2022年度「プログラミング言語」配布資料 (1)

五十嵐 淳

2022年10月02日

1 Java

1.1 Java の主要概念の復習 (超特急版)

オブジェクト インスタンス変数の集まりからなるデータとそのデータに対する操作 (メソッド) の集まり.

インスタンス変数 オブジェクトに格納されるデータを表すための変数.

メソッド オブジェクト内のデータに対する操作．メソッドが呼出されると，呼出し元から与えられる引数とインスタンス変数を使って計算を行う．場合によってはインスタンス変数の値を代入によって更新する．最終的に何らかの計算結果を呼出し元に返す (ことが多い).

シグネチャ メソッドの名前，引数の個数とそれぞれの型，返り値の型をまとめて，メソッドのシグネチャと呼ぶ．シグネチャさえわかれば，そのメソッドをどういう形式で呼出せばよいかわかる.

クラス オブジェクトの定義を与えるプログラム構成単位．インスタンス変数の宣言，インスタンス変数の初期化処理を記述するコンストラクタ，メソッド定義が含まれる.

インターフェース メソッドシグネチャをいくつか集めたものに名前をつけたもの．典型的には，同一シグネチャのメソッドを持つ (異なる) クラスのオブジェクトをまとめて扱う際に便利である．具体的には 1. オブジェクトに共通するメソッドのシグネチャを列挙したようなインターフェース `I` を定義する．1. 各クラスは，そのインターフェースを実装するという宣言 (`implements I`) をつけて定義する．(そのクラスにはインターフェースに書かれたシグネチャを持つメソッドを定義する義務が生じる．)

こうすることによって，`I` 型の変数には，`I` を実装したクラスのオブジェクトをどれでも格納することができ，その変数を通じて `I` で宣言されているメソッドを呼ぶことができる．この際，実際に呼ばれるメソッド定義は，その時に変数に格納されているオブジェクトのクラスに依存する (**動的ディスパッチ (dynamic dispatch)**).

1.2 インターフェースについて

実験 1 では必ずしもインターフェースを使ったプログラムを書いているわけではないとのことなので，詳しめに書きます.

1.2.1 Java の静的型検査

Java は静的型付言語 (プログラムを実行する前に変数が演算子が予め決められた，もしくは宣言された型に従って使われているかを検査する言語) である．Java ではクラスの名前がそのままそのクラスのオブジェクトを表す型の名前になる．例えば，クラス Foo [1] を定義した場合，そのプログラムでは型 Foo の変数が宣言でき，その変数には Foo クラスのオブジェクトを格納することができる．

```
public class Foo {
  // インスタンス変数
  ...

  // コンストラクタ
  public Foo() {
     ...
  }

  // メソッド
  public int m(int y) {
    return y * 2;
  }
}
```

```
Foo x;            // Foo 型の変数の宣言
x = new Foo();    // x に Foo クラスから生成されたオブジェクトを格納する
int i = x.m(3);    // メソッド呼び出し
```

最後の行では x に格納されたオブジェクトに対しメソッド m の呼び出しを行っている．コンパイラは，メソッド呼び出しに対し以下のような検査 (と呼び出し結果の型の計算) を行う．1. ドットの前の式の型を調べる．ここでは，式は変数 x で，その型は宣言された通り Foo である．1. クラス Foo にメソッド m が定義されていることを確認する．m が定義されていなければエラーとなる．1. 定義側のパラメータの数とそれぞれのパラメータ型と，呼び出し側に書かれた実引数の数とそれらの型があっているかを検査する．(今回の m は int をひとつ取るメソッドで，実際，呼び出し側でも実引数として整数 3 がひとつ与えられているのでＯＫ．) 1. 呼び出し結果の型を，定義に書かれた返り値型 int とする．

このような検査 (**静的型検査**という) をコンパイル時にしておくことで，メソッド名の間違いや引数の間違いを，プログラム実行前に発見することができる．しかも，静的型検査に通った Java プログラムについては，**「未定義メソッドを呼び出す」「メソッドの引数の数が定義と呼び出し側で違う」といったエラーが実行時に発生しないことが理論的に保証されている．**

さて，Foo 型の変数に，別途定義したクラス Bar のオブジェクトを格納するプログラムを書くと，静的型検査でエラーになる．

```
public class Bar {
  // インスタンス変数
```

[1] “foo” ってどういう意味ですか？という人は https://ja.wikipedia.org/wiki/%E3%83%A1%E3%82%BF%E6%A7%8B%E6%96%87%E5%A4%89%E6 でも見てください．

```
  ...

  // コンストラクタ
  public Bar() {
     ...
  }

  // メソッド
  public int n() {
    return 0;
  }
}

Foo x = new Bar();  // コンパイルエラー!!
int i = x.m(3);
```

Bar にはメソッド m が定義されていないことから，実際，実行してはまずいプログラムである．静的型検査万歳!

一方，以下のクラス Baz を考えてみよう．

```
public class Baz {
  // インスタンス変数
  ...

  // コンストラクタ
  public Baz() {
     ...
  }

  // メソッド
  public int m(int y) {
    return y * 42;
  }
}

Foo x = new Baz();  // コンパイルエラー!?
int i = x.m(3);
```

Baz には int を引数とするメソッド m() が定義されているにも関わらず，実はこのプログラムも静的型検査ではじかれてしまう．静的型検査なんて嫌いだ!

実は，Bar の場合も Baz の場合もエラーがでているのは，x への代入文である．Java の代入文についての静的型検査の規則は

- 左辺の型 (Foo) が右辺の型 (Bar や Baz) と**代入互換 (assignment compatible)** でなければいけない

となっている．単に「代入互換」という言葉で置き換えただけで何も説明していないに等しいのだが，ひとまず，等しい型は代入互換，何の関係もなく定義されたクラスは代入互換ではない，ということだけ知っておけばひとまず十分である．特に Foo，Bar，Baz は代入互換ではないのである．ここで，「何の関係もなく定義」とは何か，と疑問に思った人はえらいので，もう少し待ってください．ともかく，このように代入文の規則を定めておくことで，メソッド m(int) を持たない (かもしれない) オブジェクトを x に格納することを防ぐことができるし，メソッド呼び出しの検査においても，x に格納されたオブジェクトに m があるかを調べる代わりに，型とクラスの文面をチェックすることでエラーを発見できることにつながっているのである．(ここで示したいずれの例でも x に何が格納されているかは自明だが，一般的にはとても難しい．)

1.2.2 共通のメソッドを持つクラスをまとめるインターフェース

インターフェースは，型が同じメソッドを持つ複数のクラスをまとめて扱うための機能である．まず，実例として，「int をひとつ引数として int を返すメソッド m を持つクラスたち」をまとめて扱うためのインターフェース Qux を定義する．

```
public interface Qux {
  int m(int z);
}
```

インターフェースは class の代わりに interface というキーワードを使って定義する．後にインターフェースに与えられた名前 Qux が続いている．波括弧 {} の中にあるのは，**シグネチャ (signature)** と呼ばれるもので，メソッドの名前，パラメータ引数の数と型，返り値の型からなる．書き方としては，メソッド定義から，メソッドの処理を記述した{ ... } を省いた (セミコロンで置き換えた) ようなものを並べる．(パラメータ引数の名前 z も書かなければいけないことになっているが，実は特に意味がない．)

クラス名がそのまま型になるのと同様，Qux も型の名前として変数宣言に使うことができる．

```
Qux q;
```

そして，Qux 型の変数に対しては，Qux に書かれたシグネチャに従ってメソッド呼び出しががができる．

```
int i = q.m(100);
```

この際の静的型検査の規則は，クラスの代わりにインターフェース定義を見にいってメソッドの有無，引数・返り値の型を確認すること以外は，先程と同じである．

この Qux は，直観的には「int をひとつ引数として int を返すメソッド m を持つクラスたち」を表しており，Foo と Baz はこの条件に該当するが，Bar は該当しない．Java では，クラスを定義する時に「このクラスはインターフェースこれこれに該当しますよ」ということを明示的に宣言して関連づけをする必要がある．この関連づけのためにはクラス定義の最初に implements〈インターフェース名〉という宣言 (これを「implements 節」という．“implement”＝「実装する」) を追加する．

```
public class Foo implements Qux {
  // 前と同じ
}
public class Baz implements Qux {
  // 前と同じ
}
```

このようにすることで, はじめて Foo や Baz オブジェクトを Qux 型の変数に格納することができる. (implements 節がなければ，m を持っていても代入できない．)

```
if (...) {
  q = new Foo();
} else
  q = new Baz();
}
```

先程登場した代入互換性の規則として説明すると,「クラス C が class C implements I とインターフェース I を実装している時，C は I と代入互換である」ということになる.

q に対するメソッド m の呼び出しで実際にどういう実行がされるかは，q に格納されたオブジェクトのクラスに依存する．例えば，上の if 文の後に,

```
int i = q.m(100);
```

が実行されたとする．if の条件式が true ならば，Foo オブジェクトが q に入っているので Foo に定義された m の定義が呼び出され，引数を 2 倍した 200 が返ってくるし，false ならば，Baz オブジェクトが入っているので，引数を 42 倍した 4200 が返ってくる．このように，メソッド呼び出しが発生した時点でのオブジェクト (のクラス) によって異なる処理が呼び出されることを**動的ディスパッチ (dynamic dispatch)** という．動的ディスパッチはオブジェクト指向言語の重要な機能のひとつであり，この講義でも応用例を見ていくことになる.

ちなみに, Bar のような m を持っていないクラスや, 持っていても引数の数や型が違うクラスに対し implements Qux を付けると，クラス定義に対して静的型検査のエラーが発生する．つまり，implements 節は，プログラマの主張するインターフェースとの関連付けが妥当なものかを検査することになる.

Java でのインターフェースを使ったプログラミングは，1. インターフェースの定義 1. クラス定義での implements 節の記述

という二段階をふまなければならず，わりと面倒である．Python など静的型検査がない言語では，変数に何でもかんでも代入できるので，このような面倒なステップをふまずに,「ある変数に Foo か Baz のオブジェクトが格納されている」状況を作りだすことができる．(もちろん動的ディスパッチがあるので，メソッド呼び出しで実際に呼ばれる処理はオブジェクトのクラスによって違うものになる．) 一方で,「何が格納されているか」についてのチェックが実行前には何もないので,「実は，ケアレスミスで Bar が格納されていたので m の呼び出しに失敗してプログラムが異常終了した」ということが発生する．Java では，インターフェースがあっても，上で述べたような各種チェックのおかげで，上で述べたように,**「未定義メソッドを呼び出す」「メソッドの引数の数が定義と呼び出し側で違う」といったエラーは実行時に発生しない.**

1.3 Java プログラムの基本要素

Java プログラムは，実際にはコンパイラによって Java 仮想機械の命令に変換され，それが Java 仮想機械によって実行されるわけだが，Java 仮想機械を持ち出すことなく，Java プログラムの実行過程を理解する・説明できるようになることは，非常に大事である.

同様に，何のプログラミング言語であっても，その実行過程を，実装方式や実行する計算機のハードウェアに依存しない抽象的なレベルで理解する・説明できるようになるのが望ましい．

1.3.1 変数，オブジェクト，参照

変数は，データを格納するための，名前がつけられた (メモリ) 領域である．変数の働きを理解するためには，

- 変数をどのように宣言するのか (宣言が不要な言語もある).
- 宣言された変数は，プログラム文面のどの範囲で使用できるのか．(これを変数の**有効範囲 (scope)** という．)
- 変数には何を格納するのか.
- 変数に対してはどのような操作が許されているのか.
- 変数に対応するメモリ領域はいつどのように確保されて，その内容が初期化されるのか

といったことを理解する必要がある．ひとつの言語に上の項目が異なる，何種類もの○○変数が登場することも多い．Java での代表的な変数はインスタンス変数，局所変数，クラス変数 (static 変数とも呼ばれる) である．

1.3.1.1 各種変数の共通事項

- 変数は全て，どんなデータを格納するか，その型を指定して宣言する．

```
int x = 100;

BinarySearchTree insert(BinarySearchTree t1, int i) {...}
```

- 変数が格納するのは整数 (`int`)，倍精度浮動小数点数 (`double`) などの数値，または，オブジェクトへの**参照 (reference)** (背番号，ID のようなもの) である．オブジェクトそのものは (メモリに領域が割り当てられるが) 変数には直接格納されない．パラメータや返り値を通じてやりとりされるのもあくまでオブジェクトへの参照である．
- 変数に対しては，そこに格納されている値 (数値・オブジェクトへの参照) の**読み出し** と，新しい値の **書き込み**のふたつの操作が考えられる．

1.3.1.2 インスタンス変数

- クラス本体内でメソッド・コンスタラクタと並んで宣言される．
- 有効範囲はそのクラス内．(この説明は，実際には不正確なところがあるのだが，ひとまず近似としては十分であろう．)
- `new` 式によってオブジェクトが生成されると，インスタンス変数のための領域が**ヒープ (heap)**[2]と呼ばれるメモリ領域中に確保され，初期化子やコンストラクタで，その内容の初期化が行われる．ヒープはプログラム実行中に新たに必要となる (別の言い方をすると，プログラム実行開始時には確保されていない) メモリ領域を提供する仕組みである．後述するように，多くのプログラミング言語ではヒープ以外にも実行時スタックと呼ばれる領域も使ってプログラムの実行が進む．
- そのオブジェクトが「使われなくなると」**ごみ集め (garbage collection)** と呼ばれるプログラム実行系 (Java なら Java 仮想機械) 内の仕組みによって，その領域は解放される．

[2]このヒープという用語は，ヒープソートの「ヒープ」とは関係ない．

1.3.1.3 局所変数 (メソッドのパラメータを含む)

- メソッドのヘッダ，for 文，文の並びの途中で宣言される．(private などのアクセス修飾子はつけてはいけない．)
- メソッドパラメータ，for 文の場合，その直後のメソッド本体/ブロック{ ... } が有効範囲，文の並びの途中の宣言は，その直後からブロック終了までが有効範囲となる．
- メソッド呼び出し，for 文の実行などをきっかけとしてメモリ領域が確保され，ブロックの終了とともにその領域は解放される．

1.3.1.4 クラス変数

- クラス本体内でメソッド・コンスタラクタと並んで宣言される．インスタンス変数と区別するため static 修飾子が必要．
- 有効範囲はそのクラス．
- プログラム実行開始とともに (これも本当はちょっと不正確) 領域が確保され，初期化子によって内容が初期化される．

1.3.1.5 補足 インスタンス変数は多くの場合 private という**修飾子 (modifier)** を伴って宣言するが，後述のように，この修飾子そのものは，変数がインスタンス変数であることを**意味しない**．インスタンス変数と局所変数を区別するのはその宣言の場所である．

1.3.2 オブジェクト生成

Java のオブジェクトは，概念的には，クラス名とインスタンス変数の値が集まったものである．

Java では new <クラス名>(<式 1>,...,<式 n>) という形の式[3]でオブジェクトを生成する．この式の評価は以下のような過程で行われる:

1. <式 1>, ⋯, <式 n> を評価する (それらの値を v_1, ⋯, v_n としよう).
2. 指定されたクラス名とそのクラスのインスタンス変数用の領域が確保され (未初期化の) オブジェクトが作られる．そして，オブジェクトに新しい (既にあるオブジェクトとは異なる)ID が割り当てられる．
3. コンストラクタのパラメータのための領域が確保され，v_1, ⋯, v_n で初期化される．この時，(代入不可能な) 特別な変数 this の領域も確保され，オブジェクトの ID が格納される．
4. コンストラクタ本体の実行．実行終了とともにパラメータと this のための領域は解放される．
5. オブジェクト ID が new 式全体の値となる．

1.3.3 メソッド呼び出し

メソッド呼び出し式は一般には

<式 0>.<メソッド名>(<式 1>,...,<式 n>)

という形をしている (は変数であることもしばしばであるが，一般には式が書け，場合によっては，メソッド呼び出し式が入れ子になることもある．例えば，x.m1().m2() と書けば，これは括弧を補えば (x.m1()).m2()

[3]正確には，式文という構文カテゴリーに属する．式文は式でありながら，セミコロンをつけることで文としても使用できるような式のことをいう．

のことで，x.m1() の呼び出しの結果として返される参照が指すオブジェクトに対して m2 を呼ぶ，という意味になる．

メソッド呼び出し式の評価は以下のような過程で行われる．1. <式 0>, ⋯, <式 n> を順に評価する (それらの値を v_0, ⋯, v_n としよう)．2. パラメータのための領域が新たに確保され，v_1, ⋯, v_n で初期化される．この時，特別な (代入不可能な) 変数 this も確保され，v_0 が格納される．3. v_0 は，あるオブジェクトへの参照である (はずな) ので，参照されているオブジェクトのクラスに定義されている<メソッド名> という名前のメソッドを選び，その本体を実行する．実行終了とともにパラメータと this のための領域は解放される．4. return された値が式全体の値になる．

冒頭でふれた動的ディスパッチの本質は呼び出すメソッドを選ぶステップにある．例えば，

```
public interface BinarySearchTree {
    void insert(int n);
}

public class Leaf implements {
    ...
}

public class Branch implements {
    ...
}
```

のような定義がされている時，BinarySearchTree 型の変数は Leaf や Branch クラスから作られたオブジェクトを格納している可能性がある．この場合，同じメソッド呼び出し式でも v_0 が参照するオブジェクトのクラスによって違うメソッドが呼び出される．例えば，

```
BinarySearchTree bst = new Leaf(...);
bst.insert(5);
```

であれば，Leaf 中の insert が呼ばれ (もし Leaf に insert の定義がなかったとしたらそもそもコンパイルできない)，

```
BinarySearchTree bst = new Branch(...);
bst.insert(5);
```

であれば，Branch 中の insert が呼ばれる．一般には，変数にどちらのクラスのオブジェクトが格納されているかはわからないので，実行しながら決めることになる．

1.3.4 変数とフレーム

メソッドやコンストラクタの実行の際に確保される，パラメータ変数のための領域を**フレーム (frame)** と呼ぶ．あるメソッドの実行中に使用できる変数は，このフレームに格納された変数のみである (インスタンス変数については後述)．フレームは，メソッドの呼び出し毎に必ず**新たに**確保されることに注意したい．あるメソッド実行中に，例え同じ名前のメソッドが実行されることになっても，二度目の実行のための新たなフレー

ムが用意されて，その下で実行が行われる．このため，二度目の実行中にパラメータ変数に代入を行ったとしても，最初の実行のフレームには影響が及ばない．

呼び出された側 (**callee**) のフレームの (確保から解放までの) 生存期間は，呼び出し元 (**caller**) のフレーム生存期間に完全に含まれる，いわゆる入れ子構造になっているため，フレームはスタックを使って実装することが多い．

1.3.4.1 局所変数

```
for(int i = 0; i < 10; i++) {
  ...
}
```

の `i` のような局所変数は，現在実行中のメソッドのフレームを一時的に (`{}`内の実行中だけ) 拡張することで確保していると考えられる．局所変数の生存期間は`{}` の入れ子構造と対応する．

Java では，ある局所変数の有効範囲内で同名の変数を宣言することはできない．(インスタンス変数の名前と同じなのは許されている．)

```
int foo(int x) {
  ...
  for(int x = 0; ...; ...) {...}
  ...
}
```

はコンパイル時にエラーになる．(同名の変数を二度宣言してもエラーにならず，使う際には「一番近くで」宣言された変数を参照する，という規則になっている言語も多い．Java がなぜエラーにしているかはわからない．紛らわしくてバグの温床になるからだろうか．)

1.3.4.2 インスタンス変数の正体 上で，メソッドの実行中に使用できる変数はフレーム内の変数のみだ，と書いた．フレームにはメソッドのパラメータなどの局所変数しか入っていない．では，インスタンス変数はどこへ行ってしまたんだ，と思うかもしれない．その秘密は，コンストラクタ・メソッド呼び出しでフレームに導入される特殊な変数 `this` にある．実は，インスタンス変数は全て `this` 経由で読み書きされているのだ!

例えば `Branch` クラスがインスタンス変数 `v` を定義しており，`insert` メソッド中に

```
if (n == v) { ... }
```

という文があるとしよう．これは内部的には

```
if (n == this.v) { ... }
```

の略記として扱われている．Java には<式>.<インスタンス変数名> という「<式>の評価結果であるところの参照が指すオブジェクトのインスタンス変数」という意味になる構文が用意されている．`this` はフレーム中に，今，操作しようとしている `Branch` オブジェクトを指す変数として存在するので，うまくインスタンス変数の値が取り出せる，というわけである．

この<式>.<インスタンス変数名> という表記は，ほとんど変数のようなもので，上のような値の読み出しに使えるだけでなく，代入文で書き込みを行うこともできる．後で見るように，`branch` を表すクラス `Branch` の

コンストラクタに

```
this.left = left;
```

という代入が現れている．右辺は (内側の宣言が優先されるので) 局所変数 (コンストラクタパラメータの)left の値 (オブジェクト参照)，左辺は，インスタンス変数の値を読み出しているわけではなく「this が指すオブジェクトの中の，インスタンス変数 left を格納している場所 」を示している．

有効範囲が重なる同名の局所変数宣言 (これは上に述べたようにエラーになる) と違って，インスタンス変数の有効範囲内で同名の局所変数を宣言するのは OK[4] となっているのは，このように this. 記法を使うことができるからかもしれない．

[4] 逆に，局所変数の有効範囲内でインスタンス変数が宣言されることは構文上ありえないことに注意.