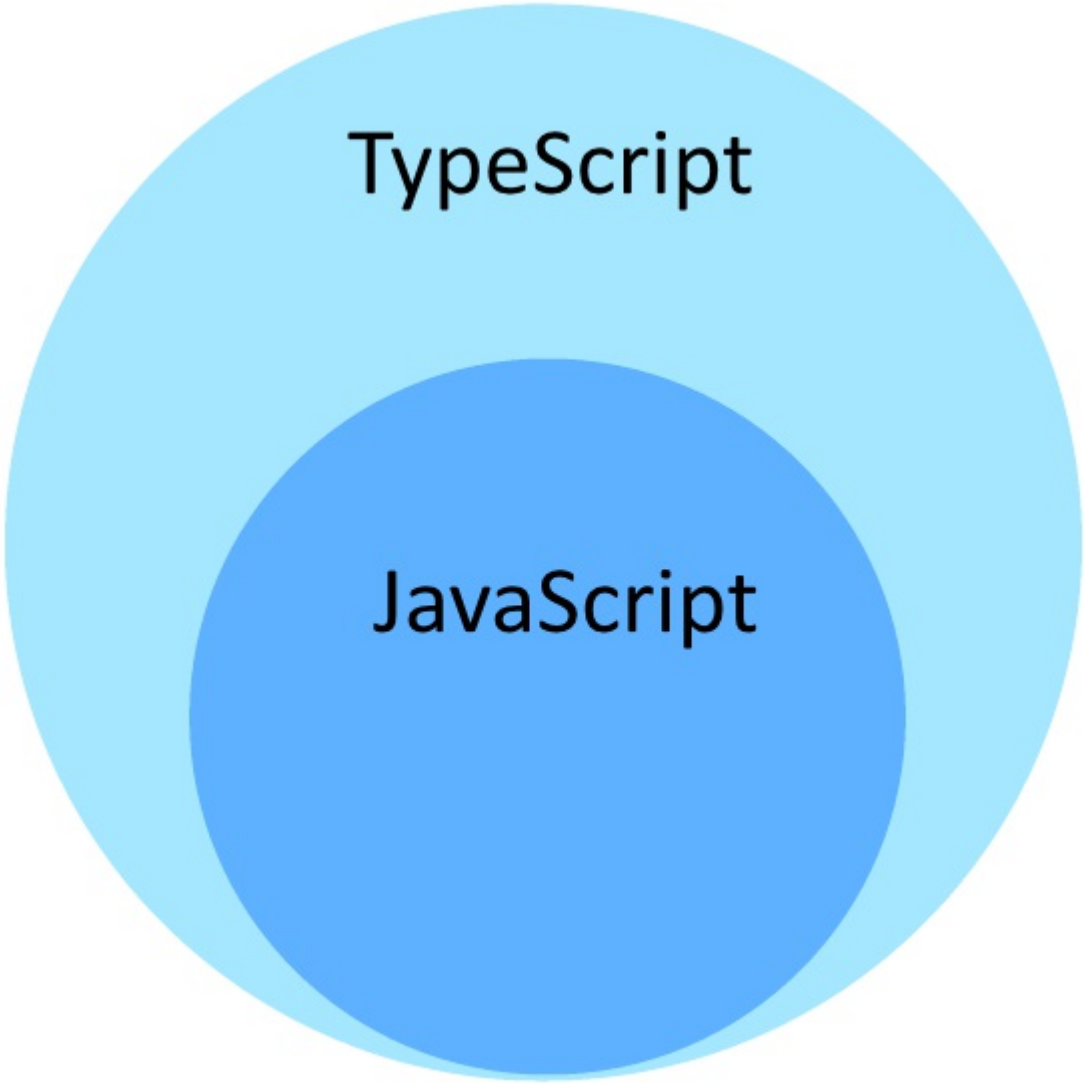
TypeScript

TypeScript とは

1. TypeScriptはJavaScriptのスーパーセットである



既存のJavaScriptプログラムは、全て有効なTypeScriptプログラムとなる

2. 静的型付けとクラスベースオブジェクト指向

3. TypeScriptは大規模なアプリケーションの開発のために設計されている

4. オープンソース

5. JavaScriptにコンパイルされる

6. TypeScriptでは変数の宣言時にデータ型を指定できる

Hello World

簡単なHello Worldプログラムを作ってみよう

```
const message: string = 'Hello World!'
console.log(message) //ブラウザでF12を押す、consoleで確認できる
alert(message) //alertが出る
```

Playground
(http://www.typescriptlang.org/play/#src=const%20message%3A%20string%20%3D%20'Hello%20World!'%3B%0D%0Aconsole.log(messag
でコンパイル

変数と宣言

```
const message: string = 'This is a book'
----------------------------------------
const: 変数を宣言するためのキーワード
message: 変数名
string: データ型
```

Var

従来の JavaScript での変数宣言には var キーワードが使われてきました。

```
function f() {
  var x = 1
  console.log(x)
  {
    var x = 2
    console.log(x)
  }
  console.log(x)
}
f()
// 1
// 2
// 2
```

Let

Letを使った場合、ブロックレベルで変数のスコープが定義される

```
function f() {
  let x = 1
  console.log(x)
  {
    let x = 2
    console.log(x)
  }
  console.log(x)
}
f()
// 1
// 2
// 1
```

const

read only

```
const x: string = 'const'
x = 'change' // error
```

let, const, var, のスコープの違い

```
(function () {
  if (true) {
    var hoge = 'hoge'
    let fuga = 'fuga'
    const piyo = 'piyo'

    console.log(hoge) //hoge
    console.log(fuga) //fuga
    console.log(piyo) //piyo
  }

  console.log(hoge) //hoge
  console.log(fuga) //ReferenceError: fuga is not defined
  console.log(piyo) //ReferenceError: piyo is not defined
}());
```

基本の型

1. boolean

真偽値（trueまたはfalseのいずれかの値）

in TypeScript

```
const done: boolean = true
```

in JavaScript

```
var done = false
```

2. number

数値はすべて浮動小数点として扱います。
16 進数、10 進数に加えて、8 進数、2 進数もサポートしてます。

in TypeScript

```
const decimal: number = 6
const hex: number = 0x2537
const binary: number = 0b1010
const octal: number = 0o22467

const numbers:number = '123' //エラー
```

in JavaScript

```
var decimal = 6
var hex = 0x2537
var binary = 10
var octal = 0o22467

var number = '123' //ok
```

3. String

文字列はダブルクォート(")またはシングルクォート(')で囲みます。

in TypeScript

```
const message: string
message = 'Happy BirthDay'

const strings: string = 1 //エラー
```

in JavaScript

```
var message = 'Happy new year'
var strings = 1 //ok
```

4. Array(配列)

in TypeScript

```
let Array1: number[] = [1,2,3]
let Array2: Array<number> = [4,5,6]
```

in JavaScript

```
var array =  [7,8,9]
```

5. Enum(列挙型)

Numeric enum

in TypeScript

```
enum Color {
  red=1,
  green,
  blue}
// red = 1 , green = 2 , blue = 3
const sky: Color = Color.blue
// sky = 3
alert(sky)
```

Playground
(http://www.typescriptlang.org/play/#src=enum%20Color%20%7B%20red%20%3D%201%2C%20green%2C%20blue%20%7D%3Bconst%
でコンパイル

in JavaScript

```
var Color;
(function (Color) {
    Color[Color["red"] = 1] = "red";
    Color[Color["green"] = 2] = "green";
    Color[Color["blue"] = 3] = "blue";
})(Color || (Color = {}));
// red = 1 , green = 2 , blue = 3
var sky = Color.blue;
// sky = 3
alert(sky);
```

上記の例は red が 1 として初期化されている数値 enum となります。
他のメンバの値は自動的にインクリメントして定義されます。
つまり、Color.red は 1、green は 2、blue は 3となります。

String enums

```
enum Direction {
    Up = 'UP',
    Down = 'DOWN',
    Left = 'LEFT',
    Right = 'RIGHT',
}

const key: Direction = Direction.Down
console.log(key) //'DOWN'
console.log(typeof key) //string
```

Playground
(http://www.typescriptlang.org/play/#src=enum%20Direction%20%7B%20Up%20%3D%20'UP'%2C%20Down%20%3D%20'DOWN'%2C%2
でコンパイル

6. Date

TypeScript で現在の日付時刻を格納する変数nowを得るには以下のように記述

```
const now = new Date()
const hour = now.getHours()
console.log('今は' + hour + '時')
const minute = now.getMinutes()
console.log('今は' + minute + '分')
const second = now.getSeconds()
console.log('今は' + second + '秒')
const year = now.getFullYear()
console.log('今日は' + year + '年')
const month = now.getMonth() +1
// new Date の第 2 引数でも、Date.prototype.getMonth でも、月は 0-11で表すので、ここは+1。
console.log('今日は' + month + '月')
const date = now.getDate()
console.log('今日は' + date + '日')
const day = now.getDay()
console.log('今日は' + day + '曜日')
const today = now.toDateString()
console.log(today)
// 曜日 月 日 年 並び 例:Wed May 30 2018
const today2 = now.toJSON().slice(0, 10)
console.log(today2)
//yyyy-MM-dd 並び 例:2018-5-30
```

Playground
(http://www.typescriptlang.org/play/#src=const%20now%20%3D%20new%20Date()%3B%0D%0Aconst%20hour%20%3D%20now.getHours
でコンパイル

7. Any

Any 型を使えば型チェックをバイパスすることができます。

変数のデータ型を指定しない場合は「any」が指定されたものと見なされる。

```
let notSure: any = 4
console.log(typeof notSure) //number
notSure = 'maybe a string instead'
console.log(typeof notSure) //string
notSure = false
console.log(typeof notSure) //boolean

//typeof 変数名は変数の型をチェックすることができる
```

Playground
(http://www.typescriptlang.org/play/#src=let%20notSure%3A%20any%20%3D%204%3B%20console.log(typeof%20notSure)%3BnotSure%
でコンパイル

Any 型は他の型の一部としても使えます。

```
let list: any[] = [1, true, 'free']
// 任意の型を含められる
```

TypeScriptでは宣言時に変数の値を代入（初期化）しておくと、変数のデータ型が自動的に推測されて決められる

```
let price // any型
price = 1000
console.log(typeof price) // number
price = '無料' // 文字列も代入できる

let value = 5000 // number型と推測される
value = '高い' // 文字列を代入しようとするとエラーになる
```

Playground
(http://www.typescriptlang.org/play/#src=let%20price%3Bprice%20%3D%201000%3Bconsole.log(typeof%20price)%3Bprice%20%3D%20'
でコンパイル

8. Void

Void は型がないことを表すもので、値を返さない関数の戻り値でよく使います。

```
function warnUser(): void {
    alert('This is my warning message')
}
```

Void 型には undefined か null しか代入できません。

```
const unusable: void = undefined
```

9. Nullと Undefined

Null 型と Undefined 型にはそれぞれの値しか代入できません。

```
const u: undefined = undefined
const n: null = null
```

10. Tuple

タプル型は、要素の個数・型が決められた配列を表現することを可能にします。
例えば、あなたが文字列と数値のペアを値として表現したいとした場合は次のようにします。

```
// タプル型の宣言
let x: [string, number];
// 初期化
x = ["hello", 10]; // OK
// 不適切な初期化
x = [10, "hello"]; // Error
```

式と演算

1. 簡単な演算

以下のコードで割引後の金額が計算できる

```
let price,discount,total:number
price = 1000
discount = 0.75
total = price*discount
alert(total)
```

Playground
(http://www.typescriptlang.org/play/#src=let%20price%2C%20discount%2C%20total%3A%20number%3B%0Aprice%20%3D%201000%3E
でコンパイル

演算とは、広い意味で「計算すること」と考えていいが、変数に値を入れる代入も演算であることに注意しよう。
また、演算に使う記号のことを「演算子」と呼ぶ。ここでは、「=」や「*」「+」が演算子だ。

2. 論理演算子

①論理演算子！

英語notの意味。

```
const a = !0
alert(a) // true
const b = !!0
alert(b) //false
```

②理論演算子 &&

英語andの意味

```
const a = true && true
alert(a) // true
const b = true && false
alert(b) // false
const c = true && (4 > 3)
alert(c) // true
```

③理論演算子 ||

英語orの意味

```
const a = true || true
alert(a) //true
const b = true || false
alert(b) //true
const c = false || false
alert(c) //false
const d = (4<3)|| 'cat'
alert(d) //cat
```

3. インスタンス作成

```
class SmartPhone{
  brand: string
  model: string
  generation: number
}

const i8 = new SmartPhone()
i8.brand = 'apple '
i8.model = 'iPhone '
i8.generation = 8

alert('I have the new' + ' ' + i8.brand + i8.model + i8.generation)
```

Playground
(http://www.typescriptlang.org/play/#src=class%20SmartPhone%7B%0D%0A%20%20%20%20brand%3A%20string%3B%0D%0A%20%20
でコンパイル

i8とbrandを区切る「.」に注目してほしい。「.」はクラスのメンバーを参照するための演算子である。
「new」も演算子の1つであり、クラスのインスタンスを作成し、その参照を返す働きを持っている。
「+」という演算子が使われているが、これは最初に見たような数値の加算ではなく、文字列を連結するという働きを持つ。

4. 二項演算子 +

string + number = string

```
const a: string = '12'
const b: number = 34
const c = a + b
alert(c) // 1234
alert(typeof c) // string
window.close()
```

Playground
(http://www.typescriptlang.org/play/#src=const%20a%3A%20string%20%3D%20'12'%3B%0D%0Aconst%20b%3A%20number%20%3D%2(
でコンパイル

string + boolean = string

```
const a: string = '12'
const b: boolean = true
const c = a + b
alert(c) // 12true
alert(typeof c) // string
window.close()
```

Playgroud
(http://www.typescriptlang.org/play/#src=const%20a%3A%20string%20%3D%20'12'%3B%0D%0Aconst%20b%3A%20boolean%20%3D%2
でコンパイル

5. 比較演算子

<,>,<=,>=,==,!=,===,!==

```
const a: number = 12
const b: string = '12'
const c: any = '12'
const d: any = 34

console.log(a > d) // false
console.log(a !== b) // compile error
console.log(a !== c) // true
console.log(b === c) // true
```

比較演算子の落とし穴

```
18 <= x < 25 //誤った例
18 <= x && x < 25 //正しい書き方
```

複数の条件を調べたいときには、論理積を求める「&&」演算子や論理和を求める「||」演算子を使う。

6. 代入演算子

```
let a, b, c, d: number
a = 1
b = 2
c = 4
d = 8

console.log(a += b) // a = a + b
console.log(c /= b) // c = c / b
console.log(b *= d) // b = b * d
```

7. 演算子の結合方向

例えば、代入演算子の「=」や「+=」は「←（右から左）」となっているので、右から左に向かって演算が実行される。

```
let a , b :number
b = 7
a = b+=2 // 「b += 2」が実行された後、「a = b」が実行される
alert ('a='+ a + 'b=' + b)
```

Playground (http://www.typescriptlang.org/play/#src=let%20a%2C%20b%3A%20number%3B%0D%0Ab%20%3D%207%3B%0D%0Aa%20%3D%20b%) でコンパイル

「+=」は、変数の値に右辺の値を加算した値を変数に代入するための演算子である

```
// 以下の2つは同じ意味
b = b + 2
b += 2
```

8. インクリメントとデクリメント

①前置型インクリメント演算子

```
let a : number = 99
alert(++a) //100
```

Playground (http://www.typescriptlang.org/play/#src=let%20a%3A%20number%20%3D%2099%3B%0D%0Aalert(%2B%2Ba)%3B) でコンパイル

②後置型デクリメント演算子

```
const b : number = 17
alert(b--) //17
alert(b) //16
```

Playground (hhttp://www.typescriptlang.org/play/#src=let%20b%3A%20number%20%3D%2017%3B%0D%0Aalert(b--)%3B%0D%0Aalert(b)%3B) でコンパイル

前置型ー変数の値が増やされたあと、変数の値が返される

後置型ー変数の値は増やされるが、返される値は元の値

9. 特殊演算子

①typeof

変数の型がチェックできる

```
const a = 'Hello'
const b: typeof a = a
alert(b) // Hello
```

Playground
(http://www.typescriptlang.org/play/#src=const%20a%20%3D%20'Hello'%3B%0D%0Aconst%20b%3A%20typeof%20a%20%3D%20a%3B%
でコンパイル

②instanceof

インスタンスの関係かどうかチェックできる

```
function a() {
    console.log('Hello world')
}
const b = new a()
const c = b instanceof a
alert(c) // true
```

Playground
(http://www.typescriptlang.org/play/#src=function%20a()%20%7B%0D%0A%20%20%20%20console.log('Hello%20world')%3B%0D%0A%7
でコンパイル

条件分岐

1. if ... else文を使った例

式の値によって異なる文を実行する。if文を組み合わせ、異なる変数の値を調べて多分岐させることもできる

```
let a: number
function dize() {
 a =Math.floor(Math.random() * 6) + 1 // a = random number from 1-6
}
dize()
if (a>3) {
  alert( a + ' is big')
}
else {
  alert( a + ' is small')
}
```

[Playground]
(http://www.typescriptlang.org/play/#src=let%20a%3A%20number%3B%0Afunction%20dize()%20%7B%0A%09a%20%3D%20Math.floor(
でコンパイル

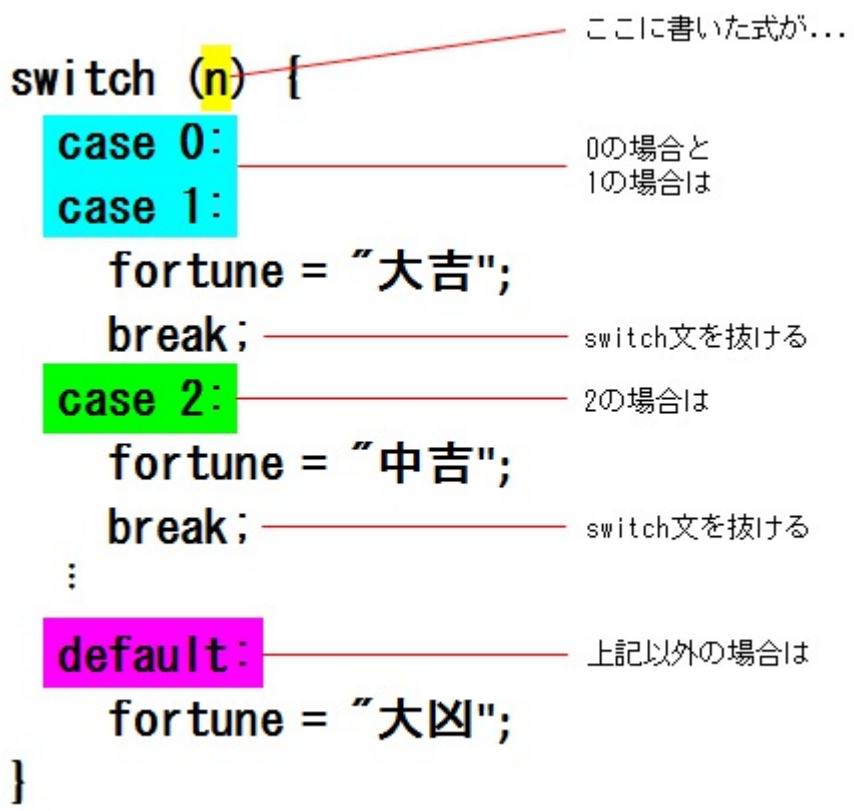
2. switch文を使った例

変数の値によって異なる文を実行する。1つの変数の値を調べて多分岐させるときに便利

```
let fortune: string
let n: number
n = Math.floor(Math.random() * 7)
switch (n) {
  case 0:
  case 1:
    fortune = '大吉'
    break
  case 2:
    fortune = '中吉'
    break
  case 3:
  case 4:
    fortune = '小吉'
    break
  case 5:
    fortune = '凶'
    break
  default:
    fortune = '大凶'
}
alert(n + ':' + fortune)
```

[Playground](http://www.typescriptlang.org/play/#src=let%20fortune%3A%20string%3B%0Alet%20n%3A%20number%3B%0An%20%3D%20Math.floo
でコンパイル

switch文の構造



1. switchの後の()の中の変数や式の値によって実行する文が変えられる
2. ()の中の変数や式の値がcaseの後に指定した値に一致すれば、それ以降の文が全て実行される
3. 1つのcaseの中には複数の文を書いてもよい
4. ただし、break文があると、そこでswitch文を抜けて、次の文に進む
5. defaultは他のcaseで指定していない全ての値に一致する（通常は「上記以外の場合」を表すために、最後に書かれる）

3. ?演算子

「条件式？式1：式2」という形、条件を満たしたら式1になる、満たさないければ式2になる。

```
const score = 59
const pass: string = (score >= 60 ? '合格' : '不合格')
alert(pass) // 不合格
```

Playground
(http://www.typescriptlang.org/play/#src=const%20score%20%3D%2059%3B%0D%0Aconst%20pass%3A%20string%20%3D%20(score%20
でコンパイル

繰り返し処理

条件を満たしている間、同じ文を繰り返して実行したり、一定の回数だけ文を繰り返し実行したりする

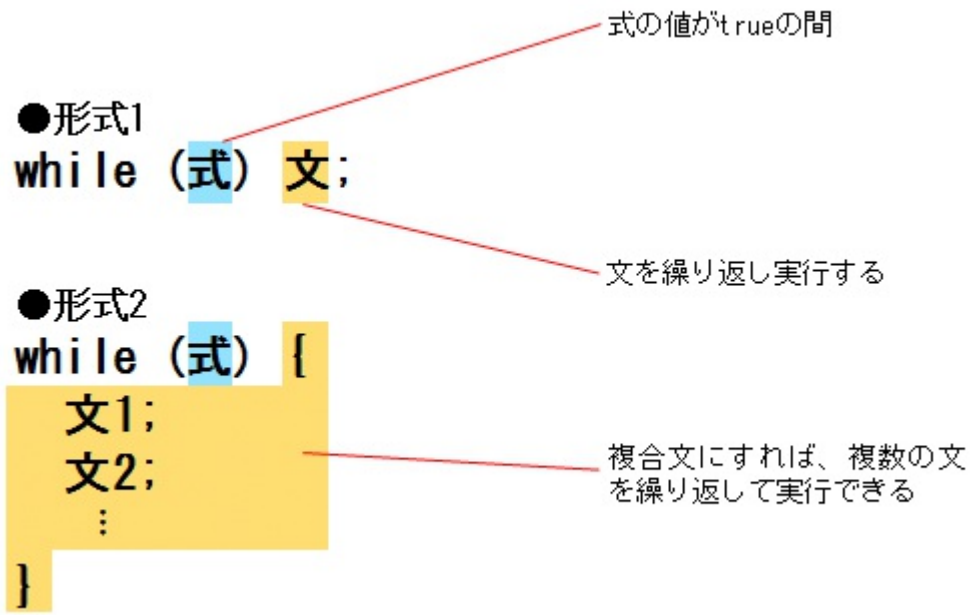
1. while 文

式の値がtrueである間、文を繰り返し実行する。式の値を毎回の繰り返しの前に判定する（前判断型while）。
以下のプログラムはサイコロを振ったときに、6が出るまでに何回かかったかを表示する。つまり、6以外の目が出ている間、サイコロを振り続ける（処理を繰り返す）というわけだ.

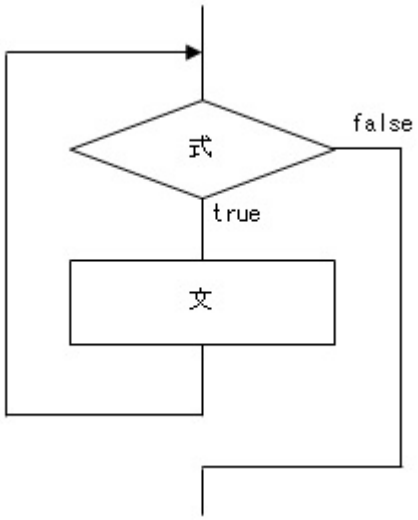
```
let count: number = 1
let dice: number = Math.floor(Math.random() * 6) + 1
const list = new Array()
while (dice != 6) {
    count++
    list.push(dice)
    dice = Math.floor(Math.random() * 6) + 1
}
alert('[' + list + ']' + ' 6が出るまで' + count + '回')
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=let%20count%3A%20number%20%3D%201%3B%0Alet%20dice%3A%20number%3B%0Adice%
でコンパイル

while文の構造



while文の流れ



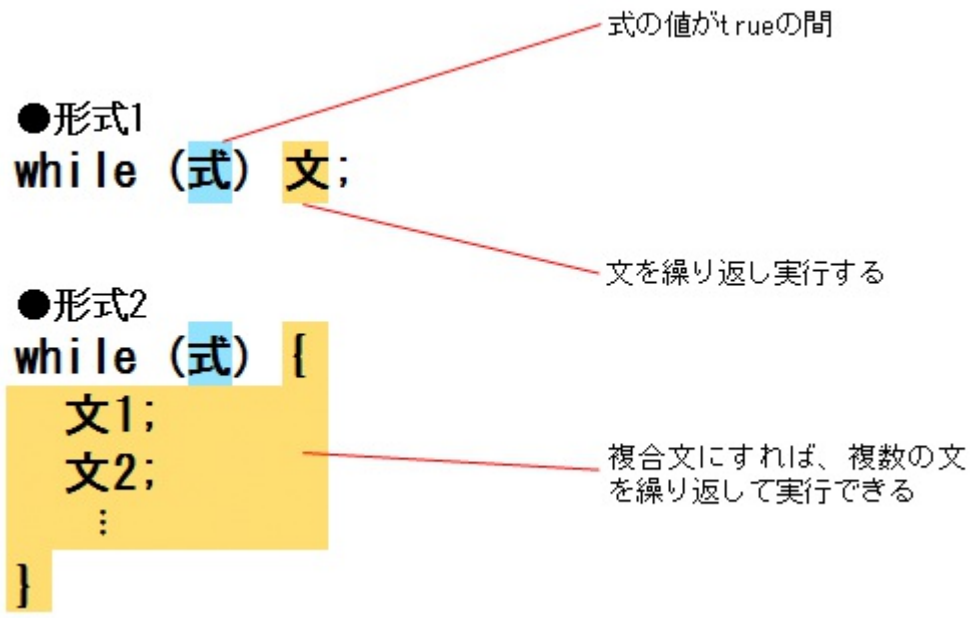
2. do...while文

式の値がtrueである間、文を繰り返し実行する。
式の値を毎回の繰り返しの後に判定する（後判断型while）
同じサイコロの例。

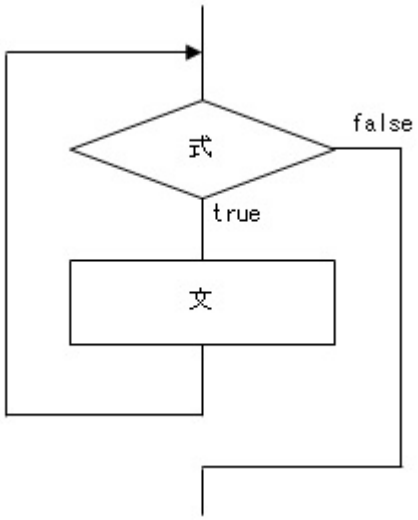
```
let count: number = 0
let dice: number
const list = new Array()
do {
    dice = Math.floor(Math.random() * 6) + 1
    list.push(dice)
    count++
} while (dice != 6)
alert('[' + list + ']' + ' 6が出るまで' + count + '回')
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=let%20count%3A%20number%20%3D%200%3B%0Alet%20dice%3A%20number%3B%0Aconst%
でコンパイル

do...while文の構造



do...while文の流れ

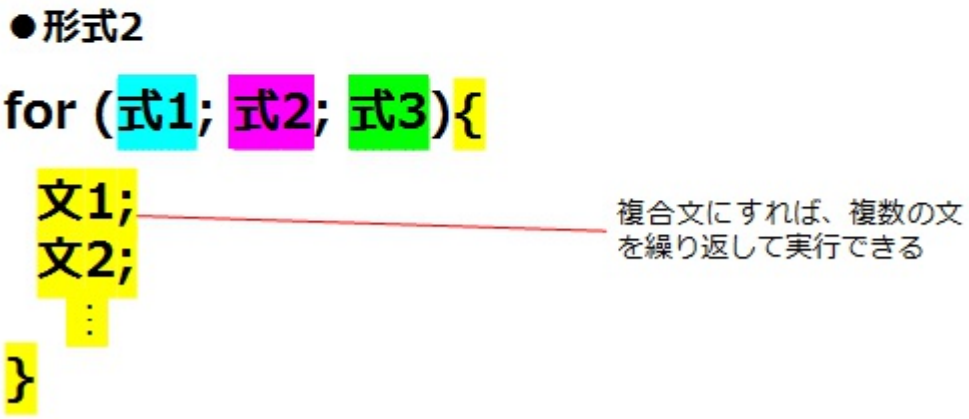


3. for文

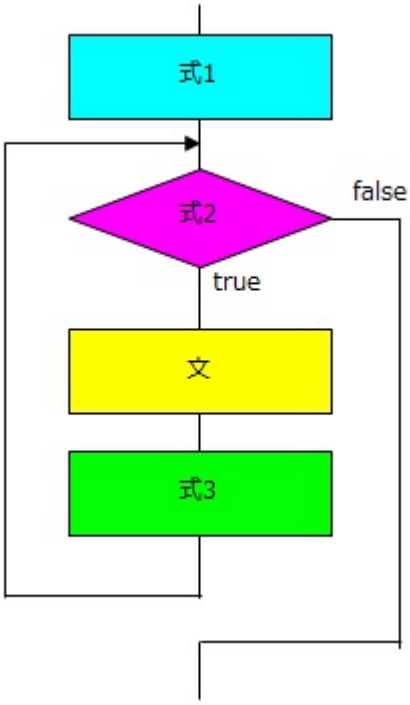
初期設定、繰り返しの条件、毎回の繰り返しの後に実行する処理をまとめて書ける文。
一定回数の繰り返し処理によく使われる。

```
let a = 0
for (let i = 1; i < 10; i++){
    a += i // a = 1+2+3+...+9
}
alert(a) // 45
```

for文の構造



for文の流れ

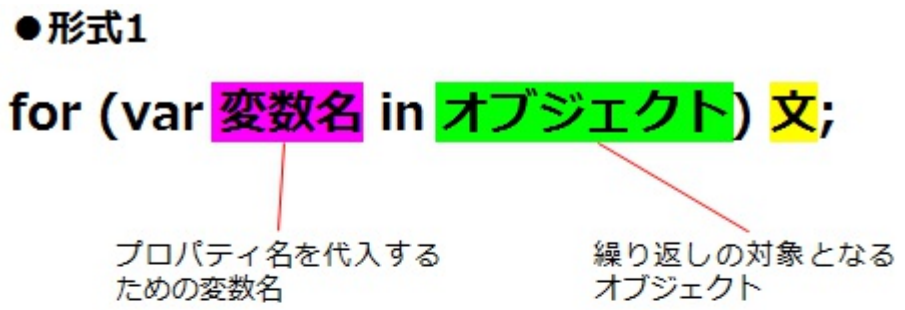


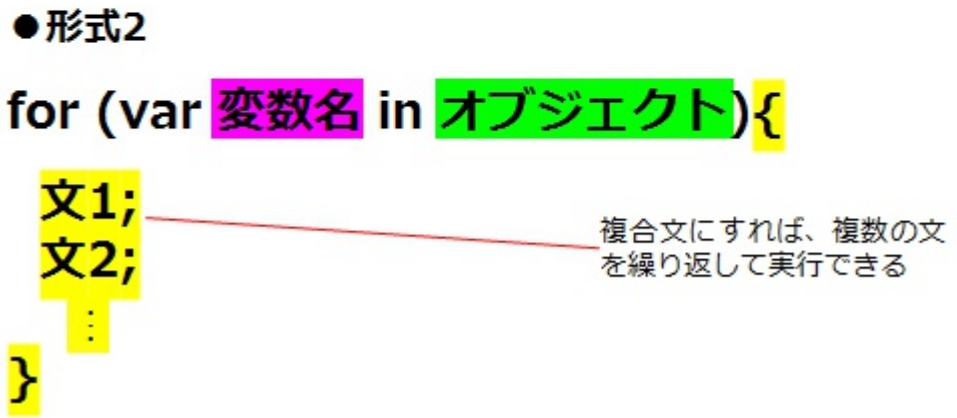
4. for...in文

オブジェクトの全てのプロパティの値に対して繰り返し処理を行う

```
class Smartphone {
    name:string
    camera:string
    screenSize:string
}
const i:Smartphone = new Smartphone()
i.name = 'iPhone8'
i.camera = '1200Mpx'
i.screenSize = '4.7"'
document.body.innerHTML = 'スマホ仕様<br/>'
for (const x in i) {
    document.body.innerHTML += x + ':' + i[x] + '<br/>'
}
```

for...in文の構造





for...of V.S. for...in

for..ofとfor..in文は、どちらもリストを繰り返し処理するものですが、その繰り返す値が異なり、
for..inはオブジェクトのキーのリストを返しますが、for..ofはオブジェクトの数値プロパティの値のリストを返します

```
let list = [4, 5, 6];
for (let i in list) {
   console.log(i); // "0", "1", "2",
}
for (let i of list) {
   console.log(i); // "4", "5", "6"
}
```

配列

配列を宣言するには、全ての要素を代表するような変数名を一つ付けておけばよい。
配列の個々の要素はインデックスと呼ばれる番号で区別する。

基本配列と出力

```
let carBrandList: string[] = new Array(4) // 4つ入りの配列を宣言し、中身は文字列
carBrandList = ['Audi', 'Benz', 'BMW', 'Lexus']

console.log(carBrandList) // 0:'Audi', 1:'Benz', 2:'BMW', 3:'Lexus' Length=4
// 配列を出力

for (let brand of carBrandList) {
    console.log(brand) // Audi Benz BMW Lexus
} // 配列の内容を一つずつ出力

console.log(carBrandList[2]) // BMW
// 指定した配列中身の順番を出力
```

Playground
(http://www.typescriptlang.org/play/#src=let%20carBrandList%3A%20string%5B%5D%20%3D%20new%20Array(4)%3B%0D%0AcarBrandL
でコンパイル

配列内容の追加と削除

```
let carBrandList: string[] = new Array()
carBrandList = ['Audi', 'Benz', 'BMW', 'Lexus']

carBrandList.splice(3) // carBrandList[3]を削除

carBrandList.push('Volks') // Volksを配列の最後尾に追加

console.log(carBrandList)

for (let brand of carBrandList) {
    console.log(brand)
}

console.log(carBrandList[2])
```

Playground
(http://www.typescriptlang.org/play/#src=let%20carBrandList%3A%20string%5B%5D%20%3D%20new%20Array()%3BcarBrandList%20%3D
でコンパイル

配列と繰り返し処理

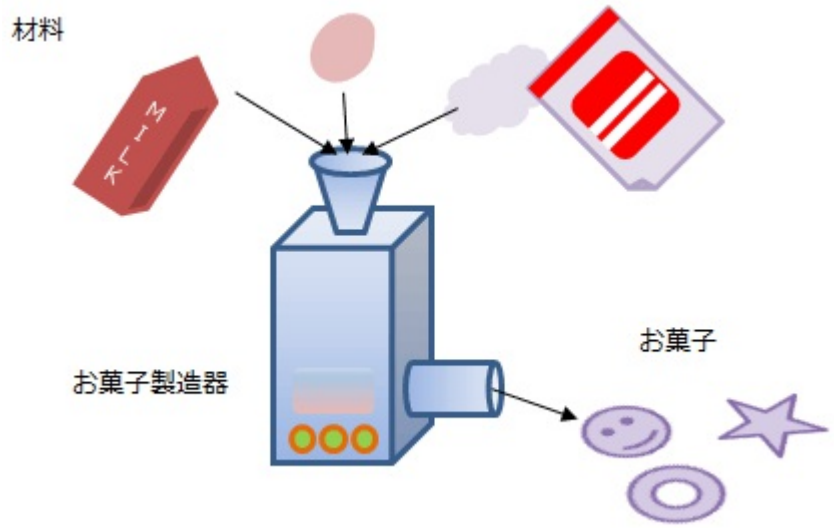
```
const board: number[] = new Array( 10 )
let temp, r1, r2: number
for ( let i = 0; i < 10; i++ ) {
    board[i] = i+1
}
for ( let count = 0; count < 50; count++ ) {
    r1 = Math.floor( Math.random() * 10 )
    r2 = Math.floor( Math.random() * 10 )
    temp = board[r1]
    board[r1] = board[r2]
    board[r2] = temp
}
alert( board )
```

[Playground]
(http://www.typescriptlang.org/play/#src=const%20board%3A%20number%5B%5D%20%3D%20new%20Array(%2010%20)%3B%0D%0Ale
でコンパイル

関数

関数とは

関数とはひとまとまりの処理を記述して名前を付けたもの。
関数の名前を指定し、必要に応じて値を与えてやれば、処理が実行され、結果が返される。
日常の例えでいえば、小麦粉や卵などの材料を入れるだけでお菓子を作ってくれる機械のようなイメージだ。



関数の基本的な考え方

関数とは、引数を与えれば、戻り値を返してくれるひとまとまりの処理のこと。
「関数」の中で何が行われているかが分からなくても、値を与えてやるだけで結果が得られる。

関数の大きな利点は、以下の2点である。

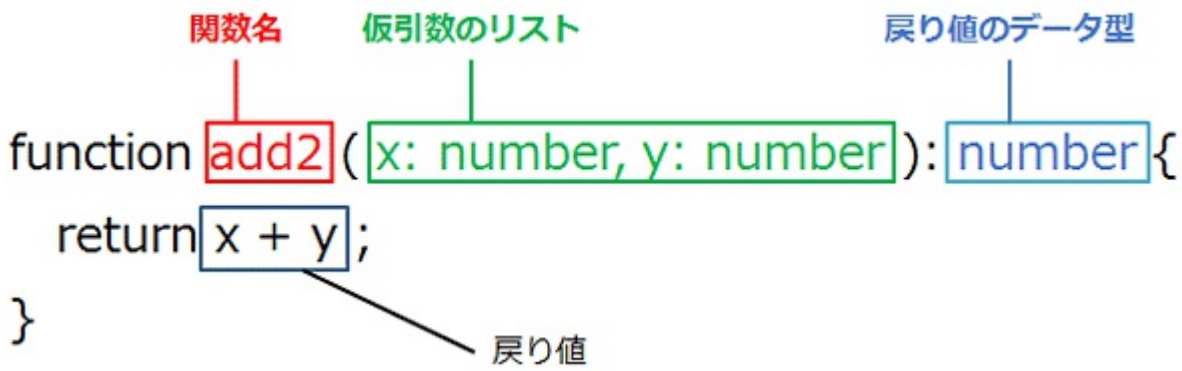
1. 一度関数を書いておけば、内部でどういう処理をしているかを詳しく知らなくても利用できる
2. 必要な箇所で何度でも利用できる

簡単な関数の書き方

```
function add2(x: number, y: number): number {
    return x + y
}
// 関数を定義
const answer: number = add2(19, 37) // 関数を呼び出す
alert(answer) // 56
window.close()
```

Playground
(http://www.typescriptlang.org/play/#src=function%20add2(x%3A%20number%2C%20y%3A%20number)%3A%20number%20%7B%0D%0
でコンパイル

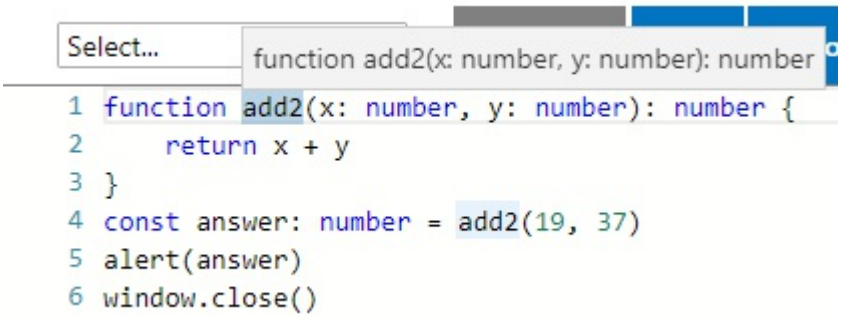
関数の書き方



変数answerを使わず、関数add2をalertメソッドの引数に直接指定してもよい

```
const answer: number = add2(10, 20)
alert(answer)
↓
alert(add2(10, 20))
```

関数オブジェクトを参照する変数のデータ型を確認する



関数オブジェクトを参照する変数は、関数の引数の並びとそれらのデータ型、戻り値のデータ型によって型が決まる。

アロー関数式を使う

「(引数のリスト) : 戻り値の型 => { 関数の処理 }」のような形式で書く

```
const mul2 = (a: number, b: number): number => {
  return a*b
}
alert(mul2(8,7)) // 56
```

[Playground]
(http://www.typescriptlang.org/play/#src=const%20mul2%20%3D%20(a%3A%20number%2C%20b%3A%20number)%3A%20number%20%
でコンパイル

戻り値が簡単な式の場合は、さらに簡略化できる。
「=>」の後に、戻り値を直接書けばよい。上の例をさらに書き換えてみよう。

```
const mul2 = (a: number, b: number): number => a * b

alert(mul2(8, 7))
```

一つの関数で複数の戻り値を返す方法

戻り値にオブジェクトを指定すれば、複数の値をまとめて返せる。
以下は金額の割引と税込みを計算する例。

```
function total(x: number, y: number) {
    const a = x * y // xは値段 yは割引
    const b = a * 1.08 // 税込みを計算
    return { price: a, taxin: b }
}

const iPhone8 = total(100000, 0.95)
alert('Price= ' + iPhone8.price + 'Tax in= ' + iPhone8.taxin)
// Price= 95000 Tax in= 102600
```

Playground
(http://www.typescriptlang.org/play/#src=function%20total(x%3A%20number%2C%20y%3A%20number)%20%7B%0D%0A%09const%20a
でコンパイル

関数の応用例

簡単なメモlistを作成してみよう

```
const txt = document.createElement('input') // input スペースを宣言
const btn = document.createElement('button') // ボタンを宣言
const memo = document.createElement('textarea') // textareaを宣言
const list: string[] = new Array()
btn.textContent = '押して'
btn.onclick = function () {
    if(txt.value!=''){
        alert(txt.value + 'を追加した')
        list.push(txt.value)
        txt.value = ''
        memo.value = list.toString()
    }
    else {
        alert('メモを入力してください')
    }
}
document.body.appendChild(txt) // inputスペースを画面で表示
document.body.appendChild(btn) // ボタンを画面で表示
document.body.appendChild(memo) // textareaを画面で表示
```

Playground
(http://www.typescriptlang.org/play/#src=const%20txt%20%3D%20document.createElement('input')%20%2F%2F%20input%20%E3%82%
でコンパイル

オプションの引数

関数を定義するときには、仮引数としてオプションの引数が指定できる。
簡単な例で見てみよう。単価(price)と数量(amount)を基に、金額を求める関数があるものとする。
ただし、メンバーランク(rank)が指定されている場合は、それだけ割り引くこととしよう。
つまり、メンバーランクは省略可能というわけだ。

```
function calCost(price: number, amount: number, rank?: string) {
  if (rank) { // もしランクが入力されてったら
    switch (rank) {
      case 'diamond':return price * amount *0.7
      case 'gold': return price * amount * 0.75
      case 'silver': return price * amount * 0.9
      default: return price * amount
    }

  }
  else {
    return price * amount
    }
}

alert(calCost(100, 3)) // 300
alert(calCost(9527, 1, 'diamond')) // 6668.9
alert(calCost(10000, 10, 'platina')) // platinaというランクはないので、100000
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=function%20calCost(price%3A%20number%2C%20amount%3A%20number%2C%20rank%3F%3
でコンパイル

ifの式はこう書くもよい

```
if(rank!=undefined)
```

引数の既定値を設定する

省略可能な引数には既定値が設定できる。

```
function circle(r: number, pi = 3.14) {
  const x = 2 * pi * r
  const y = pi * r * r
  return { perimeter: x, area: y }
}
const newCircle = circle(4)
alert('perimeter = ' + newCircle.perimeter + 'area = ' + newCircle.area)
```

Playground
(http://www.typescriptlang.org/play/#src=function%20circle(r%3A%20number%2C%20pi%20%3D%203.14)%20%7B%0D%0A%09const%2
でコンパイル

関数のオーバーロード

オーバーロードとは、同じ名前を持ち、異なる引数リストや戻り値の型を持つ複数の関数を定義すること

```
function getProfile(x: number): string
function getProfile(x: string): string
function getProfile(x: any): string {
  if (typeof (x) == 'string') {
    return x + 'のメンバー番号:1234'
     else {
    return '田中のメンバー番号は' + x
  }
}
alert(getProfile(1234)) //田中のメンバー番号は1234
alert(getProfile('田中')) //田中のメンバー番号:1234
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=function%20getProfile(x%3A%20number)%3A%20string%3B%0D%0Afunction%20getProfile(x%3
でコンパイル

ジェネリックス

ジェネリックスとは、データ型を仮に決めておき、
実際に使用するデータ型を呼び出し時に変えられるようにする機能で、総称型とも呼ばれる。
ジェネリックスを利用すると、データ型を関数の呼び出し時に決められる

```
function parrot<T>(data: T): T {
  let ret: T
  ret = data
  return ret
}
alert(parrot<number>(100)) // 100
alert(parrot<string>('Hello World')) // Hello World
alert(parrot<string>(123)) // データ型が合わないのでこれはエラーとなる
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=function%20parrot%3CT%3E(data%3A%20T)%3A%20T%20%7B%0A%09let%20ret%3A%20T%
でコンパイル

クロージャー

クロージャーとは、関数が定義された環境にある変数を利用できる機能

```
function getSerialNumber() {
  let origin = 0
  function countUp(delta: number): number {
    return origin += delta
  }
  return countUp
}
const inside = getSerialNumber()
alert(inside(2)) // 2
alert(inside(3)) // 5
alert(inside(-2)) // 3
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=function%20getSerialNumber()%20%7B%0A%09var%20origin%20%3D%200%3B%0A%09functi
でコンパイル

クラス

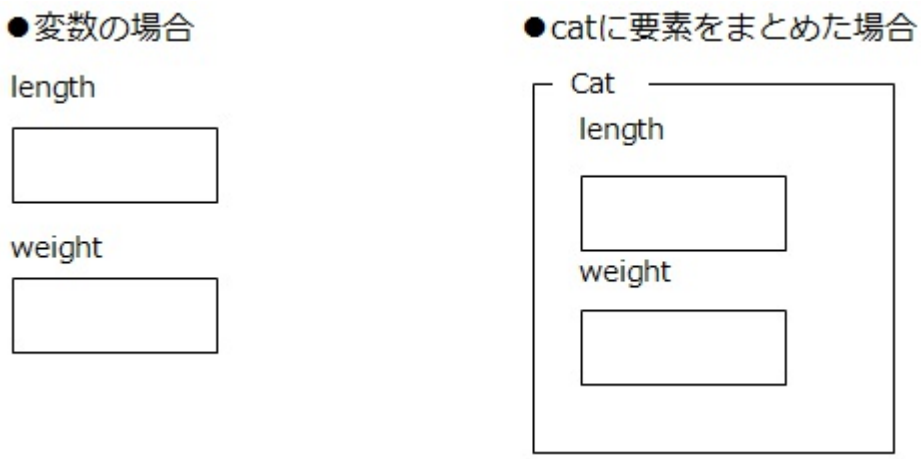
変数とオブジェクト

クラスを紹介する前、まず変数とオブジェクトを紹介する。
例えば、一匹の猫を表したいとき、基本は体長と体重が必要

```
let length: number
let weight: number
```

猫の体長と体重を表す変数を宣言した。だが、これではひと目見ただけでは猫には見えない。

変数とオブジェクトのイメージ



左の図のようにばらばらに変数が宣言されていると、まるで猫っぽくないが、
右の図のように猫の属性（体長や体重）をまとめてやれば、
現実の猫を目的に合わせてそのまま表現できる。

ある目的に従っていくつかの変数や関数をまとめたものを、
単なる変数と区別してオブジェクトと呼ぶ。

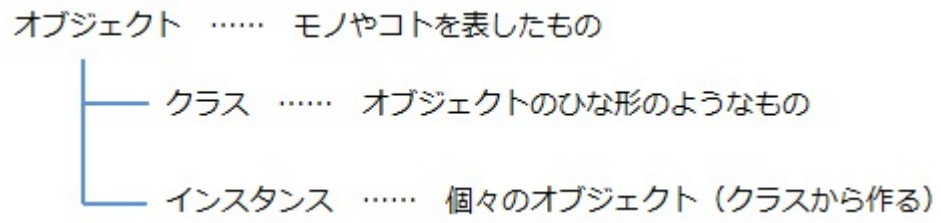
```
const nyCat = new Object()
myCat.length = 45 //cm
myCat.weight = 4 //Kg
```

このコードを見れば、変数だけを使っていた場合に比べて、これは猫だ、ということが確かに分かりやすい。
しかし、変数myCatが参照しているのは、汎用的に使われるObject型のオブジェクトであり、
他のオブジェクトと区別せずに使うこともできてしまう。

クラスを定義するには

TypeScriptでは、より厳密、かつ柔軟にオブジェクトが取り扱えるようになっている。
そのために使われるのがクラスである。クラスとは、個々のオブジェクトではなく、
オブジェクトのひな型とでもいうべきものである。

猫の例でいえば、「猫クラス」とは「一般的な猫」つまり、「猫ってこういうものだよ」という記述と考えていい。
それに対して、個々の猫は猫クラスを基に作る。その個々のオブジェクトはインスタンス（実体）と呼ばれる。
簡単にまとめると以下の図のようになる。



クラスを使って猫を表す

```
class Cat{
  length: number
  weight: number
}
```

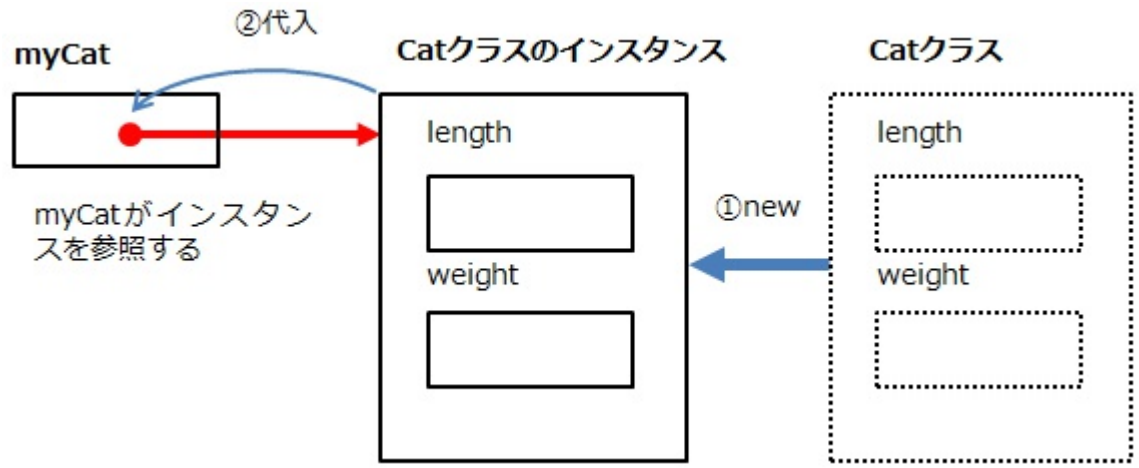
クラスの中に含まれる変数はプロパティと呼ばれ、関数はメソッドと呼ばれる。
プロパティとメソッドを合わせてクラスのメンバーと呼ぶ。

クラスからインスタンスを作成する

クラスは、いわばひな型のようなものなので、実際のデータを取り扱うにはインスタンスを作成する必要がある。
インスタンスとは「実体」とでもいうべきものである。
インスタンスの作成にはnew演算子を使う。クラスの定義も併せて書いておこう。

```
class Cat{
  length: number
  weight: number
}
cosnt myCat = new cat
```

newの後に「クラス名()」と書けば新しいインスタンスが作成できる。
インスタンスの参照が返されるのでそれを変数myCatに代入する。
従って、変数myCatを利用すれば、作成したインスタンスが利用できる。



インスタンス作成のイメージ

(1)new演算子を使って、クラスを基にインスタンスを作る。new演算子は作成されたインスタンスの参照を返す。
(2)インスタンスの参照を変数myCatに代入する。それにより、myCatを利用すればCatクラスのインスタンスが利用できるようになる。

クラスのメンバーを利用する

「.」で区切ってメンバーを書けば、クラスのメンバーが利用できる。

```
class Cat{
  name:string
  length: number
 weight: number
}

const myCat = new Cat()
myCat.name = 'mimi'
myCat.length = 45 //cm
myCat.weight = 4 //kg
alert(myCat.name + 'の体長は' + myCat.length + '体重は' + myCat.weight)
window.close()
```

Playground
(http://www.typescriptlang.org/play/#src=class%20Cat%7B%0A%20%20%20%20name%3A%20string%3B%0A%20%20%20%20length%3.
でコンパイル

クラスのメンバーとしてメソッドを定義する

これまでのCatクラスでは「猫とは体長と体重のあるもの」という定義になっている。
しかし、実際の猫は、鳴く、食べる、寝るといった行動を取る。
そういった、クラスの働きはメソッドと呼ばれる。
では、Catクラスに「鳴く(meow)」メソッドと「食べる(eat)」メソッドを追加してみよう。

```
class Cat{
    name: string
    length: number
    weight: number
    meow(): string {
        return 'にゃんー'
    }
    eat() {
        this.length += 0.1
        this.weight += 0.1
    }
}

const myCat = new Cat()
myCat.name = 'mimi'
myCat.length = 45 //cm
myCat.weight = 4 //kg
myCat.eat()
alert(myCat.name + 'の鳴き声は' + myCat.meow() + '\n体長は' + myCat.length + ' cm' + ' ,体重は' + myCat.weight
+ ' Kg')
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=class%20Cat%7B%0D%0A%20%20%20%20name%3A%20string%3B%0D%0A%20%20%20%20l
でコンパイル

public、private、protected

public (default)

プログラムを通して、宣言したメンバに自由にアクセスすることが出来ました。
TypeScriptでは、デフォルトで各メンバはpublicになります。
publicを明示的に指定することも可能です。

```
class Dog{
  public length:number
  public age:number

}
```

private

メンバにprivateを指定した場合、クラス外からのアクセスは不可になります。

```
class Animal {
    private name: string
    constructor(theName: string) { this.name = theName }
}
new Animal("Cat").name // Error: 'name' is private;
```

protected

protected修飾子は、指定されたメンバが継承先のクラスのインスタンスからでもアクセス可能であることを除いて、ほとんどprivate修飾子のように振る舞います。

```
class Person {
    protected name: string;
    constructor(name: string) { this.name = name; }
}
class Employee extends Person {
    private department: string;
    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }
    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}
let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // error
```

メソッドのオーバーロード

メソッドはオーバーロードできる。
つまり、同じ名前で異なる引数を持つ関数を宣言し、いずれの場合にも対応できるように実装すればよい。
ここでは、meowメソッドに引数を指定しなかった場合には「にゃーん」と鳴き、
引数を指定した場合には、その引数を鳴き声とする。

```
class Cat{
    name: string
    length: number
    weight: number
    meow(): string
    meow(s: string): string
    meow(s?: any): string
    {
        if (typeof (s) == 'string') {
            return s
        }
        else {
            return 'にゃんー'
        }
    }
    eat() {
        this.length += 0.1
        this.weight += 0.1
    }
}

const myCat = new Cat()
myCat.name = 'mimi'
myCat.length = 45
myCat.weight = 4
myCat.eat()
alert(myCat.name + 'の鳴き声は' + myCat.meow() + '\n体長は' + myCat.length + ' cm' + ' ,体重は' + myCat.weight
+ ' Kg')
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=class%20Cat%7B%0D%0A%20%20%20%20name%3A%20string%3B%0D%0A%20%20%20%20l
でコンパイル

コンストラクター

コンストラクターとは、インスタンスの作成時に自動的に実行されるメソッドで、
初期値の設定などに使われる。

```
class Cat {
  length: number
  weight: number
  name: string
  constructor() {
    this.name = '名なし'
  }
}

const myCat = new Cat()
alert('名前は' + myCat.name + 'です') // 名前は名なしです
window.close()
```

Playground
(http://www.typescriptlang.org/play/#src=class%20Cat%20%7B%0A%09length%3A%20number%3B%0A%09weight%3A%20number%3B%
でコンパイル

コンストラクターのオーバーロード

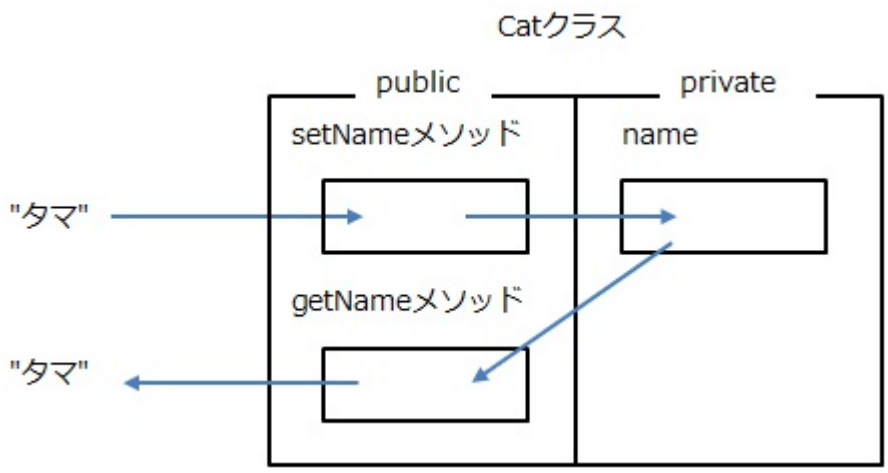
```
class Cat {
  length: number
  weight: number
  name: string
  constructor()
  constructor(s: string)
  constructor(s?: string) {
    if (typeof (s) == 'string') {
      this.name = s
    } else {
      this.name = '名なし'
    }
  }
}

const myCat = new Cat('タマ')
const yourCat = new Cat()
alert('私の猫の名前は' + myCat.name + '\nあなたの猫の名前は' + yourCat.name + 'です')
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=class%20Cat%20%7B%0A%09length%3A%20number%3B%0A%09weight%3A%20number%3B%
でコンパイル

情報の隠蔽

プライベート(private)なメンバー(ここではnameプロパティ)はクラス外からはアクセスできないが、
パブリック(public)なメンバー(ここではsetName／getNameメソッド)
はクラスの外からでもアクセスできる。
ここでは名前を8文字に制限することにしよう。
そのためには、nameプロパティをクラス外から自由にアクセスできないようにして、
長さのチェック機能を持ったメソッドを使ってのみアクセスできるようにすればよい。



重要な情報をプライベートな変数にして、クラスの外から勝手に変更されないようにすることを「情報の隠蔽」と呼ぶ。

```
class Cat {
    private name: string
    public setName(s: string) {
        this.name = s.slice(0, 8)
    }
    public getName(): string {
        return this.name
    }
}

const myCat = new Cat()
myCat.setName('アームストロングサイクロンジェットアームストロング砲')
alert('私の猫の名前は' + myCat.getName() + 'です')
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=class%20Cat%20%7B%0D%0A%20%20%20%20private%20name%3A%20string%3B%0D%0A%(
でコンパイル

setNameメソッドを使ってnameプロパティに値を設定できる。
ただし、長い文字列を指定しても、最初の8文字だけが使われることになる。
getNameメソッドを使って値を取り出せる。
変数の宣言やメソッドの定義の前に何も書かなかった場合には、publicが指定されたものと見なされる。

クラスの継承とメソッドのオーバーライド

継承とは、元のクラス（親クラス）の機能を全て受け継いだ新しいクラス（子クラス）を定義すること。
クラスを継承するには、新しいクラスの名前の後にextendsというキーワードを書き、
その後に親クラスの名前を書けばよい。
親クラスのメソッドを子クラスで定義し直すことを、オーバーライドと呼ぶ。

```
class Cat {
  private name: string
  public setName(s: string) {
    this.name = s.slice(0, 8)
    }
  public getName(): string {
  return this.name
  }
  public meow(): string {
    return 'にゃーん'
  }
}

class Tiger extends Cat {
  public meow(): string {
    return 'がおー'
  }
}

const myTiger = new Tiger()
myTiger.setName('とらお')
alert('私の虎の名前は' + myTiger.getName() + 'で、' + myTiger.meow() + 'と鳴きます')
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=class%20Cat%20%7B%0A%09private%20name%3A%20string%3B%0A%09public%20setName(s
でコンパイル

親クラスのメソッドを呼び出したいときには、メソッド名の前にsuperを付ければよい。

```
class Cat {
    private name: string
    public setName(s: string) {
        this.name = s.slice(0, 8)
    }
    public getName(): string {
        return this.name
    }
    public meow(): string {
        return 'にゃーん'
    }
}

class Tiger extends Cat {
    public meow(): string {
        return 'がおー'
    }
    public meowlikecat(): string {
        return super.meow()
    }
}

const myTiger = new Tiger()
myTiger.setName('とらお')
alert('私の虎の名前は' + myTiger.getName() + 'ですが、甘えているときには' + myTiger.meowlikecat() + '鳴きます')
window.close()
```

[Playground]
(http://www.typescriptlang.org/play/#src=class%20Cat%20%7B%0A%09private%20name%3A%20string%3B%0A%09public%20setName(s
でコンパイル

インターフェース

TypeScriptの核となる基本原則のひとつに、値の型チェックが値が持つ形状に焦点を当てていることがあげられます。
これは、時には"ダックタイピング"または"構造的部分型"と呼ばれます。
TypeScriptでは、インターフェースはこれらの型の名付けの規則を満たし、
また、プロジェクトの外観を構成するだけでなく、コードの構造を定義する強力な方法になります。

初めてのインターフェース

インターフェースがどのように動作するのかを、簡単な例で確認してみましょう。

```
function printLabel(labelledObj: { label: string }) {
    console.log(labelledObj.label);
}
let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

型チェッカーは、printLabel呼び出しをチェックします。printLabel関数はひとつの引数を持ち、
labelと呼ばれる文字列型のプロパティを持つオブジェクトが渡されることを必要とします。
実際のオブジェクトはこれより多くのプロパティを持ちますが、
コンパイラは必要とされている最低限のひとつのプロパティが存在し、
それが必要とされている型とマッチしていることしかチェックしないことに注意してください。
TypeScriptを寛大にさせないように、こちらで少しカバーすることになるケースが存在します。

```
interface LabelledValue {
    label: string;
}
function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}
let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

LabelledValueインターフェースは、先の例で必要とされていたものを表現するのに使用できる名前になります。
ここでも依然として、labelと呼ばれる文字列型の単一のプロパティを持つことが表されます。
printLabelに渡すこのオブジェクトが、他の言語でそうする必要があるように、
このインターフェースを実装すると明確に記述していない事に注意してください。
ここでは、事柄を形作っているだけに過ぎません。

関数に渡すオブジェクトが必須となるものを揃えてさえいれば、それは問題なく受け入れられます。
重要な点は、型チェッカーがこれらのプロパティに対して正しい順序であることを必要とせず、
インターフェースで指定されているプロパティが提供され、必須となる型を持っているかだけを必要としていることです。

任意のプロパティ

インターフェースの全てのプロパティを必須にする必要はありません。
幾つかのものはある条件下でのみ存在し、または全く存在しないようにすることも可能です。
これら任意のプロパティは、関数に必要なプロパティだけを持たせたオブジェクトを渡す際の、
"option bags"のようなパターンを作る際によく使用されます。
下記はこのパターンの例になります。

```
interface SquareConfig {
    color?: string;
    width?: number;
}
function createSquare(config: SquareConfig): {color: string; area: number} {
    let newSquare = {color: "white", area: 100};
    if (config.color) {
        newSquare.color = config.color;
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}
let mySquare = createSquare({color: "black"});
```

任意のプロパティを持つインターフェースの書き方は、他の言語のインターフェースの書き方に似ています。
宣言の中で任意にしたいプロパティの末尾に?の印を付けます。
任意のプロパティの利点は、利用できるプロパティを言い表し、
インターフェースの一部では無いプロパティが使用されることを防ぐ役割も果たしてくれます。
例えば、createSquareのcolorプロパティの名前を打ち間違えたとしても、
エラーメッセージがそのことをすぐに知らせてくれます。

```
interface SquareConfig {
    color?: string;
    width?: number;
}
function createSquare(config: SquareConfig): { color: string; area: number } {
    let newSquare = {color: "white", area: 100};
    if (config.color) {
        // Error: Property 'collor' does not exist on type 'SquareConfig'
        newSquare.color = config.collor;
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}
let mySquare = createSquare({color: "black"});
```

読み込み専用プロパティ

オブジェクトが作成される最初の時には、いくつかのプロパティは編集可(modifiable)のはずです。
プロパティ名の前にreadonlyを置くことで、読み込み専用の指定をすることが可能です。

```
interface Point {
    readonly x: number;
    readonly y: number;
}
```

Pointをオブジェクトリテラルを割り当てて作成することができますが、
その割当後にxとyを変更することはできません。

```
let p1: Point = { x: 10, y: 20 };
p1.x = 5; // error!
```

TypeScriptのReadonlyArrayの型は、変更処理を行う全てのメソッドが削除されたArrayと同義です。
そのため、配列の作成後に変更できないことを確認できます。

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;
ro[0] = 12; // error!
ro.push(5); // error!
ro.length = 100; // error!
a = ro; // error!
```

readonly vs const
readonlyまたはconstのどちらを使用するのかを覚える最も簡単な方法は、これは変数で使用するのかプロパティで使用するのかを確認することです。変数であればconstを使用し、プロパティであればreadonlyを使用します。

Function型

インターフェースには、JavaScriptオブジェクトがとり得る型の概要を記述することが可能です。
プロパティを持つオブジェクトを記述することに加え、インターフェースは関数の型を記述することも可能です。
インターフェースを使用して関数型を記述するために、インターフェースにコールシグネチャを与えます。
これはパラメーターの一覧と戻り値の型のみが指定された関数宣言のようなものです。
各パラメーターは名前と型の両方を必要とします。

```
interface SearchFunc {
    (source: string, subString: string): boolean;
}
```

定義すれば、他のインターフェースのようにこの関数型を使用することが可能になります。
ここで、関数型の変数をどのように作成し、同じ型の関数の値をどのように適用するのかをお見せします。

```
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    let result = source.search(subString);
    if (result == -1) {
        return false;
    }
    else {
        return true;
    }
}
```

関数型の型のチェックにおいて、パラメーターの名前は一致する必要はありません。
例えば、上記の例を下記のように書くことも可能です。

```
let mySearch: SearchFunc;
mySearch = function(src: string, sub: string): boolean {
    let result = src.search(sub);
    if (result == -1) {
        return false;
    }
    else {
        return true;
    }
}
```

Class型

C#とJavaのような言語での最も一般的なインターフェースの使われ方のひとつに、
クラスが明示的に特定の条件を満たすことを強制させるというものがあり、
それはTypeScriptでも可能です。

```
interface ClockInterface {
    currentTime: Date;
}
class Clock implements ClockInterface {
    currentTime: Date;
    constructor(h: number, m: number) { }
}
```

クラスに実装するインターフェース内のメソッドに対しての記述も可能であるため、
下記の例ではsetTimeに対してそれを行っています。

```
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}
class Clock implements ClockInterface {
    currentTime: Date;
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { }
}
```

インターフェースにはクラスのpublic、privateの両方ではなく、publicだけが記述されます。
これは、クラスのインスタンスのprivateに、クラスが特定の型を持たせているかチェックすることを禁止します。

インターフェースの拡張

クラスのように、インターフェースは拡張することが可能です。
これにより、あるインターフェースのメンバを別のものにコピーすることが可能になり、
インターフェースを再利用性のある部品へ柔軟に分離できるようにしてくれます。

```
interface Shape {
    color: string;
}
interface Square extends Shape {
    sideLength: number;
}
let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

インターフェースは複数のインターフェースを拡張することが可能で、全てのインターフェースを結合したものを作成します。

```
interface Shape {
    color: string;
}
interface PenStroke {
    penWidth: number;
}
interface Square extends Shape, PenStroke {
    sideLength: number;
}
let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
square.penWidth = 5.0;
```