーにアクセスできる配列

```
let student =  
    {name: '太郎', birthday: '2002/02/02'};  
console.log(student.name);  
//太郎が表示される  
console.log(student['birthday']);  
//2002/02/02が表示される
```

let 変数名 = { キー名:値, キー名:値,...}

として初期化。キーは文字列でも数値でも可

変数名.キー名 (ドット演算子)

変数名['キー名'] (添字演算子)

としてアクセスが可能。ドット演算子については後述。

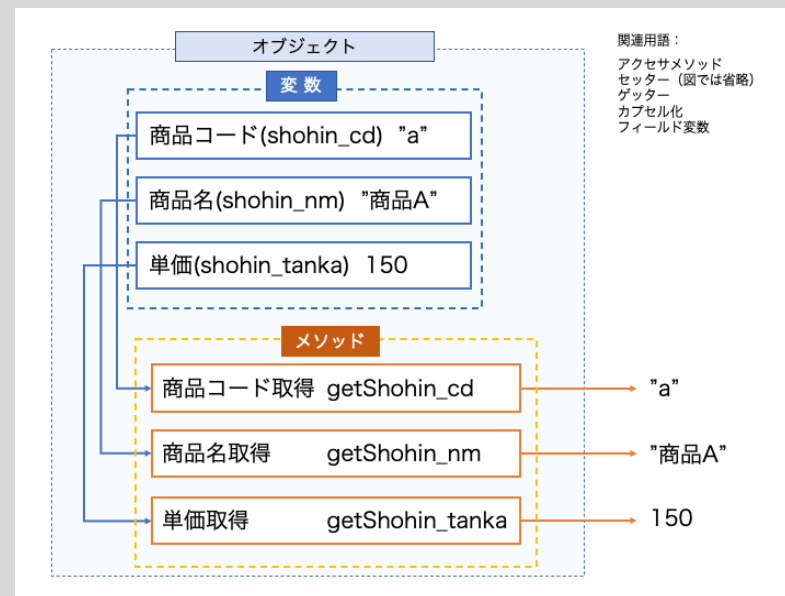
オブジェクトの構成

- プロパティ
 - データプロパティ
オブジェクトの状態や属性を表す情報
 - アクセサープロパティ
データプロパティへ外部変数からの入出力を行う
ゲッター関数/セッター関数

- メソッド
オブジェクトを操作するツールであり、つまり関数のこと

➡ 関数の章で詳述

プロパティとメソッドを合わせてオブジェクトのメンバという



オブジェクトの基本要素

- インスタンス化

オブジェクトのひな型からコピーを生成すること

- インスタンス

インスタンス化によって生成されたコピー

- コンストラクタ

オブジェクト内に用意されているオブジェクトと同名のメソッド

インスタンスと初期化

```
let date = new Date( '2003/03/14 00:00' );  
//Dateオブジェクトのインスタンス化
```

let 変数名 = new オブジェクト名([引数,...])

としてオブジェクトをインスタンス化して、変数を宣言する

new修飾子：オブジェクトをインスタンスするときを使用

- 変数のデータ型はインスタンス化したオブジェクト型となる
- 引数は先述の**コンストラクタ**に与えられ、オブジェクトの**初期化**を行う

プロパティやメソッドの参照

```
console.log(date.getFullYear());  
//dateのgetFullYearメソッドにより  
//西暦年を4桁で取得
```

変数名.プロパティ名[= 設定値];

としてインスタンス変数のデータプロパティにアクセス



getterとsetterを呼び出してアクセスしている

変数名.メソッド名([引数,...]);

としてインスタンス変数のメソッドにアクセス

。.(ドット演算子)

インスタンス変数のプロパティやメソッドにアクセスするための演算子

静的プロパティ/静的メソッド

```
let now = Date.now();  
//UNIXエポックからの経過ミリ秒を取得
```

- 静的プロパティ/メソッド

インスタンス化をせずにオブジェクトそのものにアクセスが可能

オブジェクト名.プロパティ名[= 設定値];

オブジェクト名.メソッド名([引数,...]);

- 静的メソッドやプロパティを使用することで、

グローバル変数を減らすことが出来る

組み込みオブジェクト

- 基本的なオブジェクト、JavaScriptに組み込まれている

```
let str = '電研';  
//Stringオブジェクトとなる
```

- 特別な宣言や定義なしに利用可能

```
let num = 5;  
//Numberオブジェクトとなる
```

- インスタンス化やnew修飾子等も基本的には使用しない

オブジェクトまとめ

- **連想配列**は名前を**キー**にしてアクセスが出来る配列
- **オブジェクト**は**複数属性**で構成される、ひとつのモノ
- オブジェクトには**プロパティ**と**メソッド**がある
- オブジェクトは**インスタンス化**して使用するが、
静的プロパティ/メソッドはインスタンス化しなくても使用可能

参考文献

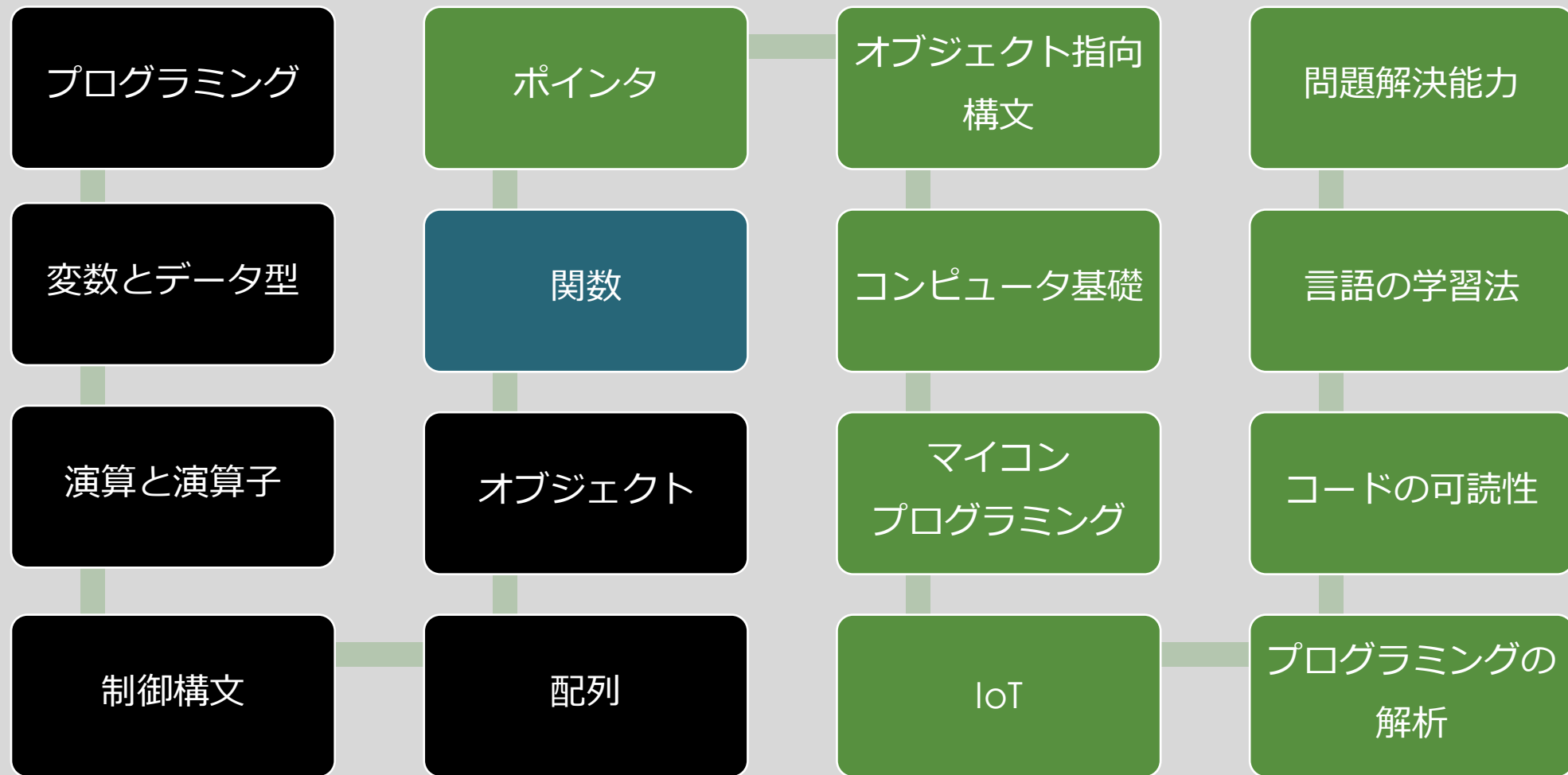
- <https://products.sint.co.jp/topsic/blog/object-oriented>
- <https://blog.codecademy.com/object-orientation-explanation>



関数

～繰り返し処理をひとつに～

進度



関数

- 処理をひとつにまとめて名前を付けたもの
- 関数はプログラム中のどこでも呼び出すことが可能
- メインループもサブルーチンも関数であり、
プログラムは全て関数でできている

基本要素

- 定義

関数の宣言を行う。関数の処理などを決定する

- 呼び出し

関数の処理を実際に行う

- 実引数

呼び出し元が関数に与える値。関数側では**仮引数**という

- 戻り値

関数の呼び出し後、関数から戻ってくる値

関数の定義

```
int sum(int a, int b){  
    int result = a + b;  
    return result;  
}
```

- 以下のように宣言する（静的はC、動的はJSの例）
 - 戻り値の型 関数名(仮引数の宣言) { 処理}
 - function 関数名(仮引数){処理}
- **return** 命令で呼び出し元に返す戻り値を指定し終了できる
- function 命令では引数にletなどを使用して宣言はしない
- 戻り値や引数を指定しない場合、**void**型を使用する

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

オブジェクト指向言語における関数

- オブジェクトが持つ固有の関数：メソッド
- 前頁のように宣言した関数はユーザ定義関数となる
- 両者に機能的な違いはない
- JavaScriptにてユーザ定義関数の宣言法は他にも
Functionオブジェクト、関数リテラル、アロー関数などがある

関数の呼び出し

```
int sum(int a, int b){  
    int result = a + b;  
    return result;  
}
```

引数の流れ

aに2
bに5

```
int val = sum(2, 5);  
printf("%d", val);
```

戻り値の流れ

valにresult

- 関数名(実引数1, 実引数2 ...)

として関数の処理を実行し値を取りだすことが可能

- 実引数を与えて処理を実行し
return文で指定した値が返却される

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

引数の流れ

aに2
bに5

```
let val = sum(2, 5);  
console.log(val);
```

戻り値の流れ

valにa+b

関数のオーバーロード

- C++ などでは同じ名前で処理の違う関数を**複数個**宣言できる
- 引数の**個数**や**データ型**によって振り分けが行われる
- **戻り値のみ**異なる関数はエラーになる

```
int val1 = sum(2, 5);  
int val2 = sum(2, 4, 5);
```

```
int sum(int a, int b){  
    int result = a + b;  
    return result;  
}  
int sum(int a, int b, int c){  
    int result = a + b + c;  
    return result;  
}
```

変数の有効範囲

- **ブロック有効範囲**

関数の**ブロック**({ から } まで)での有効範囲

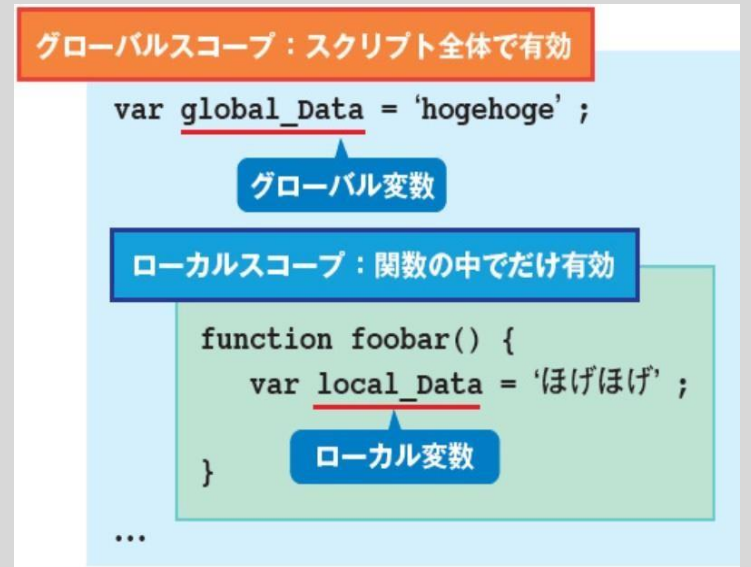
宣言した変数(**ローカル変数**)は外から扱えない

- **ファイル有効範囲**

どのブロックにも属さない一番外側での有効範囲

宣言した変数(**グローバル変数**)はファイル内のどこからでも扱える

➡ 変更されるタイミングが不明なので**可読性**は×



ヘッダファイルとライブラリ

- ~.h (ヘッダファイル)では、
ライブラリにて使用される関数の宣言をしている
- #include 命令を使用することでライブラリ関数が使用可能に
- Arduinoではセンサーなどを扱うときに使用されることがしばしば
- JavaScriptやC#など多くのオブジェクト指向言語では
ヘッダファイルではなくライブラリに様々なメソッドがまとめられている

関数のまとめ

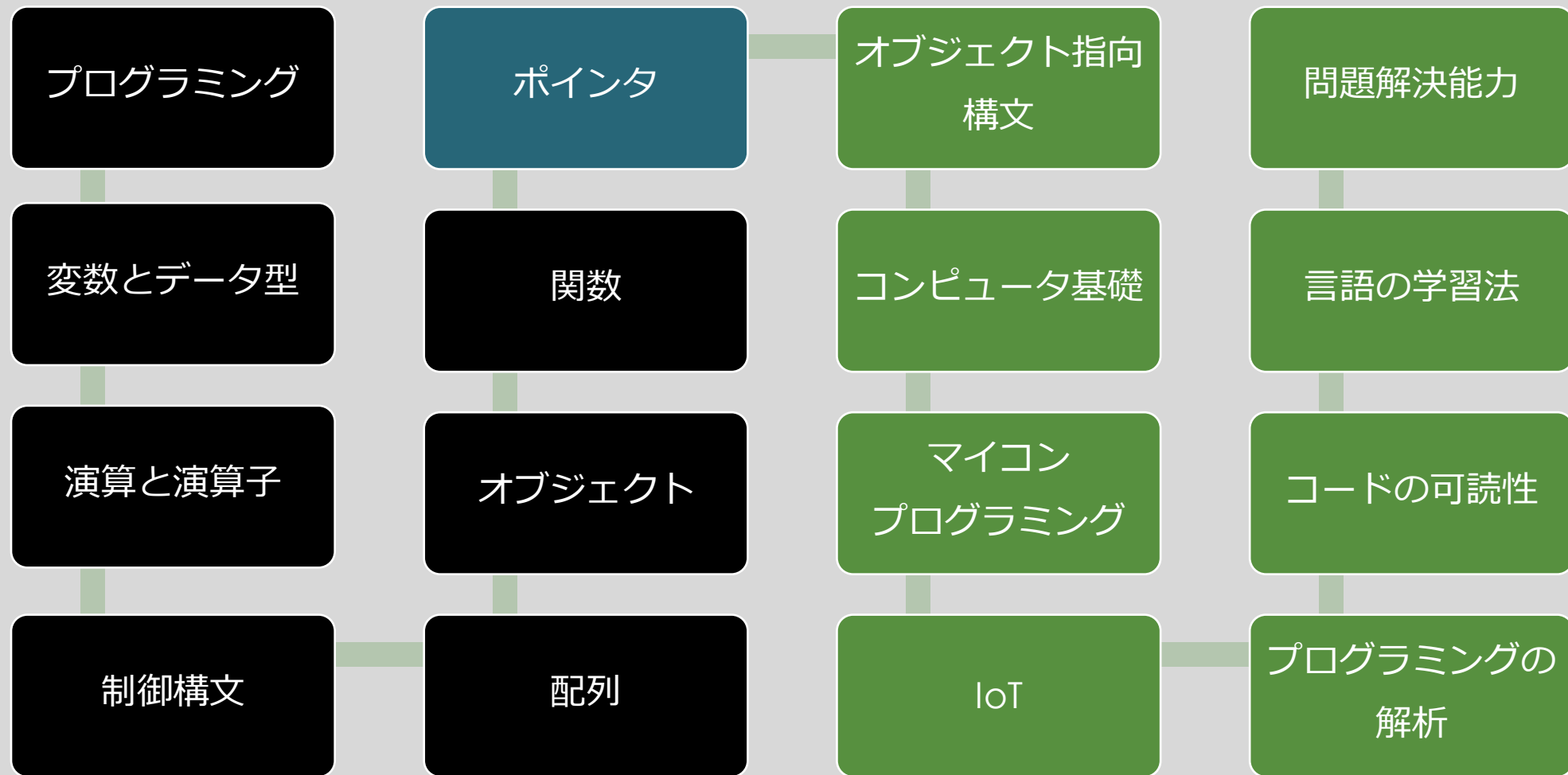
- 関数の操作には定義と呼び出しがある
- 関数内では仮引数として受け取り戻り値をreturnする
- 呼び出し元では実引数を与えて戻り値を受け取る
- 変数の有効範囲は関数のブロックにより決定する



ポインタ

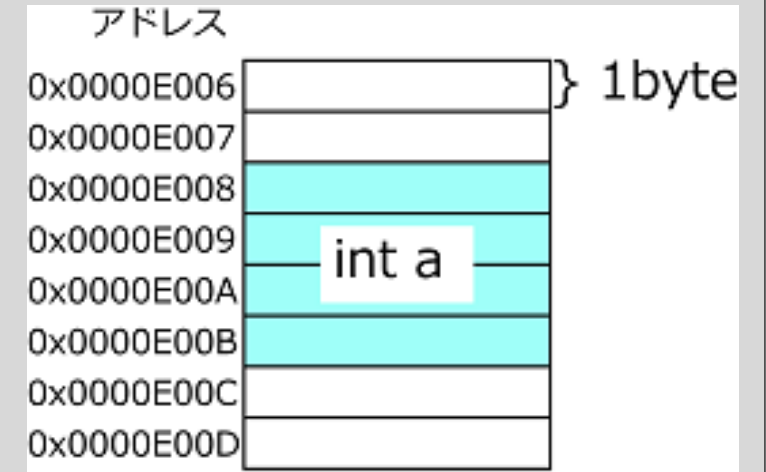
～アドレスからプログラムを操作しよう～

進度



変数の住所

- 。コンピュータにて変数はメモリ上に確保される領域
- 。メモリ上のどこに変数があるかを指すもの ➡ アドレス
- 。アドレスを見ると時々の変数の値が分かる



ポインタ

- 決まったデータ型のアドレスを指す特殊なデータ型
- 変数と同じように扱う
- そのまま代入をするとアドレス自体が変更される

ポインタ基本要素

- 宣言

ポインタ型の変数を宣言

- 初期化

ポインタ型の変数に初期値のアドレスを与える

- 代入

何もつけないとアドレス自体を上書きする

間接演算子を用いることで変数本体を上書きできる

宣言と初期化

- データ型 *ポインタ名 = &変数名
としてポインタ変数を宣言する

- アドレス演算子(単項&演算子)
指定された変数名のアドレスを返す

- ポインタ宣言子(*)
ポインタを表す宣言子、後述の関節演算子とは別物

- ポインタとはアドレスを指す変数のこと！

```
int x = 2;  
int y = 3;  
int *p = &x;
```

2種類の代入

```
p = &y;  
*p = 5;  
//yが5に変更される
```

- ポインタ名 = &変数名

とすることでポインタ自体の値（アドレス）を変更できる

- * ポインタ名 = 値

とすると、ポインタの指すアドレスの変数が上書きされる

➡ 参照による代入という

- この* を間接演算子という

関数への参照渡し

- 仮引数にポインタを宣言すると…
 - 戻り値無しで変数そのものの値を変更



有効範囲外からでも操作が可能に

➡ 変数そのものを渡す、参照渡し

```
void changeval(int *p){  
    *p *= 5;  
}  
int main(void){  
    int x = 2;  
    changeval(&x);  
    //xの値が5倍に  
    return 0;  
}
```

配列の受け渡し

- 参照渡しを使えば**配列**の受け渡しも可能に
- 配列のポインタを**加算**すると
加算した分後ろの要素を指す
- 繰り返し処理でも扱いやすく

```
void changeval(int *a, int n){  
    for(int i = 0; i < n; i++){  
        *(a + i) *= 5;  
    }  
}  
  
int main(void){  
    int x[3] = {1,2,3};  
    changeval(x,3);  
    //単独の配列名は  
    //配列の先頭の変数のアドレス  
    return 0;  
}
```

ポインタのまとめ

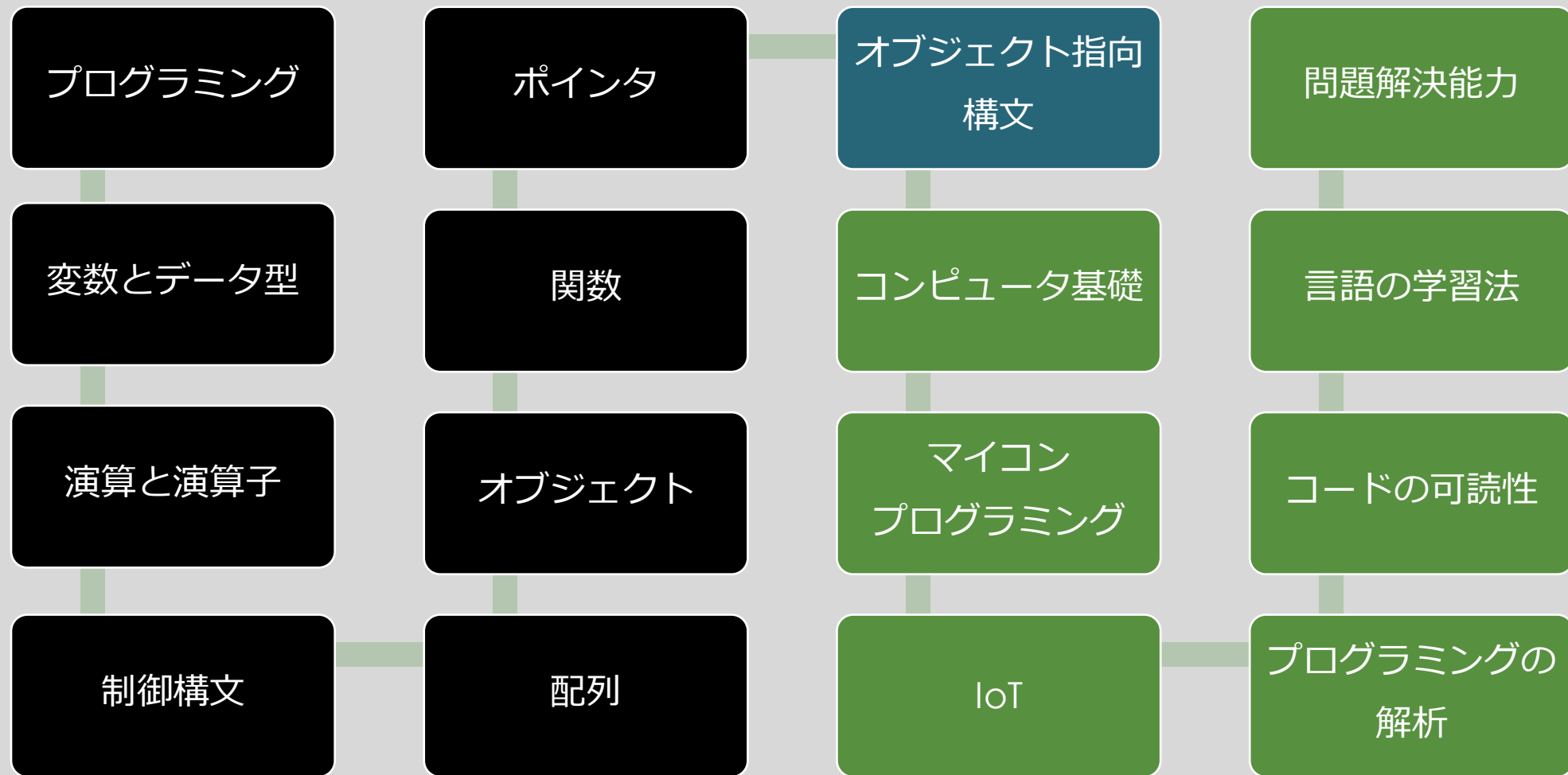
- ポインタは変数のアドレスを指す変数
- 間接演算子を用いて参照による代入が可能
- 関数に参照渡しを行うことで配列の操作も可能に



オブジェクト指向構文

～大規模なプロジェクトを扱いやすく～

進度



クラスベースとプロトタイプベース

。オブジェクト指向

先述したオブジェクトを複製し、新しいオブジェクトを操作する方式

1. クラスベース

複製するオブジェクトのひな型がクラスと呼ばれている

クラスはオブジェクトではなく、生成されたインスタンスがオブジェクトとなる

2. プロトタイプベース

複製するオブジェクトのひな型がプロトタイプオブジェクトとなる

厳密なクラスは存在せず、ひな形のオブジェクトを複製する

クラスの要素

- 定義

クラスの組み立て、変数の宣言のようなもの

- 継承

あるクラスを引き継ぎ

そのクラスに機能を追加した別クラスを定義する

- 操作

クラスをインスタンス化し、インスタンス変数进行操作する

クラスの定義

class クラス名{クラスの定義}

としてクラスを宣言する

- JSでは、コンストラクター内で

this.プロパティ名 = 値

としてプロパティを宣言

- C#では変数の宣言に加え

後述の**アクセス修飾子**がつく

- thisキーワードは

インスタンス自信をさす

```
class member{
    constructor(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }
    //メソッド
    getName(){
        return this.firstName + this.lastName;
    }
}
```

```
class member{
    //データプロパティ
    private string firstName;
    private string lastName;
    //コンストラクタ
    public member(string first, string last){
        this.firstName = first;
        this.lastName = last;
    }
    //メソッド
    public getName(){
        return this.firstName + this.lastName;
    }
}
```

クラスの定義

- JSでは**constructor**

C#では同名の**メソッド**

でコンストラクタを定義

- **メソッド**は

メソッド名(引数){処理} で宣言

C#では**アクセス修飾子**がつく

```
class member{
    constructor(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }
    //メソッド
    getName(){
        return this.firstName + this.lastName;
    }
}
```

```
class member{
    //データプロパティ
    private string firstName;
    private string lastName;
    //コンストラクタ
    public member(string first, string last){
        this.firstName = first;
        this.lastName = last;
    }
    //メソッド
    public getName(){
        return this.firstName + this.lastName;
    }
}
```

その他クラスの定義

- **static**修飾子を使用すると
静的メソッド/プロパティを定義することができる
- C#ではメソッドやコンストラクタの**オーバーロード**が可能
引数の有無や個数で**複数**のコンストラクタを定義可能

クラスの継承

JSでは **extends** クラス名

C#では **:** クラス名

としてクラスを**継承**

```
class BusinessMember extends Member{  
    work(){  
        return this.getName() + 'は働いています。';  
    }  
}
```

```
class BusinessMember : member{  
    public work(){  
        return this.getName() + 'は働いています。';  
    }  
}
```

- **基底クラス**

継承元となったクラス、上の例ではmemberクラス

- **派生クラス**

継承後のクラス、上の例ではBusinessMemberクラス

クラスのまとめ

- オブジェクト指向にはクラスベースとプロトタイプベースがある
- クラスやプロトタイプには定義、継承、操作という要素がある
- JSではconstructor、C#では同名のメソッド
によってコンストラクタを生成する
- JSではextends、C#では : を使用してクラスを継承する

プロトタイプのコストラクタ

```
let member = function(){};
```

- 。プロトタイプも基本的な要素はクラスと同じ

let オブジェクト名 (コストラクタ名) = function(){};

としてプロトタイプオブジェクトのコストラクタを宣言

インスタンス化はクラスと同様にする

- 。JSでは関数オブジェクトをプロトタイプのコストラクタとしている

プロトタイプオブジェクトのプロパティとメソッド

コンストラクタ関数内で

this.プロパティ名 = 値

としてプロパティを初期化

this.メソッド名 = function(引数){処理}

としてメソッドを初期化する

インスタンス変数.メソッド名 = function(引数){処理}

と任意の場所に記述すると、動的にメソッドを

生成された**インスタンス**に対して追加することも可能

```
let member = function(firstName,lastName){  
  //データプロパティ  
  this.firstName = firstName;  
  this.lastName = lastName;  
  //メソッド  
  this.getName = function(){  
    return this.lastName +  
      this.firstName;  
  }  
};
```

コンストラクターとプロトタイプ

- コンストラクタ

インスタンスの度にメンバのメモリ領域を確保

プロパティは個別が良いが、メソッドは共通にしたい...

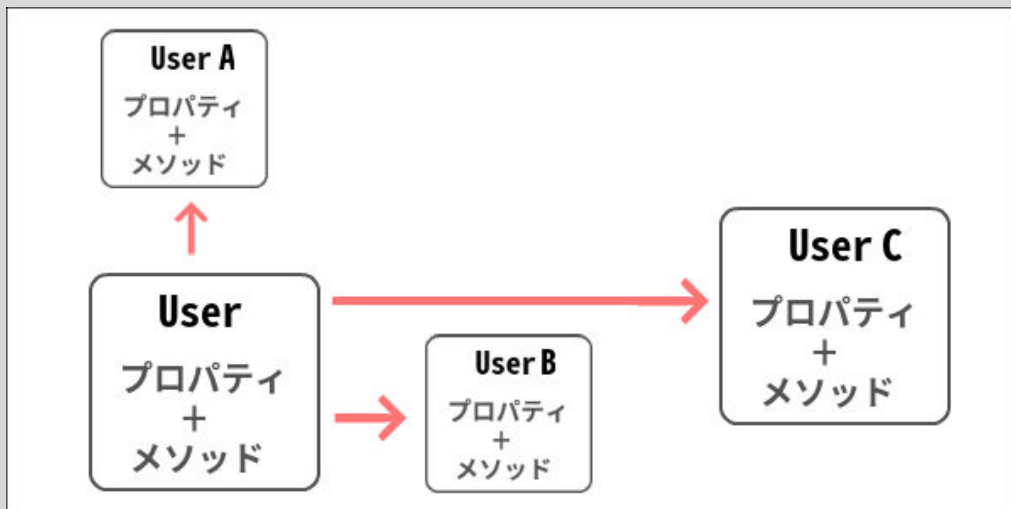


- プロトタイププロパティ

インスタンス化しても内容がコピーされないプロパティ

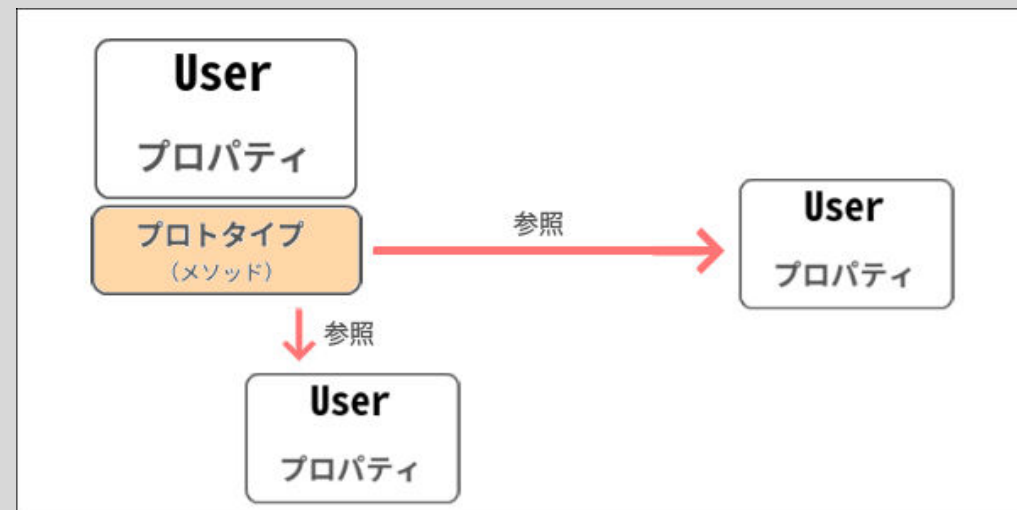
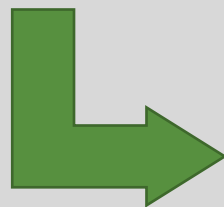
インスタンスから元オブジェクトのプロトタイプに暗黙的な参照を行う

コンストラクターとプロトタイプ



コンストラクタ
インスタンス毎にコピーされる

プロトタイプ
インスタンスから参照される



プロトタイププロパティ

```
let member = function(firstName, lastName){  
  this.firstName = firstName;  
  this.lastName = lastName;  
}  
//コンストラクタ  
member.prototype.getName = function(){  
  return this.firstName + ' ' + this.lastName;  
}  
//プロトタイププロパティ
```

オブジェクト名.**prototype.メソッド名** = function(){処理}
としてプロトタイプのメソッドを設定

- 。通常、プロトタイプは**メソッド**でのみ使用する
データプロパティでも使用できるが**する必要がない**

プロトタイププロパティ

オブジェクト名.prototype = {プロトタイプメソッド}

でまとめてプロトタイプメソッドを定義することが可能

◦ {}内では

メソッド名: function(引数){処理},

メソッド名: function(引数){処理},...

```
let member = function(firstName, lastName){
  this.firstName = firstName;
  this.lastName = lastName;
}
//プロトタイププロパティ
member.prototype = {
  getName: function(){
    return this.firstName + ' '
      + this.lastName;
  },
  setFirstName: function(firstName){
    this.firstName = firstName;
  }
}
```

静的メソッド/プロパティ

オブジェクト名.プロパティ名 = 値
として静的プロパティを設定

オブジェクト名.メソッド名 = function(){処理}
として静的メソッドを設定

```
let Area = new function(){  
  //コンストラクタ  
  Area.version = 1.0;  
  //静的プロパティ  
  Area.triangle = function(base, height){  
    return base * height / 2;  
  }  
  //静的メソッド
```

プロトタイプの継承

◦ 継承

1. コンストラクタを**基底オブジェクト**から
オブジェクト名.call(this, 引数,...)
で呼び出したり、新たに**定義**したりする
2. **派生オブジェクト名.prototype = new 基底オブジェクト名();**
としてプロトタイプを継承、基底オブジェクトのメンバを参照可能に

◦ プロトタイプチェーン

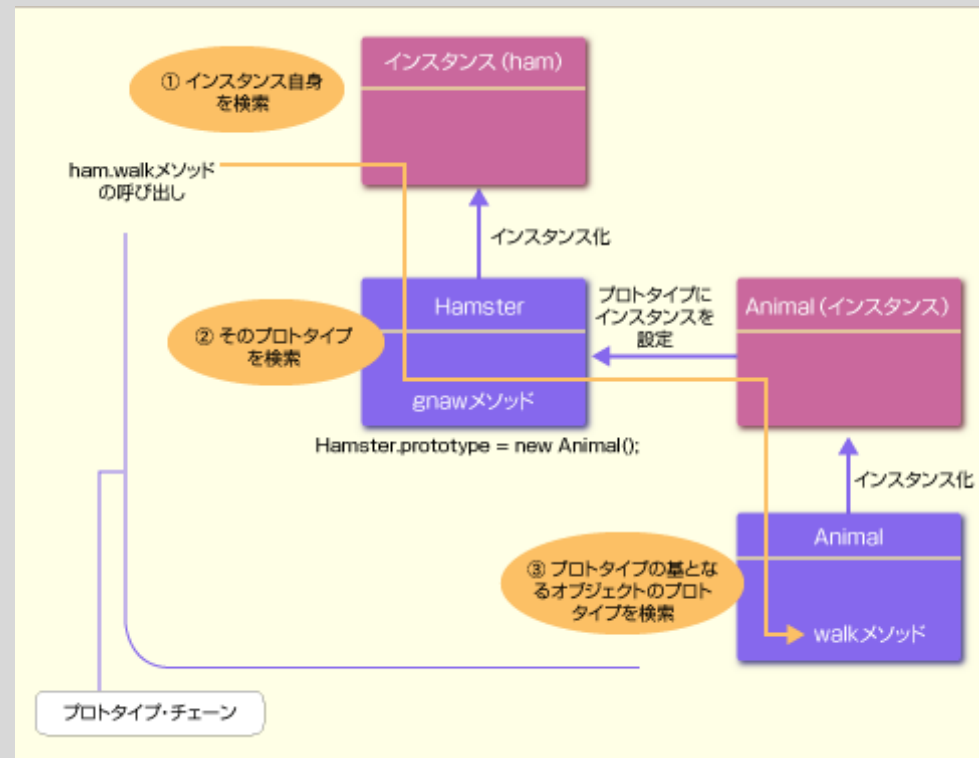
プロトタイプ内のメソッドを呼び出すには、決まった**順番**がある

```
let Animal = function(){};
//基底オブジェクトのコンストラクタ
Animal.prototype.walk = function(){
  console.log('トコトコ...');
}
//基底オブジェクトのプロトタイプ
let Dog = function(){
  Animal.call(this);
}
//派生オブジェクトのコンストラクタ
//基底オブジェクトから呼び出している
Dog.prototype = new Animal();
//派生オブジェクトのプロトタイプに
//基底オブジェクトのインスタンスを設定
Dog.prototype.bark = function(){
  console.log('ワンワン!');
}
//派生オブジェクト独自に
//プロトタイプメソッドを設定
```

プロトタイプチェーン

。プロトタイプは以下の順番で
メソッドを呼び出す

1. 派生オブジェクトのインスタンス自身のメンバ
2. 派生オブジェクトのプロトタイプメソッド
3. 派生オブジェクトに登録した基底オブジェクトのインスタンス自身のメンバ
4. 基底オブジェクトのプロトタイプメソッド



プロトタイプのおまとめ

- プロタイプオブジェクトは関数オブジェクトでコンストラクタを宣言
- コンストラクタ内でプロパティやメソッドを設定
- 派生オブジェクトのプロトタイププロパティに
基底オブジェクトのインスタンスを設定して継承
- プロトタイプチェーンに沿ってメソッドを呼び出す

メンバのアクセス

- プライベートメンバ

クラスやプロトタイプオブジェクト内部からのみ

アクセスが出来るメンバ

- パブリックメンバ

クラスやオブジェクトの内外から自由にアクセスが出来るメンバ

メンバのアクセス

- JSではコンストラクタ内で
 - **let**を用いてプライベートメンバを宣言
 - **this**キーワードを用いてパブリックメンバを宣言

厳密にはプライベートメンバを宣言する構文は存在しない

- C#ではアクセス修飾子を用いて
 - **private修飾子**：プライベートメンバ
 - **public修飾子**：パブリックメンバ

として明示的に宣言することが可能

ゲッターとセッター

- ゲッター

プライベートメンバを外部に出力するメソッド

- セッター

プライベートメンバに外部から入力するメソッド

言語によって様々な方法がある

JSはそれぞれをメソッドと同じように定義する


名前空間

- クラスが多くなった時に、クラスの名前が競合する可能性が
- クラスを名前空間で仕分け、競合を回避
 - JSでは`let`を用いた空のオブジェクト
 - C#では`namespace`修飾子
で名前空間を定義することが出来る



オブジェクト指向のアクセスまとめ

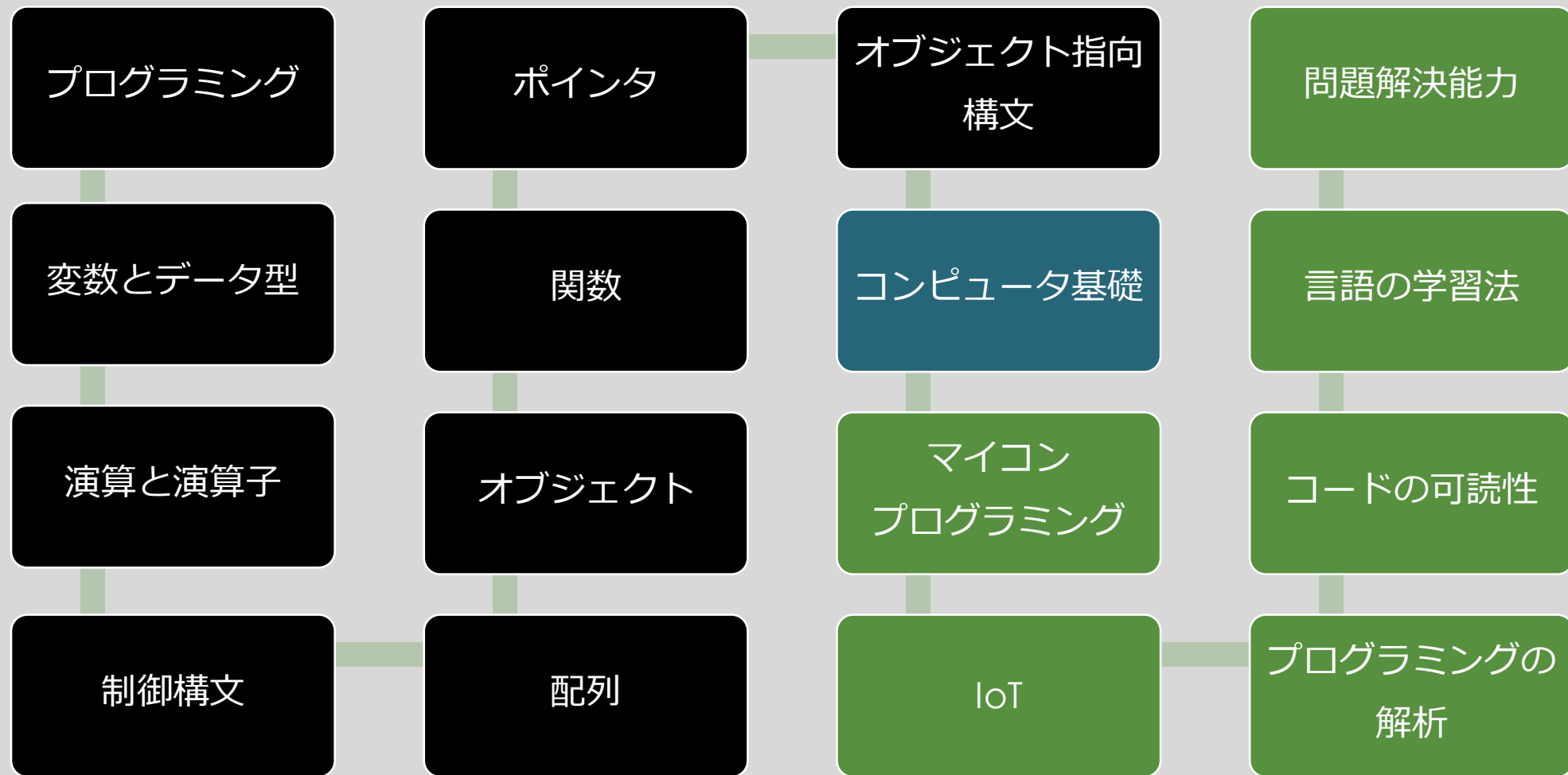
- メンバには外部からアクセス不可の**プライベートメンバ**と外部からアクセス可能な**パブリックメンバ**がある
- プライベートメンバを外部に入出力するために**ゲッター**と**セッター**がある
- **名前空間**を使用するとクラス名で**競合**を避けることが可能



コンピュータ基礎

～コンピュータの仕組み～

進度



ノイマンコンピュータの大原則～ 5 大装置～

- CPU

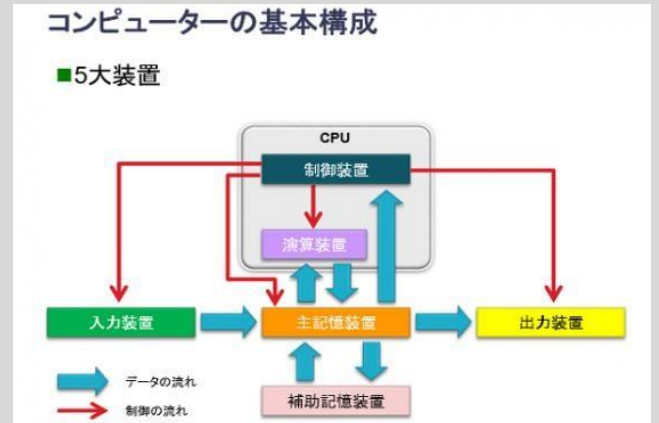
プログラムを実行する**制御**装置、算術演算を行う**演算**装置

- インターフェース

人とコンピュータを繋ぐ装置。人からの**入力**、人への**出力**を行う。

- メモリ

プログラムやデータを**記憶**する装置。



CPU

- 保存されたプログラムを実行し、コンピュータ全体を制御する**制御装**



算術演算や論理演算を実行し、データ処理を行う**演算装置**



CPU : Central Processing Unit
中央処理装置



メモリ

- 主記憶装置

一時的にデータを保存しておく装置。**RAM**と呼ばれる
SDRやDDR、レジスタやキャッシュ（CPU内部メモリ）など

- 補助記憶装置

永続的にデータを保存しておく装置。**ROM**と呼ばれる。
HDDやSSDなど、様々な物理的機構を用いて保管



インターフェース

- 入力装置

人からコンピュータへデータを送る装置。

マウスやキーボード、マイクやカメラに加えてペンタブなども

- 出力装置

コンピュータから人へデータを送る装置。

ディスプレイやスピーカー、プリンタなど

コンピュータとプログラム

フラッシュメモリにプログラムを**保存**



電源がつくとコンピュータがプログラムを**実行**



電源がついている限りプログラムの処理を**ループ**し続ける⇐**無限ループ**

スリープ状態はボタンの**入力**を**待機**するプログラムが…

プログラミング言語とアセンブリ

- 高水準言語

C言語～の言語。人が理解しやすいプログラム

- 低水準言語

～アセンブリの言語。機械が理解しやすいプログラム

コンパイルとアセンブル

- コンパイル

高水準言語から低水準言語に翻訳すること


- アセンブル

アセンブリ言語を機械語（バイナリ）に翻訳すること

コンパイルとアセンブルはほとんどの場合セット

コンピュータ基礎まとめ

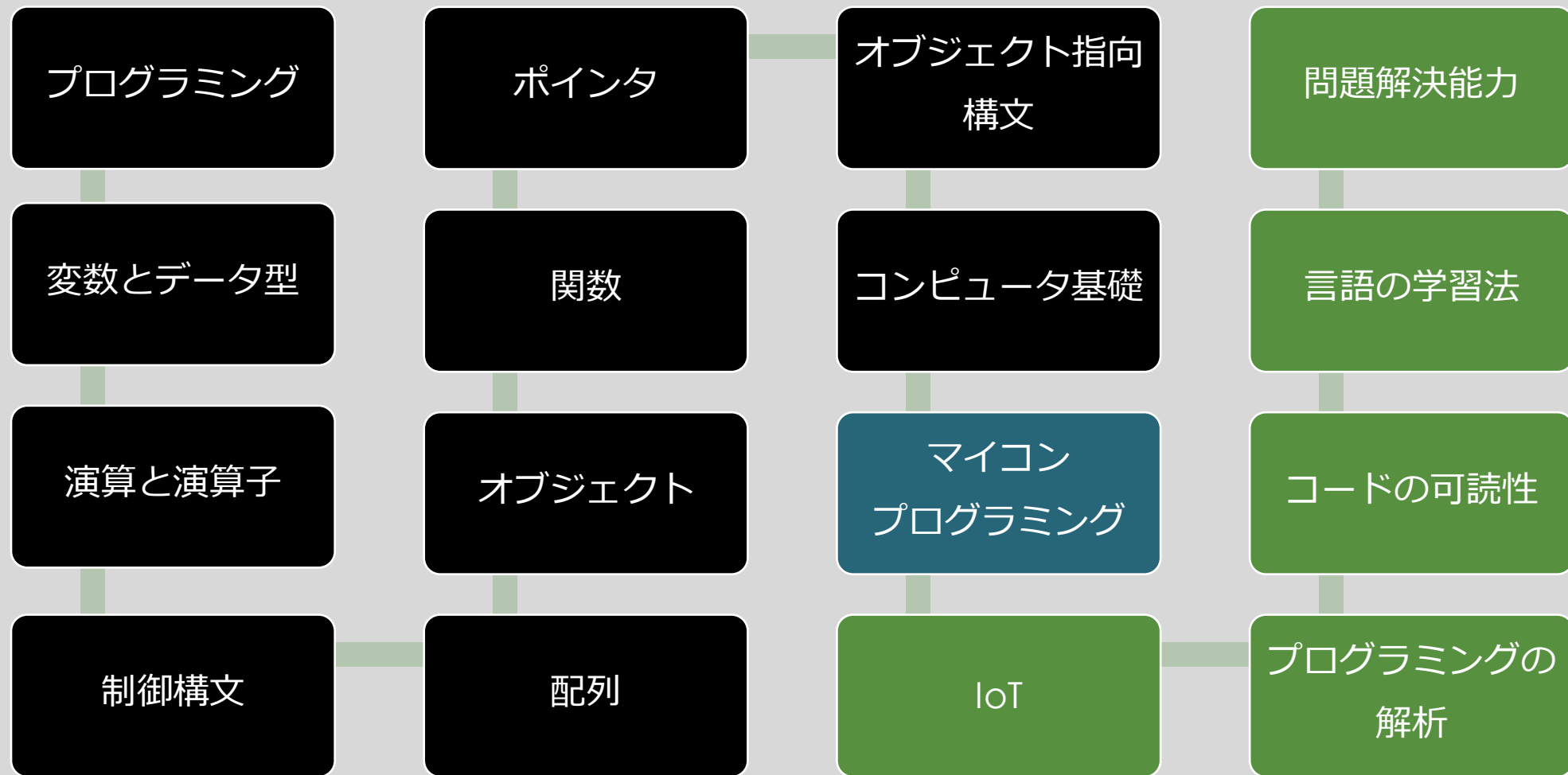
- コンピュータにはCPU、メモリ、インターフェースの3つに
重要な5つの機能が含まれている
- プログラムはコンピュータの電源を付けたときから無限ループで開始
- プログラミング言語をコンパイルして機械語がコンピュータに指令を



マイコンプログラミング

～小さな部品に使用される大きな技術～

進度



マイクロコンピュータ

- 。マイクロコンピュータ（コントローラ）、マイコン
フラッシュメモリにプログラムした通りに動く電子部品

中に保存されたプログラム（ソフトウェア）が
接続された回路（ハードウェア）の脳となり動かす

マイコンの種類

- AVRマイコン

アトメル社のマイコン、PICマイコンのライバル

- PICマイコン

マイクロチップ社のマイコン、AVRマイコンのライバル

- ARMマイコン

arm社のアーキテクチャを搭載したマイコン、非常に高性能

マイコンプログラミングの基礎

- **グローバル変数**はなるべく控える

フラッシュメモリの容量を圧迫してしまうため

- **メインループ**が電源を切断するまで回り続ける

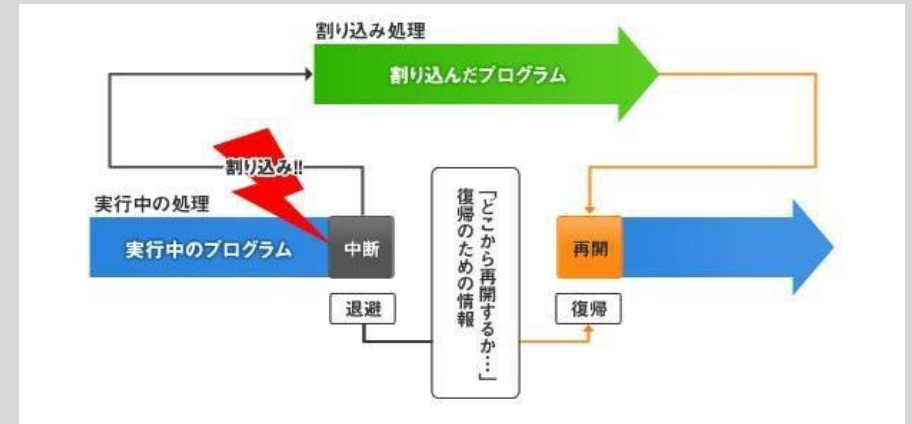
前章参照、変数設定などの前処理はメインループ外で

- **デバッグ**はシリアル通信で動的解析

動的解析は後述、**Serial.println**を使用して値を表示する

割り込み

- 外部の**入力**や予め決めた**時間**に発生する
- メインループの処理が**ストップ**し定義しておいた処理が発生
- 割り込み処理が**終了**すると、元の位置からメイン処理が**再開**



ファームウェアとソフトウェア

- ファームウェア

ハードウェアを**直接的**に制御するプログラムをファームウェアという

- ソフトウェア

ハードウェアをファームウェアを介して**間接的に**制御するプログラム

 マイコンプログラムなど

ハードウェアやファームウェアは回路やチップで決定する = **固い**

ソフトウェアはプログラムで柔軟に変化する = **柔らかい**

ハードウェア

- **ハードウェア**

マイコンを含める**回路全体**、駆動部までを含めることもある

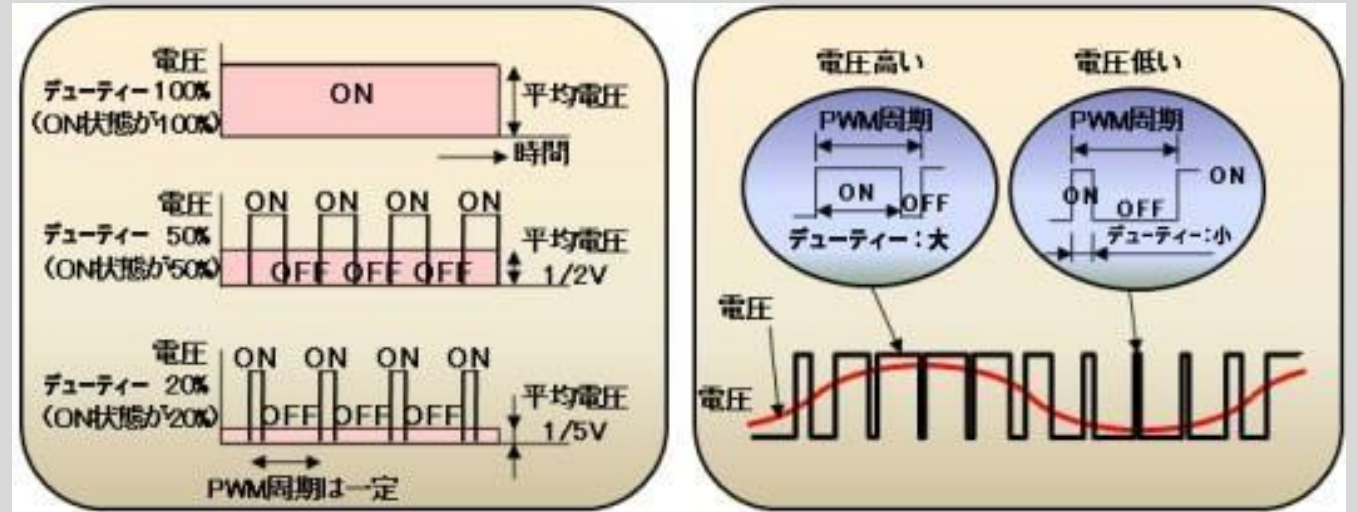
- ハードウェアをマイコンで制御する様々な制御方法を紹介

GPIO

- 汎用IO（デジタル入出力ピン）のこと
- 0又は1の2種類の電圧をLow又はHighで表現して入出力を行う
- 一般的なマイコンの入出力ポート

PWM

- 。パルス幅変調のこと
- 。デジタルで表現される矩形波のOn/Off時間を制御することで疑似的なアナログ電圧を作ることが可能に
- 。AC/DCコンバータやモータの制御など様々なものに活用されている



DACとADC

- **DAC**（デジタルアナログコンバータ）
 - PWMによる制御で**デジタルな信号から**アナログ電圧を生成する
- **ADC**（アナログデジタルコンバータ）
 - PWMによって**アナログの入力値**をデジタル信号に変換する
 - マイコンにもよるがGPIOのうち使用できるピンが限られている

UART

- **パラレル**を**シリアル**に変換して行う高効率の通信のこと
- マイコンは**UART**でPCとシリアル通信を行っている
- Arduinoでは**Serial.println**という関数を使用して出力できる

SPIとI2C

- **SPI**通信

同期式のシリアル通信で主に**センサ類**と行う通信方式
通信する**センサの数**の配線が必要

- **I2C**通信

SPIと同じく同期式のシリアル通信で**センサ類**と行う通信
配線が共通で、**複数**個のセンサを**まとめて**扱えるという利点がある

マイコンプログラミングのまとめ

- マイコンはプログラム通りに動く**電子部品**
- 様々な種類があり、プログラミングするときには**注意**が必要
- マイコンは**ハードウェア**を様々な技術を使って制御している

参考文献

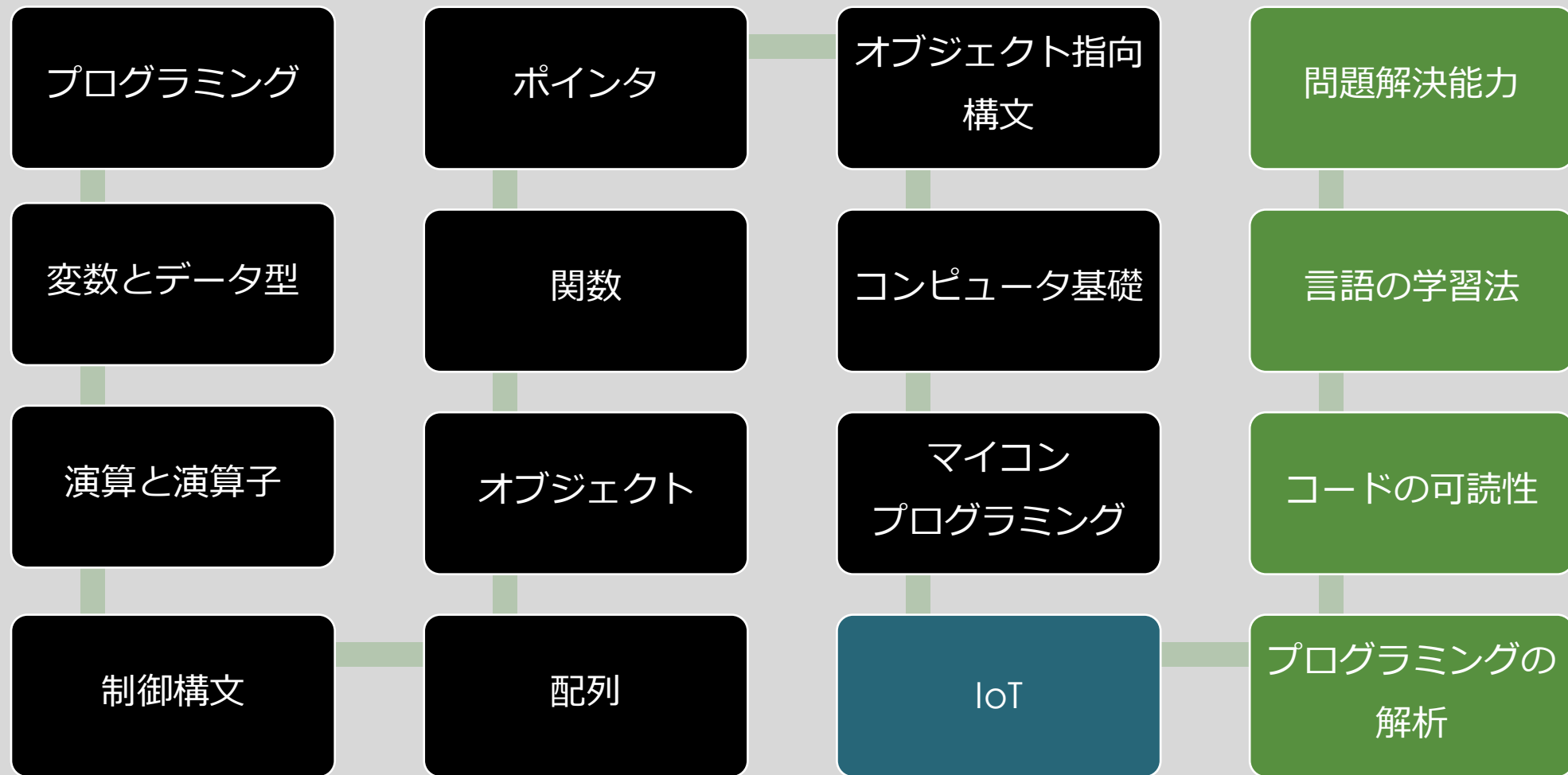
- <https://eng-entrance.com/linux-shellscript-variable>
- <https://xtech.nikkei.com/it/atcl/column/14/091700069/091700002/>
- <https://wa3.i-3-i.info/diff446programming.html>
- <http://www.b.s.osakafu-u.ac.jp/~hezoe/pro/chapter5.html>
- https://kanda-it-school-kensyu.com/php-super-intro-contents/psi_ch09/psi_0905/
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch06/jbi_0606/
- <https://itmanabi.com/structured-objectoriented-prog/>
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch07/jbi_0704/
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch07/jbi_0703/
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch07/jbi_0702/
- <https://www.crucial.jp/memory/ddr4/ct2k16g4dfd824a>
- <https://brain.cc.kogakuin.ac.jp/~kanamaru/lecture/prog1/06-01.html>
- <https://www.javadrive.jp/start/array/index7.html>
- <https://www.renesas.com/jp/ja/support/engineer-school/mcu-programming-peripherals-04-interrupts>
- <https://toshiba.semicon-storage.com/jp/semiconductor/knowledge/e-learning/brushless-motor/chapter3/what-pwm.html>



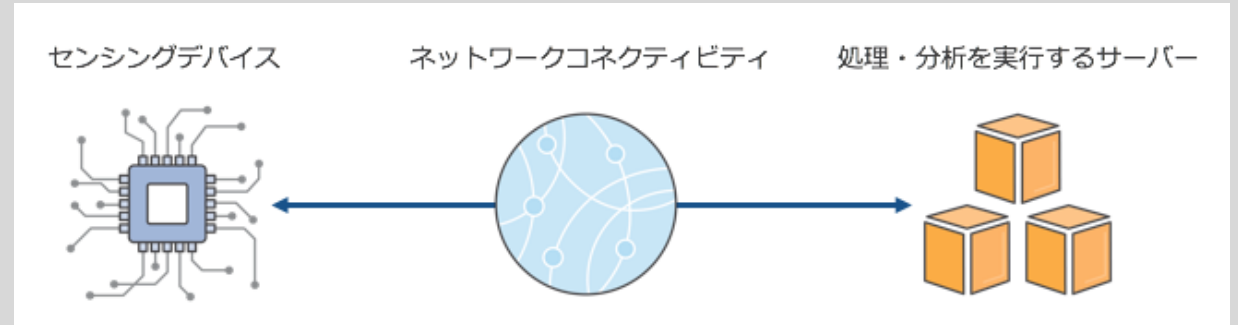
IOT

～インターネットで人もデバイスも一つに～

進度



IoTとは



- **Internet of Things** ～モノのインターネット～
モノ作りは各種デバイスの開発から、それを**統合**したIoTの時代へ
- ハードウェアがインターネットで**通信**しネットワークを構築
サーバーが主体となってコマンドひとつで管理
- IoTには**操作**、**状態取得**、**動きの検知**や**モノの通信**の4要素がある

モノの一括操作

- 離れた場所にある

様々なデバイスを**操作**する



- **Web**のプラットフォームから、

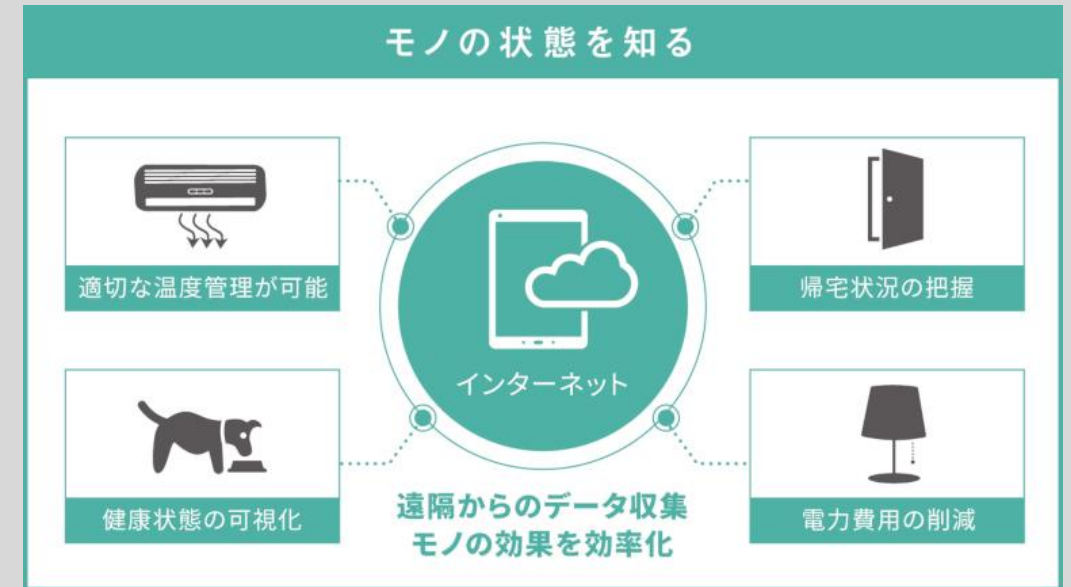
同じインターネットに接続されている**複数のデバイス**を一括管理

状態の取得

- デバイスに接続される**センサ**を使用しリアルタイムに**モノの状態**を取得

- **WebAPI**を使用して

天気予報や**交通情報**などのビッグデータを取得することも



動きの検知

- 。デバイスに接続されるセンサを使用しリアルタイムにモノの動きを取得



- 。位置情報サービスを使用した**使用者の位置**や
人感センサを使用した**人の検知**、画像認識による**人の動き**など

モノの通信

- 。状態や動きというデータをデバイスから取得し、サーバが**判断**する

- 。情報を**処理**し、サーバからデバイスを**操作**し、様々な**自動化**を行う



デバイスとサーバで**相互に通信**し

モノのネットワークでより良い社会を実現する動き



レイヤー

- モノづくりには段階的な層が存在しているという考え方
- IoTでいうと以下のような層が考えられる
 1. ハードウェア
 2. 通信技術
 3. ウェブやサーバー

IoTを実現する技術

- IoTプログラミングは

高いレイヤーから低いレイヤーまで様々な言語が使用される

- ハードウェアにはマイコンが組み込まれ、通信モジュールとセンサを制御
- インターネットで通信するためのWiFiなど情報通信の技術
- ウェブのプラットフォームを使用するためウェブやサーバーの技術
- 情報をハッカーから保護するセキュリティ分野の技術

情報セキュリティ

- IoTには各層のレイヤーで情報を保護しないといけない
- ハードウェアではメモリやCPUなど、
ソフトウェアでは暗号などで情報を保護
- IoTセキュリティ：ハードウェアからソフトウェアまでの
幅広い分野で情報を守っている

IoTまとめ

- IoTには操作、状態取得、動きの検知やモノの通信の4要素がある
- IoTを実現するために様々なレイヤーの様々な技術が存在
- 攻撃者から様々な層で情報を守るIoTセキュリティという分野もある

参考文献

- <https://monstar-lab.com/dx/technology/about-iot/>



プログラミングの解析

～処理を見極める～

進度



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

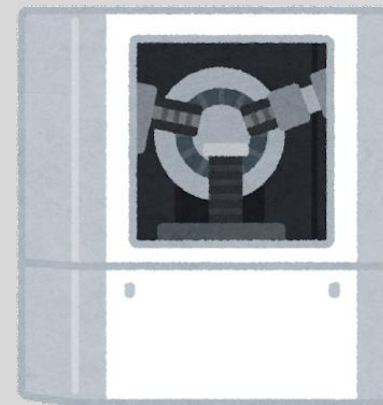


プログラムの解析で
読む**ファイル**を決定



コードの解析で
改良や**引き出し**を
行う

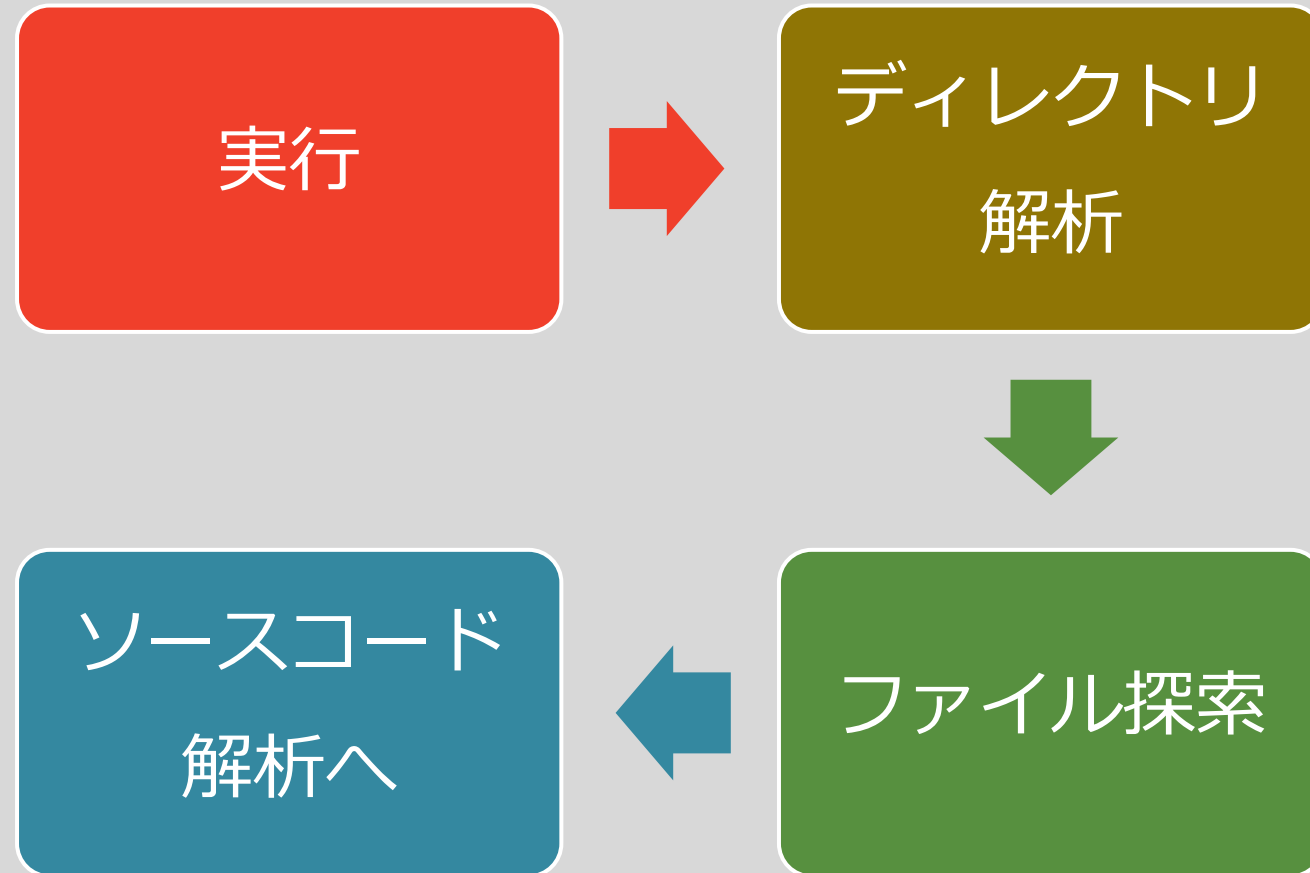
プログラム解析



- 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



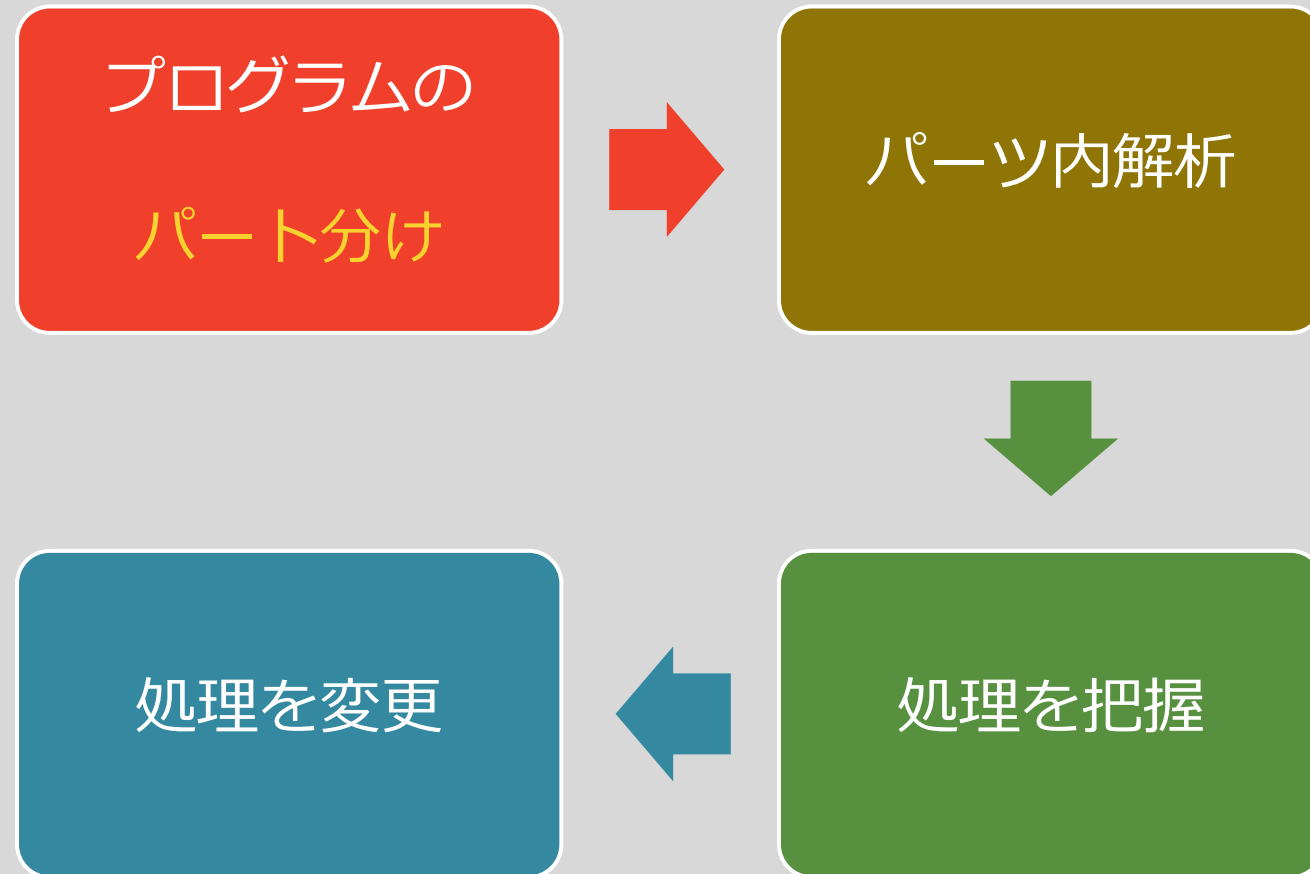
- プログラミングの解析における**醍醐味**
- ソースコードを解析するには、以下の手段が存在
 - **動的解析**
デバッグを用いる方法であり、変数の値などを随時**確認**できる
 - **静的解析**
手動で読み解く方法であり、変数の値などは完全に**予測**する

ソースコードの動的解析

～デバッグ～

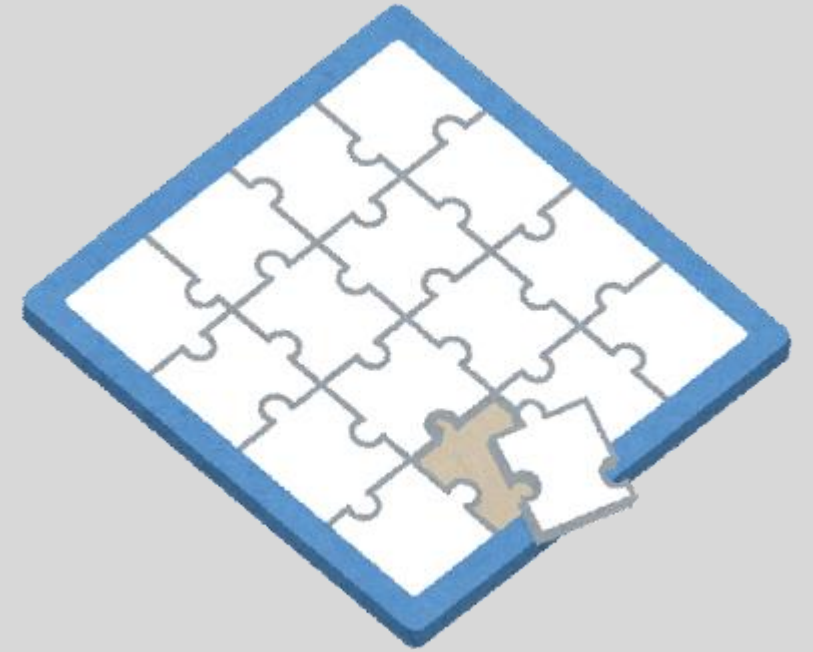
- コードを実行し、**任意の場所**を**観測**することで解析する方法
- デバッガーを使用する方法
 - VisualStudio**などの**デバッグツール**を用いて
コードを**改変せず**に必要な情報を得る
- デバッガーを使用しない方法
 - printf**や**console**などの**標準関数**を確認する場所に**挿入**する

ソースコード静的解析の手順



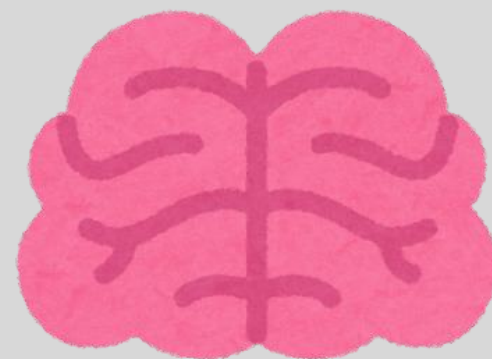
パート分け

- 。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

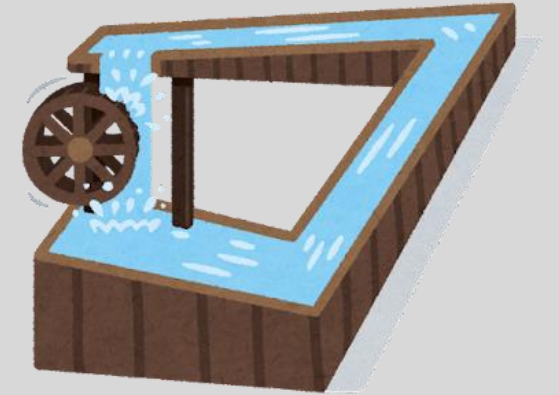
- 。分けたパーツを確認し、任意のパーツを絞り出す



- 。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



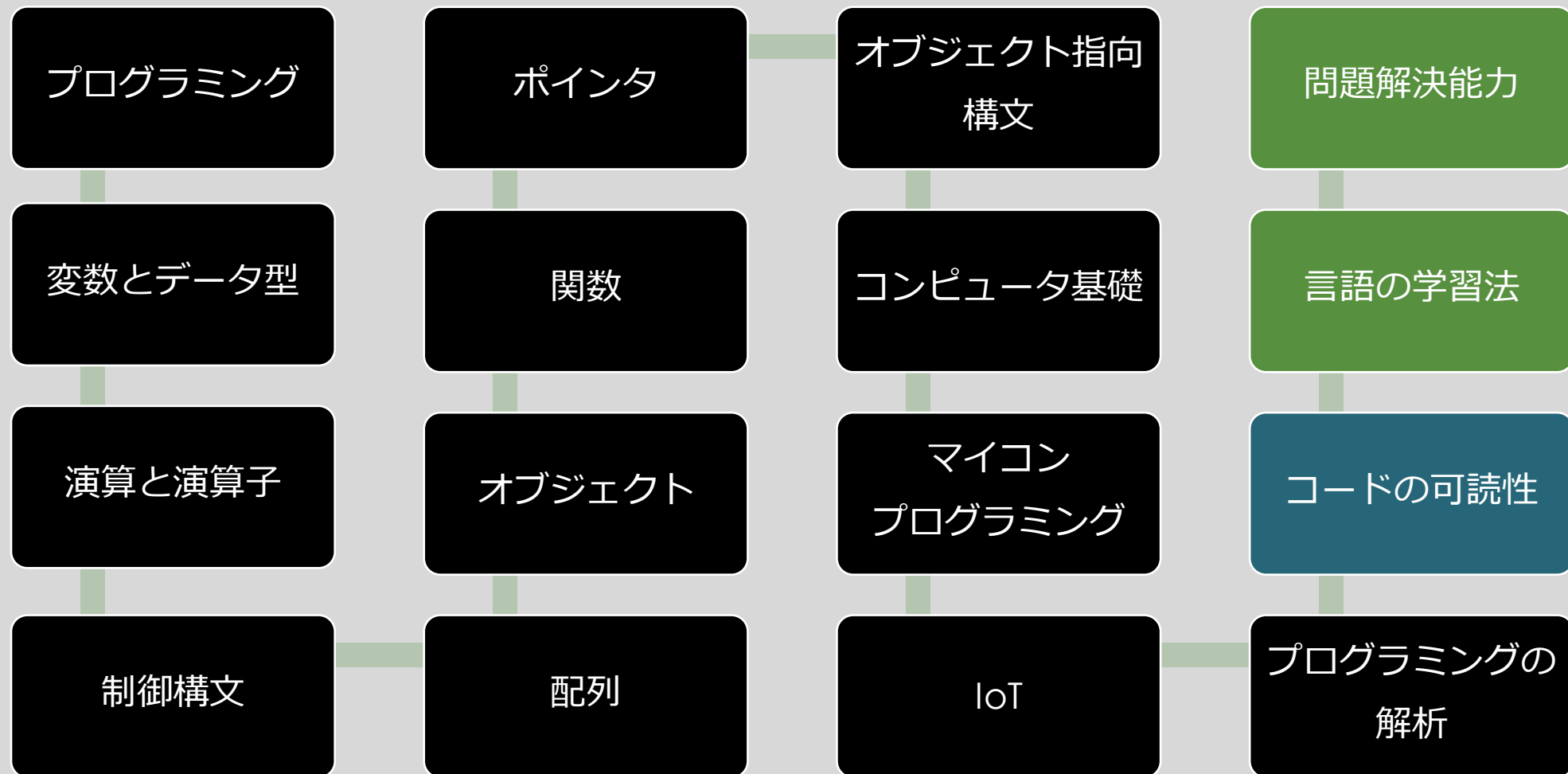
- パーツ分け、パーツ内解析、処理の把握、改変を繰り返し、
目的の動作へ近づけていく
- 論理的思考を用いた道筋に基づき処理の流れを理解する
- 動作や処理の予測と、実際に動かしたときの確認がキーポイント
- 大きなソースコードを、小さく区切って考えていく



コードの可読性

～読みやすいコードのために～

進度



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or2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)

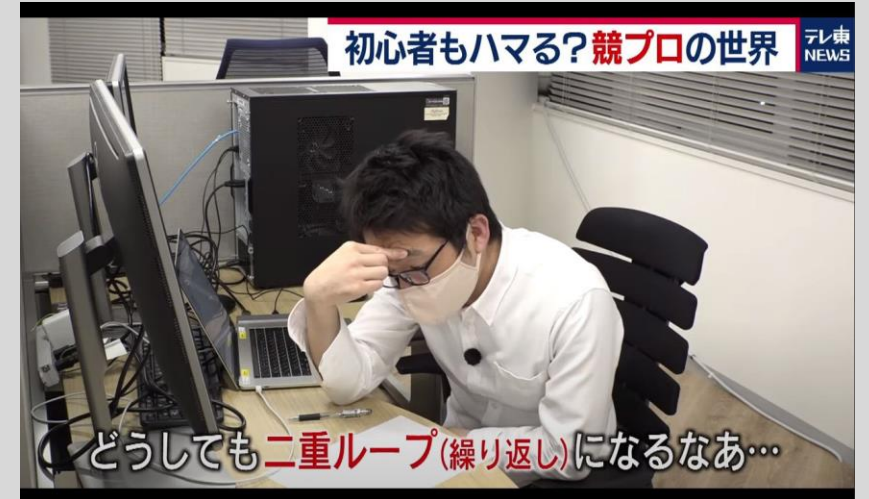
コメント文



- 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



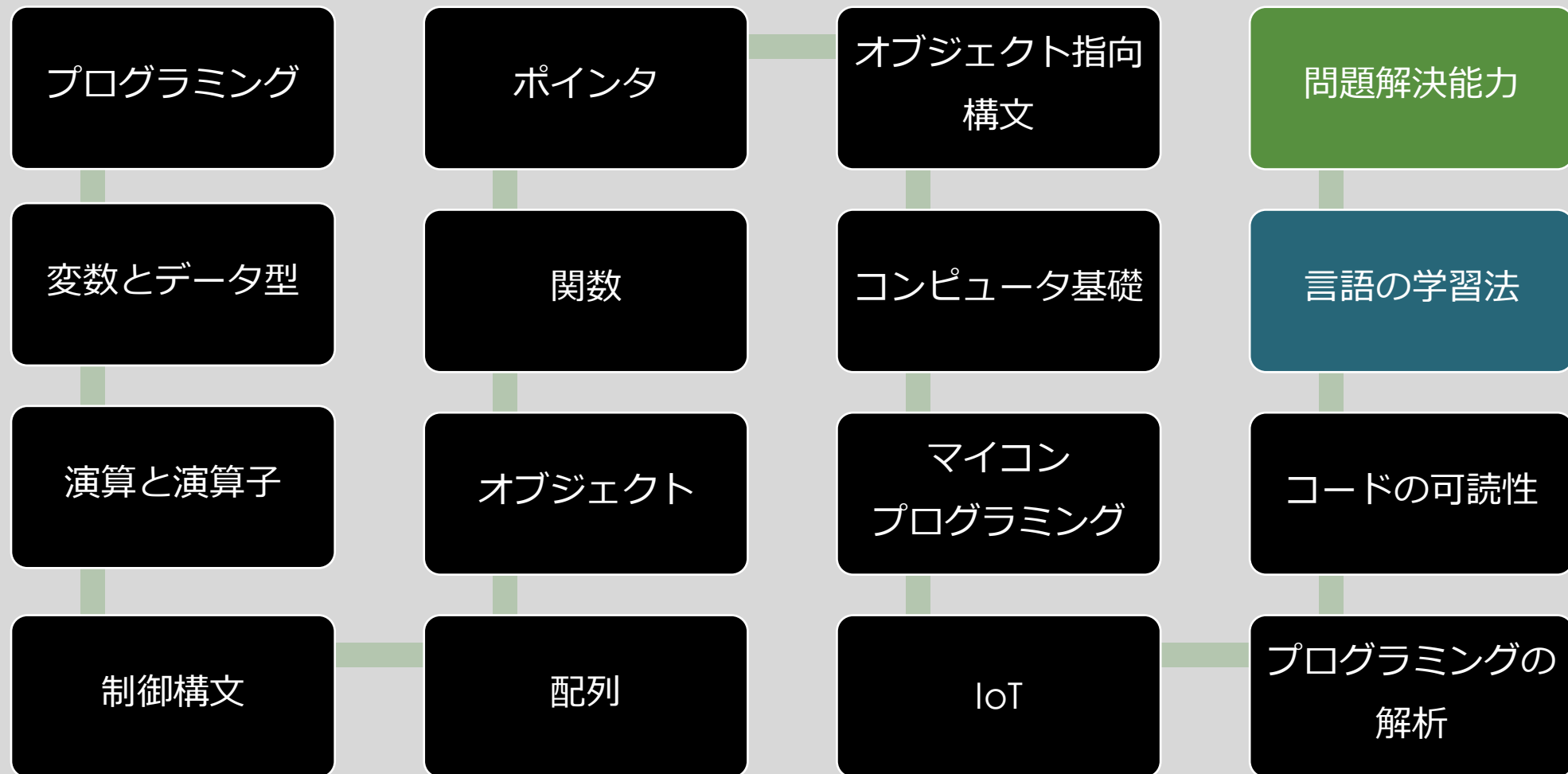
- コードを読みやすくするためには、**可読性**を意識する必要がある
- 記述**の可読性
 - 命名**や**演算子**、**インデント**、**コメント**、**ネスト**などに
注意して記述する必要がある
- 処理**の可読性
 - 関数**や**クラス**、**配列**、**オブジェクト**、**ファイル分け**や**ライブラリ**などに
注意して記述する必要がある



言語の学習法

～いろんなところに適応しよう～

進度



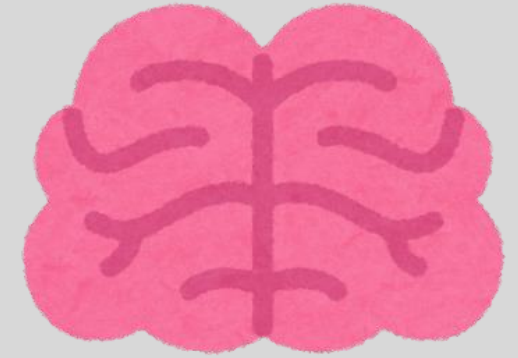
言語習得の手順

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



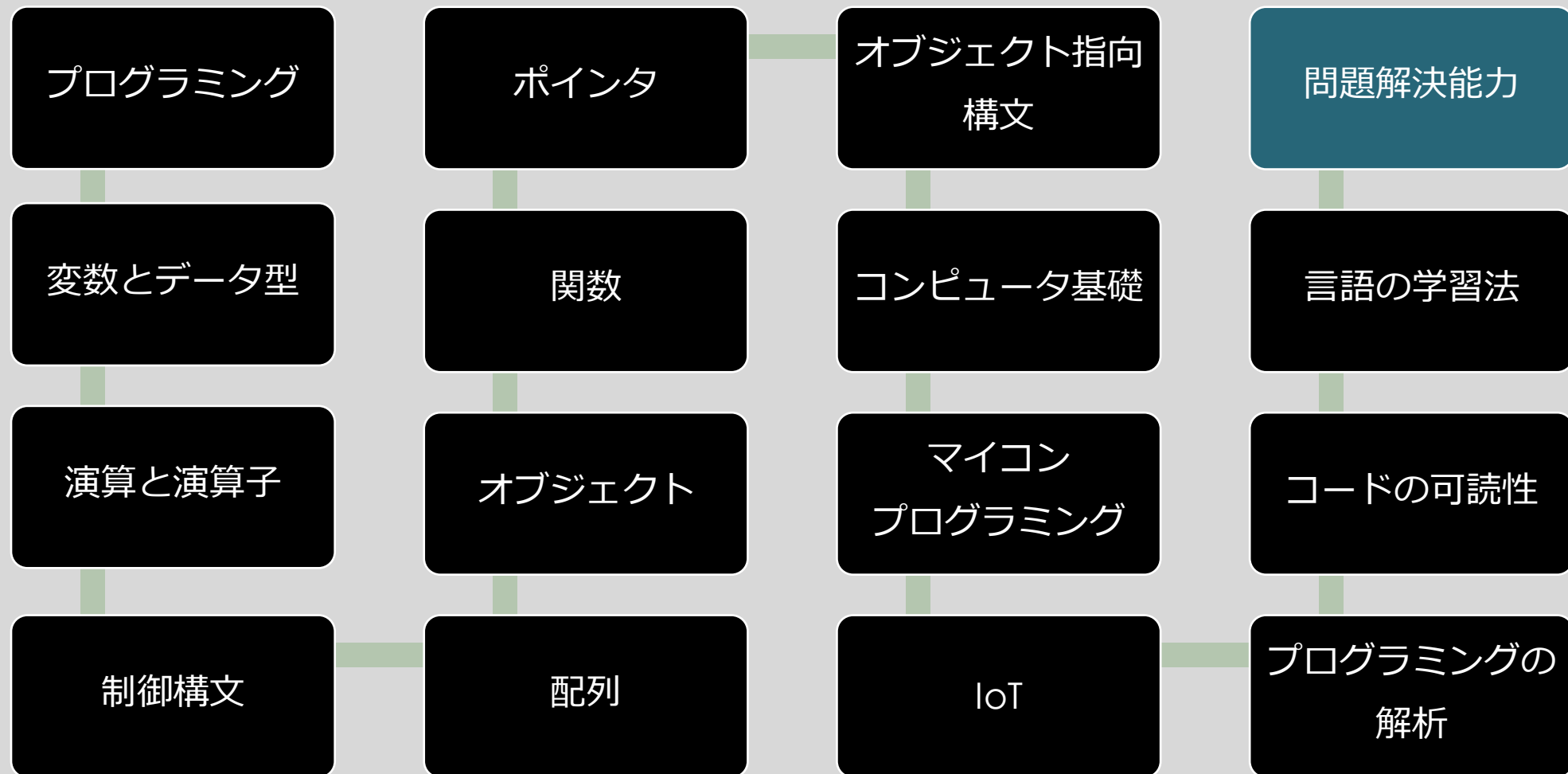
- 記述方法、基本の骨格を知り
それを活かしてプログラミング解析を重ねて言語習得をする
- モノづくりではエラーを対処するだけでなく
課題の本質理解から真の問題解決を実現することが重要
- 問題解決を繰り返すことで予測と確認の手順が身に付き
最も効率の良い言語習得法が完成する



問題解決能力

～全体のまとめ～

進度



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！

参考文献

- <https://eng-entrance.com/linux-shellscript-variable>
- <https://xtech.nikkei.com/it/atcl/column/14/091700069/091700002/>
- <https://wa3.i-3-i.info/diff446programming.html>
- <http://www.b.s.osakafu-u.ac.jp/~hezoe/pro/chapter5.html>
- https://kanda-it-school-kensyu.com/php-super-intro-contents/psi_ch09/psi_0905/
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch06/jbi_0606/
- <https://itmanabi.com/structured-objectoriented-prog/>
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch07/jbi_0704/
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch07/jbi_0703/
- https://kanda-it-school-kensyu.com/java-basic-intro-contents/jbi_ch07/jbi_0702/