

Swift Programming

Lecture 2

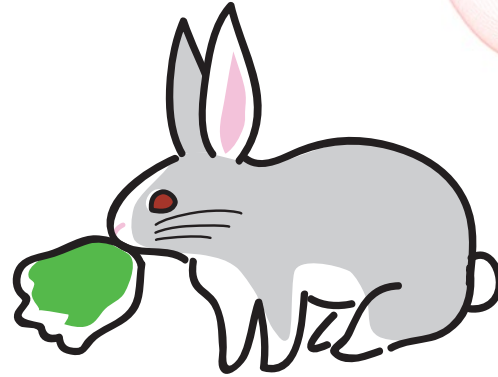
Literal and Expression

Tatsuo Minohara

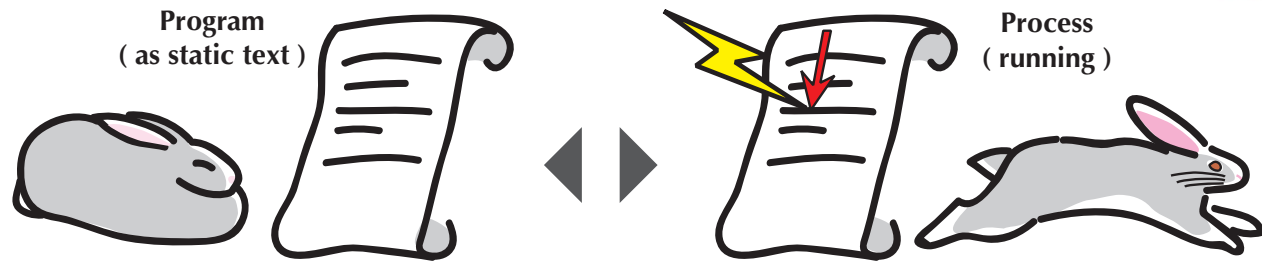
命令型プログラム

Imperative Program

Go to the center of the garden ;
Find a rabbit ;
Hold her ;
Bring her to your home ;
Give her a bit of lettuce ;
Bring back her to the garden ;



プロセスとプログラム





Swiftの文について

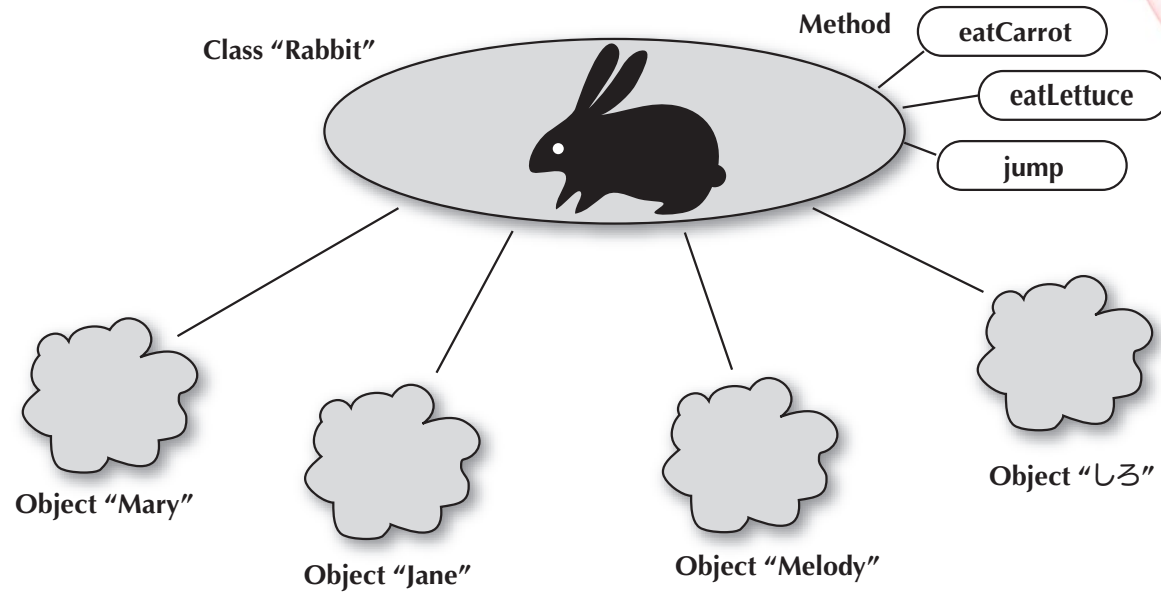
- 1 行に 1 つの命令を記述する
- 上から順番に実行
 - 例：
 - `print( 1 )`
 - `print( 2 )`
 - `print( 3 )`
- 1 行の中で 2 つ以上の命令を記述する場合は、命令を
; (セミコロン) で区切る。命令は、左から右に実行される
 - 例：`print( 1 ); print( 2 ); print( 3 )`



空白とコメント

- 左側に空白が空いているのがアウトラインのレベルを示す
➡TABキーを使う, Deleteで戻せる
- インデントがあってもなくても動作はする
- 読みやすくするために、インデントを下げる形を使う
- // この後改行するまでがコメント
- /* */ 囲まれた範囲がコメント

クラスとオブジェクト



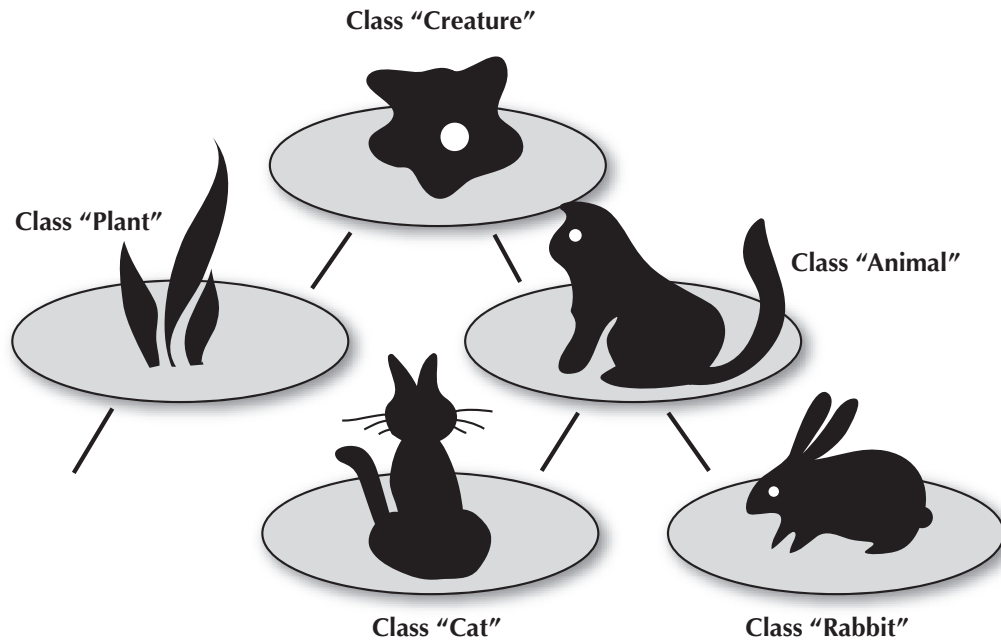


クラスを使うのは？

- 共通の特性を持つ対象について、その特性だけをプログラムで記述すれば、同じ特性を持つ対象をどんどん増やしていける。

クラス間の階層

Inheritance



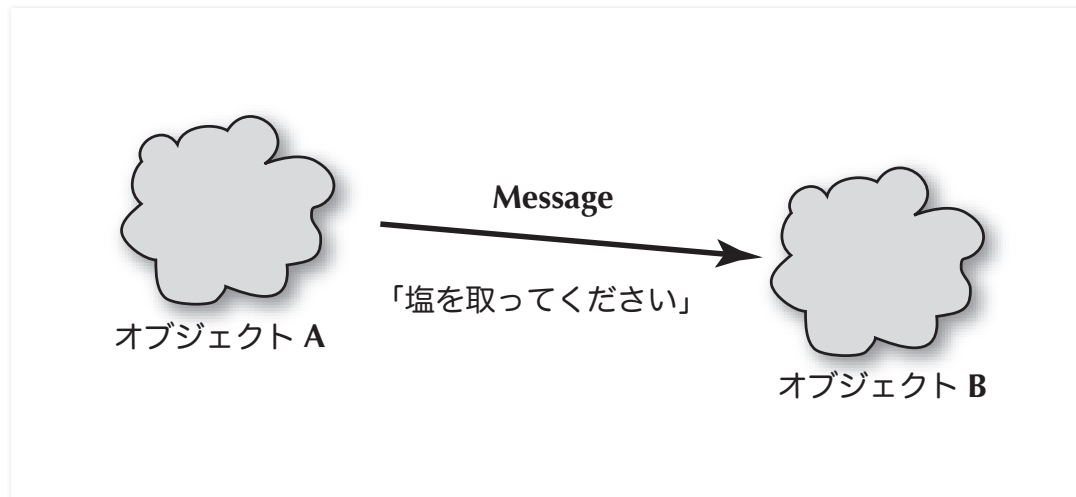


差分プログラミング

- 上位クラス（スーパークラス）で用意されている機能を使えば、その部分はプログラムする必要はない。
- 下位クラス（サブクラス）では、差分として必要な部分だけをプログラムすれば良い。
- オブジェクト指向の利点

オブジェクト指向の基本

- 対象を作る
- 相手（対象：Object）を指定して、何かを頼む



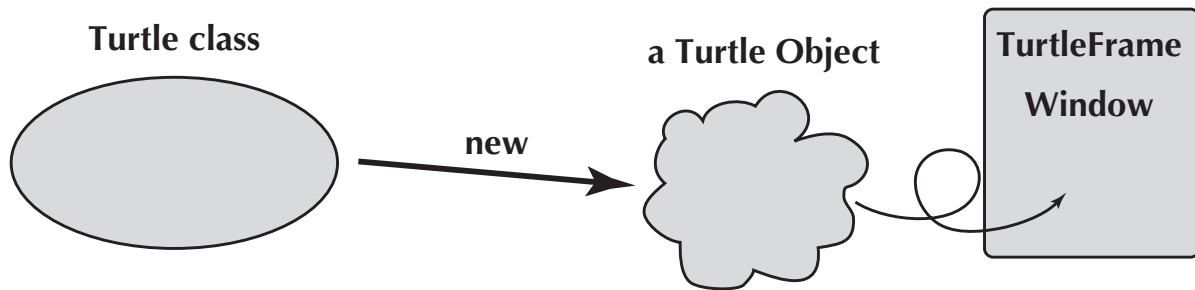
オブジェクトの生成

- 対象となるオブジェクトを作る書式：

クラス名(パラメータ)

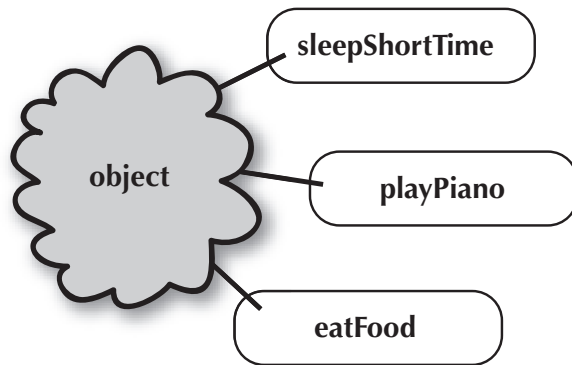
- Turtleクラスのオブジェクトを作る場合

➡ Turtle(window)



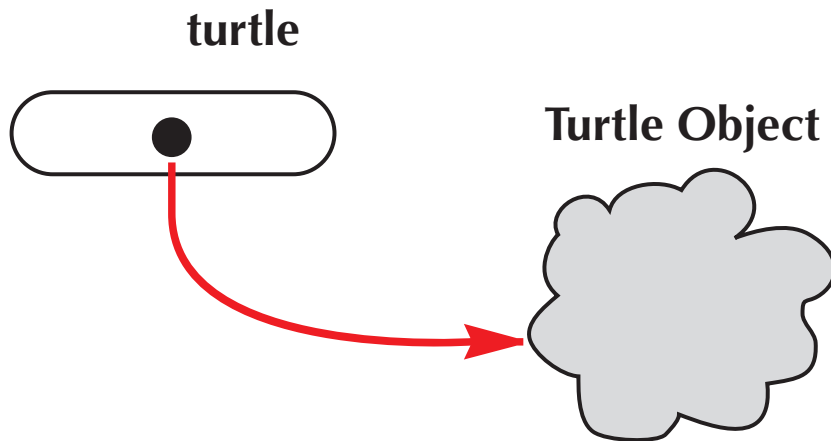
オブジェクト指向の基本

- 相手（対象）は頼まれたことを、処理する記述がなければならない。



オブジェクトを保持する変数に割付け

- `let turtle = Turtle( window )`

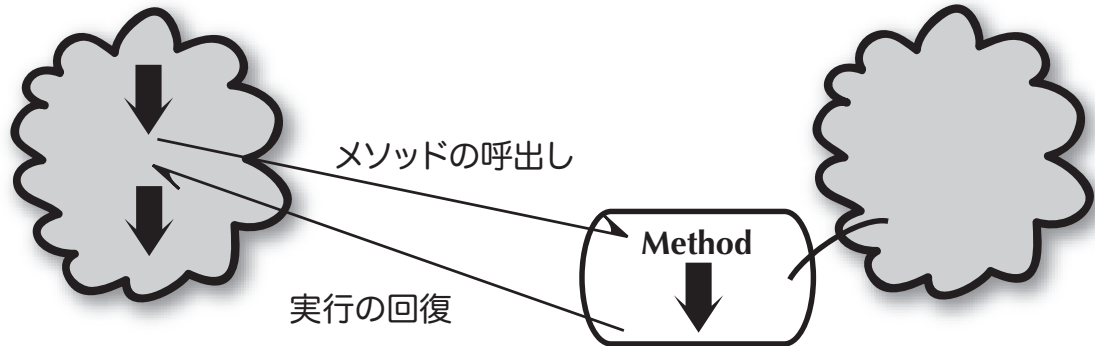


呼出し側と受取り側

- オブジェクト.メソッド名(パラメータ)
 - ▶ 例 : `window.drawString( "Hello", 20, 20 )`

メソッドの呼出し側

メソッドを持つオブジェクト





Swiftでの記述

- Swiftの文法では

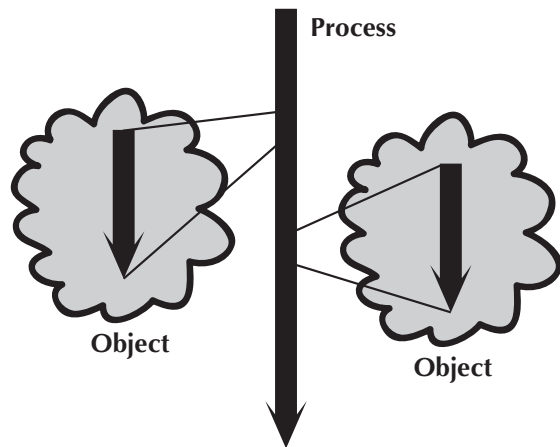
頼む相手.頼みたいこと(パラメータ)

パラメータは頼むときに渡したい情報

例： `window.setSize( 500, 200 )`

オブジェクト指向の動き

- オブジェクトのメソッドを呼び出しながら、進んで行く



Swiftの記号

記号	呼び方	意味
;	セミコロン	文の区切りを示す（1行に複数の文がある場合）
.	ドット	所属を示す「～の～」
,	カンマ	羅列を示す「～と～」
"	ダブルクォーテーション	囲んで文字列を示す
//	ダブルスラッシュ	行の終わりまでコメント
/* */	スラッシュアスタリスク	囲まれた範囲がコメント
*	アスタリスク	「ポインタ」または「掛け算」



プログラムは式を記述していく

- 式の構成要素
 - 変数
 - 定数（リテラル）
 - 演算子
- 定数は、値のこと
 - 一定の値を取り続けるということで、定数(constant)と呼ばれる
 - プログラミング言語では、定数が型を持つ



値の型

- Java/C/C++/C#/Swiftでは、値の型が厳格に参照される。
 - Strict type languages 厳格な型言語
- Python/JavaScript/Ruby/Swiftなどでは、結果的に値の型が決まっていく
- プログラム上で値を記述できる型 (primitive type)には次のようなものがある
 - 論理型
 - 整数型
 - 実数型
 - 文字列型
- プログラム上で記述された値 (型を持つ) をリテラル (literal) と呼ぶ



論理型

- 条件が満足されたかどうかを示す真偽値。
- **Bool**型（ブール代数から）と呼ばれる。
- 値としては、次の2つだけになっている。
 - **true**...条件を満足した（真）
 - **false**...条件を満足しない（偽）
- 値の型を求める組込みの関数として、`type( of: 値 )`がある
 - `type( of: true ) ⇒ Bool`



型に入らない特殊なリテラル

- `nil`...オブジェクトを指していない値
- 直接単独で記述することはできないが、条件式などに入れると記述することができる
 - 例： `nil == nil` → `true`になる

整数型と実数型

- 整数は、離散数 (Discrete Number) と呼ばれ、実数は連続数 (Linear Number) と呼ばれている。

Integer(離散数)



Real(連続数)



整数型

- 整数型の型名は、Int（標準）, UInt（符号無し）
- 符号付きと符号なしおよび、ビットサイズを指定した型も用意されている。
 - Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64
- Swiftでは、整数の桁数の制限がある

型	最大値	最小値
Int	2147483647 9223372036854775807	-2147483648 -9223372036854775808
Int8	127	-128
Int16	32767	-32768
Int32	2147483647	-2147483648
Int64	9223372036854775807	-9223372036854775808

整数型の記述例

- 符号無し整数型の最小値・最大値

型名	最大値	最小値
UInt	4294967295 18446744073709551615	0
UInt8	255	0
UInt16	65535	0
UInt32	4294967295	0
UInt64	18446744073709551615	0

- 整数の例：

- 7 2147483647 3
- 792281625142643
- 100_000_000_000



整数型の各型への変換

- 標準では整数リテラルを記述すると、その型はInt型になる
- UInt64やUIntなどは、そのままではリテラルとしては、式中に記述することはできない。そのため、オブジェクトの生成式を用いる
 - `UInt64( 1234 )` ⇒ UInt64型の1234を値を持つオブジェクトが生成される
 - `Int32( 345 )` ⇒ Int32型の345という値を持つオブジェクトが生成される
- また、型の変換演算子 **as** を使っても各型の値を持つオブジェクトを生成することができる
 - `1234 as UInt64`
 - `345 as Int32`



8進数、2進数と16進数の整数

- 8進数の整数

- ➡ 0oを先頭につける(0から7までしか使えない)

- ➡ 0o262730

- 16進数の整数

- ➡ 0xを先頭につける

- ➡ 0x45a

- 2進数の整数

- ➡ 0bを先頭につける

- ➡ 0b01010101



実数型

- 浮動小数点数で表現される
- Swiftでの型名としては、Double, Float
- ビット数を指定したものでは、Float32, Float64, Float80がある
- 計算精度は、Float80が一番高いが、計算速度はDoubleより落ちる
- 小数点をつけると自動的に実数型として認識される
 - 2.3 4. → 4.0 .05 → 0.05
- 指数表記が可能
 - 2.45×10^{16} → 2.45e16 あるいは 2.45e+16
- 実数の例：
 - 3.14 10. .001 1e100 3.14e-10 0e0 3.14_15_93



実数型の各型のリテラル

- 標準では、実数型のリテラルを記述すると、その型はDouble型（倍精度実数型：Float64と等価）となる
- Float（単精度実数型：Float32と等価）の実数型リテラルは、オブジェクトの生成式か、as演算子を使った型変換で生成する
 - `Float( 3.14 )` ⇒ 単精度の3.14の値を持つオブジェクトを生成
 - `3.14 as Float` (Float型の実数に変換する)
- Float80は、IntelのCPUが持つFPUだけで使えるので、M1 CPUのMacでは、使うことができない



文字列型

- 文字はすべてUnicodeで符号化されている。
- ファイルなどを読み込むときに、別の符号（JISやShift JIS）を用いるときは、符号（エンコーディング：encoding）の指定をしなければならない。
- 文字列はString型になっている
- Swiftでは、どちらかの引用符のスタイルを用いる
 - 値を記述するときは二重引用符で囲む "a"
 - 複数行の場合には、以下のように3つの二重引用符で囲む

```
"""
abcdefg
"""
```
- #で囲む
 - #"ABCDE"GGH"AAA"#



JIS/Shift JIS/EUC/Unicodeの違い

- ASCII...アメリカが作った英語用のコード（1バイト）0x00～0x7f
- JIS X0208...日本でつくられた2バイトの漢字・かななどが入った文字コード体系
- JIS X0201...日本でつくられた1バイトの半角カナを含む文字コード0x80以降も含む \...0x5c ¥...0x5c
- C:\Program Files\Excel.exe ← C:¥Program Files¥Excel.exe
- JIS X0208の漢字コードと半角カナを両方使いたい→Shift JISコード
- EUCは、1バイト単位でJIS + 0x80
- Unicodeは、Apple/Microsoftなどアメリカが勝手に考えたコード



Unicodeによる文字列

- Stringクラスという形で定義されている（ときどきNSStringも顔を出すけど同じ）
- `\n`は改行、`\t`は水平タブ、`\0`はnull文字、`\r`はMac固有の改行
- `\"`は二重引用符、`\'`は一重引用符
- `\\`はバックスラッシュ自身を表わす
- `\u{}`を用いて16進数4桁あるいは8桁で、文字を指定できる。
 - `\u{4e00}` → 一
 - `\u{00010C00}` → ヷ
- パレットでUnicodeのコード表を見てみましょう
 - 環境設定（キーボード）→キーボードタブ
 - メニューバーに絵文字とキーボードビューアを表示にチェック
 - 言語メニューから、絵文字と記号を表示→歯車メニューからリストをカスタマイズ
 - Unicodeにチェックを入れる



構造的な型のリテラル

- タプル

- 書式：(値, 値, ...) あるいは (キー名: 値, キー名: 値, ...)
- 異なる型の複数の値をまとめることができる
- 例： (34, "ABC", 78.9)

- リスト

- 書式：[値, 値, ...]
- 複数の値をまとめることができる
- リストでは同じ型の値しかまとめられない(4.0版まで)
- 例： ["34", "ABC", "78.9"]



構造的なリテラル（その他）

- 辞書

- 書式：[キー値：値, キー値：値, ...]
- 複数の値をまとめることができる
- すべてのキー値は同じ型の方が良い
- すべての値も同じ型でなければならない（4.0版まで）
- 同じキー値の要素は1つしか登録されない
- 例： ["Five": 56, "ABC": 12, "A": 79]