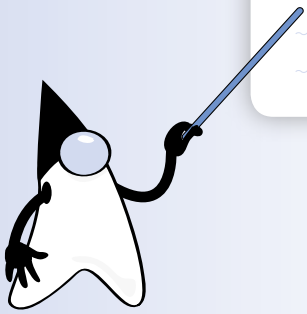
第12章

クラスの派生と多相性

本章では、既存クラスの資産を継承して新しいクラスを作るクラスの派生と、それを応用した多相性について学習します。

- □派生による資産の継承
- □上位／スーパークラス・下位／サブクラス
- □super(...) によるスーパークラスのコンストラクタ呼出し
- □super によるスーパークラスのメンバアクセス
- □is-A と多相性
- □参照型のキャスト（アップ／ダウン）
- □オーバライドと @Override アナテイション
- □final クラス
- □継承とアクセス性



12-1 継承

本節では、既存クラスの資産を継承しつつ、新しくクラスを作るための技術である、クラスの派生について学習します。

銀行口座クラス

第８章で《銀行口座》クラス *Account* を作成しました。ここでは、“定期預金”を表せるように変更することを考えます。以下に示すフィールドとメソッドを追加しましょう。

- 定期預金の残高を表すフィールド
- 定期預金の残高を調べるメソッド
- 定期預金を解約して全額を普通預金に移すメソッド

▶ 現実の銀行口座では、複数の定期預金をもつことができますが、ここでは簡単のために一つだけとします（定期預金の期日なども省略します）。

上記のフィールドとメソッドを追加した《定期預金付き銀行口座》クラス *TimeAccount* を **List 12-1** に示します。

▶ このクラスは、銀行口座クラスの第２版（p.277）を書きかえたものです。個々の口座に識別番号を与える第３版（p.338）をもとにしたものではありません。

List 12-1 Chap12/TimeAccount.java

```
// 定期預金付き銀行口座クラス【試作版】

class TimeAccount {
   private String name;           // 口座名義
   private String no;             // 口座番号
   private long balance;          // 預金残高
   private long timeBalance;      // 預金残高（定期預金）

   //--- コンストラクタ ---//
   TimeAccount(String n, String num, long z, long timeBalance) {
      name = n;                          // 口座名義
      no = num;                          // 口座番号
      balance = z;                       // 預金残高
      this.timeBalance = timeBalance;    // 預金残高（定期預金）
   }

   //--- 口座名義を調べる ---//
   String getName() {
      return name;
   }

   //--- 口座番号を調べる ---//
   String getNo() {
      return no;
   }
```

```
    //--- 預金残高を調べる ---//
    long getBalance() {
        return balance;
    }

    //--- 定期預金残高を調べる ---//
    long getTimeBalance() {
        return timeBalance;
    }

    //--- k円預ける ---//
    void deposit(long k) {
        balance += k;
    }

    //--- k円おろす ---//
    void withdraw(long k) {
        balance -= k;
    }

    //--- 定期預金を解約して全額を普通預金に移す ---//
    void cancel(long k) {
        balance += timeBalance;
        timeBalance = 0;
    }
}
```


このプログラムを完成させるのは、容易です。クラス *Account* のソースプログラムをコピーして、網かけ部を追加・修正するだけだからです。

*

クラス *TimeAccount* は、『定期預金を扱う』という目的を満たします。しかし、その一方で、**普通預金だけの銀行口座クラス *Account* との互換性は失われています。**

そのことを、以下に示すメソッドで考えましょう。これは、二つの口座 *a* と *b* の預金残高を比較して、その大小関係を整数値 1，-1，0 として返すメソッドです。

```
// どちらの預金残高が多いか
static int compBalance(Account a, Account b) {
    if (a.getBalance() > b.getBalance())          // aのほうが多い
        return 1;
    else if (a.getBalance() < b.getBalance())     // bのほうが多い
        return -1;
    return 0;                                     // aとbは同じ
}
```

このメソッドに対して、*TimeAccount* 型インスタンス（への参照）を渡すことは不可能です。というのも、クラス *Account* とクラス *TimeAccount* は、別ものだからです。

*

あるクラスをコピーして、部分的な追加・修正を施すプログラミングを続けていくと、互換性の無い“似て非なる”クラスが溢れかえってしまうため、プログラムの開発効率・拡張性・保守性が低下することになります。

重要 ソースプログラムの安易な切り貼りによって、新しいクラスを作るべきではない。

派生と継承

このような問題を解決するのが、クラスの**派生**（*derive*）です。派生とは、既存クラスのフィールドやメソッドなどの《資産》を**継承**（*inheritance*）した新しいクラスを作り出すことです。なお、派生の際は、資産を**継承**するだけではなく、フィールドやメソッドを**追加**したり、**上書き**したりすることもできます。

Base というクラスがあって、その資産を継承したクラス *Derived* を派生する例を示したのが、**Fig.12-1** です。派生によって新しく作るクラスの宣言では、**extends** に続いて派生元のクラス名を書きます。

extend は『拡張する』という意味の動詞ですから、クラス *Derived* の宣言は、

> クラス *Derived* は *Base* を拡張したクラスですよ！

と読めますね。

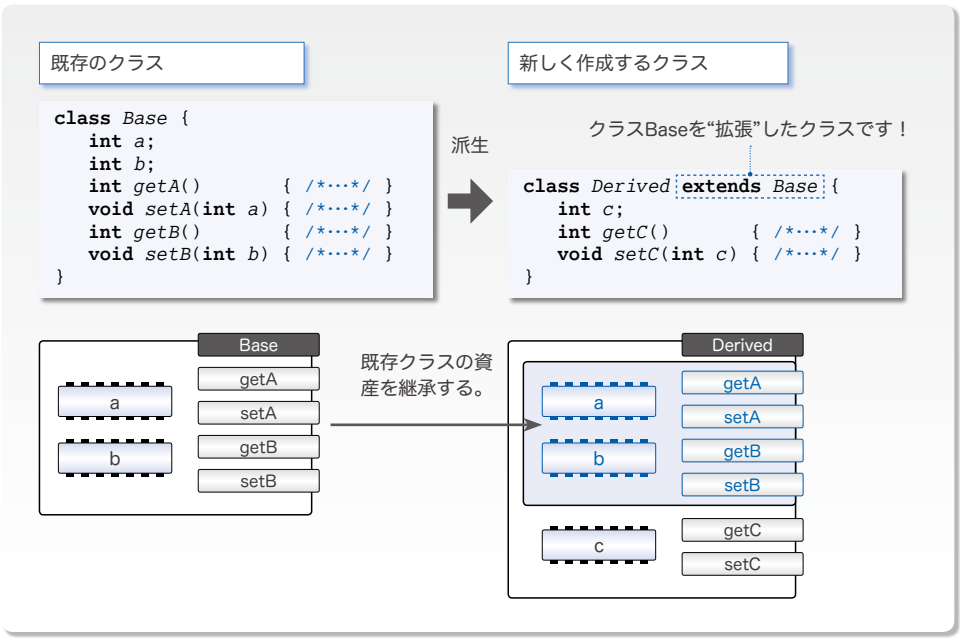


Fig.12-1　クラスの派生

なお、派生の元になるクラスと、派生によって作られたクラスのことを、以下のように表現します（いろいろな呼び方があります）。

> 派生元のクラス … 親クラス ／ 上位クラス ／ 基底クラス ／ スーパークラス
> 派生したクラス … 子クラス ／ 下位クラス ／ 派生クラス ／ サブクラス

本書では、**上位／スーパー**と**下位／サブ**の用語を使うことにします。

すなわち、『クラス *Base* からクラス *Derived* が派生している』ことを、以下のように表現します。

・クラス *Derived* にとって、クラス *Base* はスーパークラス（上位クラス）である。
・クラス *Base* にとって、クラス *Derived* はサブクラス（下位クラス）である。

▶ 上位・スーパー／下位・サブといった用語については、p.398 で再考します。

各クラスの資産は、次のようになっています。

■ クラス *Base*

フィールドは *a*，*b* の **2 個**で、メソッドは *a*，*b* のセッタとゲッタの **4 個**です。

■ クラス *Derived*

このクラスで宣言しているのはフィールド *c* と、そのセッタとゲッタです。とはいえ、クラス *Base* のフィールドとメソッドを**そっくり継承している**のですから、それらを合わせると、フィールドは **3 個**、メソッドは **6 個**となります。

サブクラスは、スーパークラスのフィールドやメソッドなどの資産を継承しますので、スーパークラスのメンバは、サブクラス中に含まれます。そのため、スーパークラスとサブクラスの資産は、**Fig.12-2** a に示す関係となります。

重要 サブクラス（下位クラス）は、スーパークラス（上位クラス）の資産を継承するとともに、それを部分として含むクラスである。

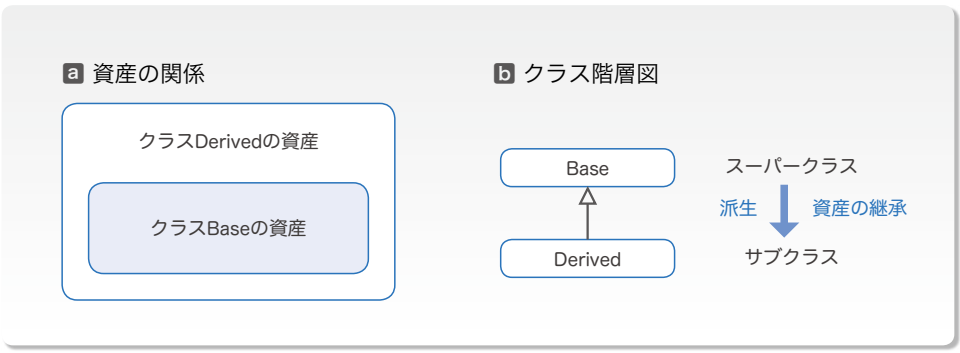


Fig.12-2 派生による資産の継承とクラス階層図

サブクラスは、スーパークラスから生み出された“子供”にたとえられます。この親子関係は、図 b に示すクラス階層図として表現されます。クラス階層図では、サブクラスからスーパークラスに向かって矢印を結びます。資産の継承とは**逆向き**です。

派生とコンストラクタ

派生において継承されない資産があります。その一つがコンストラクタです。そのため、サブクラスでは、コンストラクタを新しく作るのが原則となります。

重要 クラスの派生において、コンストラクタは継承されない。

▶ クラスが回路の設計図であるとすると、コンストラクタは電源スイッチ用のチップに相当するのでしたね。スーパークラスの電源スイッチが、それを拡張したサブクラスでもそのまま使える、ということは一般的にはありえませんから、コンストラクタは継承されないのです。

コンストラクタについては、クラスの派生と関連して、いくつかの点を必ず押さえておく必要があります。まずは、派生を行う具体的なプログラム例である **List 12-2** を例に考えていきましょう。

List 12-2 Chap12/PointTester.java

```
// ２次元座標クラスと３次元座標クラス

// ２次元座標クラス
class Point2D {
    int x;    // Ｘ座標
    int y;    // Ｙ座標

    Point2D(int x, int y) { this.x = x; this.y = y; }

    void setX(int x) { this.x = x; }     // Ｘ座標を設定
    void setY(int y) { this.y = y; }     // Ｙ座標を設定

    int getX() { return x; }             // Ｘ座標を取得
    int getY() { return y; }             // Ｙ座標を取得
}

// ３次元座標クラス
class Point3D extends Point2D {
    int z;    // Ｚ座標

    Point3D(int x, int y, int z) { super(x, y); this.z = z; }

    void setZ(int z) { this.z = z; }     // Ｚ座標を設定
    int getZ() { return z; }             // Ｚ座標を取得
}

public class PointTester {

    public static void main(String[] args) {

        Point2D a = new Point2D(10, 15);
        Point3D b = new Point3D(20, 30, 40);

        System.out.printf("a = (%d, %d)\n",      a.getX(), a.getY());
        System.out.printf("b = (%d, %d, %d)\n", b.getX(), b.getY(), b.getZ());
    }
}
```

継承される

スーパークラスのコンストラクタの呼出し

実行結果

```
a = (10, 15)
b = (20, 30, 40)
```

① スーパークラスのコンストラクタは super(...) によって呼び出せる

このプログラムでは、２次元座標クラス *Point2D*、３次元座標クラス *Point3D*、それらをテストするクラスが定義されています。

▶ フィールド名やメソッド名などは異なるものの、これらのクラスは、**Fig.12-1**（p.390）に示したクラス *Base* および *Derived* と、ほぼ同じ構造です。

・クラス *Point2D* … ２次元座標クラス（X座標／Y座標）

座標を表す２個のフィールド *x*，*y* と、４個のセッタ・ゲッタと、コンストラクタとから構成されるクラスです。

コンストラクタは、仮引数 *x*，*y* に受け取ったX座標とY座標の値を、フィールド *x* と *y* に設定します。

・クラス *Point3D* … ３次元座標クラス（X座標／Y座標／Z座標）

２次元座標クラス *Point2D* クラスから派生したクラスです。

X座標とY座標に関するフィールドとメソッドは、そのまま継承します。新しく追加されたのは、Z座標のフィールド *z* と、そのセッタ・ゲッタです。

*

コンストラクタ内の網かけ部である **super(***x*，*y***);** に着目しましょう。**super(...)** という式は、**スーパークラスのコンストラクタの呼出し**です。

super(*x*，*y***)** の呼出しを行うのは、仮引数 *x*，*y* に受け取った値をフィールド *x* と *y* に代入する作業を、スーパークラスのコンストラクタに委ねるためです。その結果、クラス *Point3D* のコンストラクタ内で直接値を代入するのは、新たに追加されたZ座標用フィールド *z* だけですんでいます。

なお、**super(...)** の呼出しは、**コンストラクタの先頭でのみ行えます**。

> **重要** コンストラクタの先頭で **super(...)** を実行することによって、スーパークラスのコンストラクタを呼び出せる。

▶ スーパークラスのコンストラクタを呼び出す **super(...)** は、同一クラス内の他のコンストラクタを呼び出す **this(...)** と似ていますね。なお、一つのコンストラクタの中で、**super** と **this** の両方を呼び出すことはできません。

*

二つの座標クラスを利用する **main** メソッドに注目しましょう。変数 *a* は２次元座標クラス、変数 *b* は３次元座標クラスです。

３次元座標クラスでは定義されていないメソッド *getX* とメソッド *getY* を、変数 *b* に対して呼び出しています。このようなことが行えるのは、スーパークラス *Point2D* の資産であるメソッド *getX* と *getY* をクラス *Point3D* が継承しているからです。

② サブクラスのコンストラクタ内では、スーパークラスに所属する“引数を受け取らないコンストラクタ”が自動的に呼び出される。

クラス *Point3D* のコンストラクタから **super(...)** の呼出しを削除して、以下のように書きかえてみましょう。そうすると、コンパイルエラーが発生します。

```
// コンパイルエラー
Point3D(int x, int y, int z) { this.x = x; this.y = y; this.z = z; }
```

このような **super(...)** を明示的に呼び出さないコンストラクタには、スーパークラスに所属する“引数を受け取らないコンストラクタ”の呼出し、すなわち **super()** の呼出しがコンパイラによって自動的に挿入されます。

すなわち、コンストラクタは、以下のように書きかえられているのです。

```
// コンパイルエラー
Point3D(int x, int y, int z) { super(); this.x = x; this.y = y; this.z = z; }
                               ↑ コンパイラが挿入
```

コンパイルエラーとなる理由は単純です。スーパークラス *Point2D* に“引数を受け取らないコンストラクタ”が存在せず、それを呼び出せないからです。

重要 明示的に **super(...)** を呼び出さないコンストラクタの先頭行には、スーパークラスに所属する“引数を受け取らないコンストラクタ”を呼び出す **super();** が挿入される。

③ コンストラクタを一つも定義しなければ、super() を呼び出すだけのデフォルトコンストラクタが自動的に定義される。

コンストラクタを 1 個も定義しないクラス *X* に対しては、何も行わない空(から)のコンストラクタである《デフォルトコンストラクタ》が、コンパイラによって以下の形式で自動的に定義されることを、第 8 章で学習しました。

```
X() { }
```

そこでの解説は、厳密にいうと説明不足のものでした。自動生成されるデフォルトコンストラクタは、本当は、以下のように定義されるのです。

```
X() { super(); }  // コンパイラによって生成されるデフォルトコンストラクタ
```

すなわち、デフォルトコンストラクタは、スーパークラスに所属する“引数を受け取らないコンストラクタ”を呼び出すコンストラクタとして定義されるのです。

重要 コンストラクタを 1 個も定義しないクラスには、以下に示す形式のデフォルトコンストラクタが自動的に定義される。

```
X() { super(); }
```

このことを検証する具体的なプログラム例を **List 12-3** に示します。

List 12-3 Chap12/DefaultConstructor.java

```
// スーパークラスとサブクラス（デフォルトコンストラクタの働きを確認）

// スーパークラス
class A {
   private int a;

   A() { a = 50; }

   int getA() { return a; }
}

// サブクラス
class B extends A {
   // コンストラクタを定義していない（デフォルトコンストラクタが生成される）
}

public class DefaultConstructor {

   public static void main(String[] args) {
      B x = new B();

      System.out.println("x.getA() = " + x.getA());
   }
}
```

実行結果

```
x.getA() = 50
```

クラス *A* がもつ唯一のフィールドが、**int** 型の *a* です。コンストラクタは、そのフィールド *a* に 50 を代入します。メソッド *getA* は、その値のゲッタです。

クラス *B* は、クラス *A* のサブクラスです。コンストラクタが 1 個も定義されていないため、以下のデフォルトコンストラクタがコンパイラによって定義されます。

```
B() { super(); }      // デフォルトコンストラクタ
```

クラス *B* のインスタンス生成時は、このコンストラクタが起動します。その際、**super()** によってクラス *A* のコンストラクタが呼び出されて、フィールド *a* に 50 が代入されます。実行結果からも、そのことが確認できます。

▶ スーパークラスのコンストラクタが継承されないとはいえ、“引数を受け取らない”コンストラクタが間接的な形で継承されることが分かりますね。

*

なお、クラス *A* のコンストラクタを以下の定義に置きかえると、クラス *B* がコンパイルエラーとなります。クラス *B* のコンストラクタで **super()** を呼び出せなくなるからです。

```
A(int x) { a = x; }  // クラスBがコンパイルエラーとなる
```

このことから、以下に示す注意点が得られます。

重要 クラスにコンストラクタを定義しない場合、そのスーパークラスが、“引数を受け取らないコンストラクタ”をもっていなければならない。

メソッドの上書きと super の正体

スーパークラスのコンストラクタを呼び出すための **super** の正体は、そのクラスに部分として含まれるスーパークラス部への参照です。そのため、次の規則があります。

> **重要** “**super**. メンバ名” によって、スーパークラスのメンバをアクセスできる。

このことを、**List 12-4** に示すプログラムで確認しましょう。

List 12-4 Chap12/SuperTester.java

```
// スーパークラスとサブクラス

// スーパークラス
class Base {
   protected int x;  // 限定公開（このクラスと下位クラスからアクセスできる）

   Base()      { this.x = 0; }
   Base(int x) { this.x = x; }

   void print() { System.out.println("Base.x = " + x); }
}

// サブクラス
class Derived extends Base {
   int x;      // スーパークラスと同一名のフィールド

   Derived(int x1, int x2) { super.x = x1; this.x = x2; }

   // スーパークラスのメソッドを上書き（オーバーライド）
   void print() { super.print(); System.out.println("Derived.x = " + x); }
}

public class SuperTester {

   public static void main(String[] args) {
      Base a = new Base(10);
      System.out.println("-- a --");  a.print();

      Derived b = new Derived(20, 30);
      System.out.println("-- b --");  b.print();
   }
}
```

Baseのx
Derivedのx
Baseのprint

実行結果

```
-- a --
Base.x = 10
-- b --
Base.x = 20
Derived.x = 30
```

■ クラス Base

このクラスで宣言されたフィールド *x* は、**protected** 付きで宣言されています。このように宣言されたメンバは、たとえパッケージの外であってもサブクラスからアクセスできる《**限定公開**》アクセスとなります（p.383）。

フィールド *x* の値を表示するのが、メソッド *print* です。

▶ クラス *Base* と *Derived* は、同じ “無名パッケージ” に所属しています。もし二つのクラスの所属するパッケージが異なっていても、クラス *Derived* からクラス *Base* の *x* にアクセスできることになります。なお、フィールドは **private** としておき、そのフィールドをアクセスするセッタやゲッタなどのメソッドを **protected** にするのが、本来の手法です。

■ クラス Derived

このクラスでは、フィールド *x* を宣言しています。クラス *Base* のフィールド *x* と同名ですが、**Fig.12-3** に示すように、クラス *Base* から継承したものとは別ものとして扱われます。

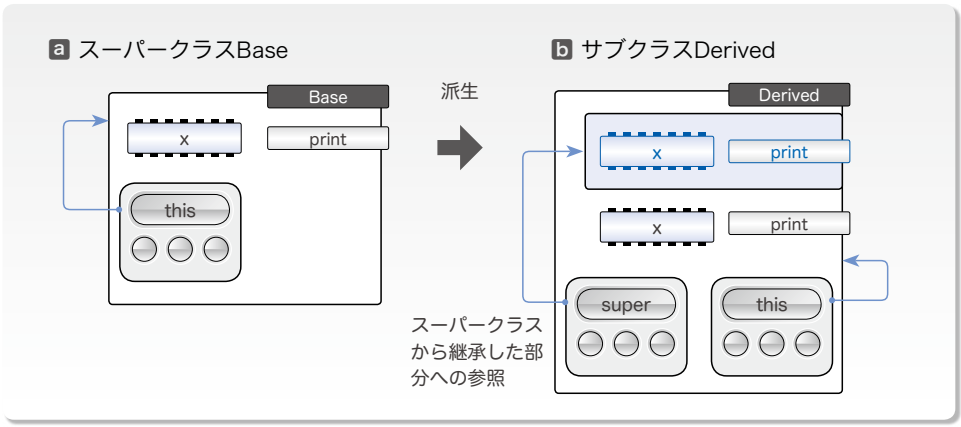


Fig.12-3 クラスの派生とsuper

コンストラクタに注目しましょう。仮引数 *x1* と *x2* に受け取った値を、二つの *x* に代入しています。**super.***x* は、スーパークラス *Base* から継承したフィールド *x* のことであり、**this.***x* は、自分自身のクラス *Derived* で宣言されたフィールド *x* のことです。

▶ なお、**super.** も **this.** も付けずに、ただ *x* とすれば、**this.***x* を指すことになります。同名のフィールドやメソッドがある場合、スーパークラスのほうの名前が**隠される**のです。

メソッド *print* は、引数を受け取らず値を返却しないという点で、クラス *Base* のものと同じ形式です。このメソッド中の **super.***print*() は、スーパークラス *Base* に所属するメソッド *print* の呼出しです。

そのため、クラス *Derived* のメソッド *print* を呼び出すと、まずクラス *Base* の *x* が表示され、それからクラス *Derived* の *x* が表示されることになります。

▶ スーパークラスのメソッドと同一形式のメソッドをサブクラスで定義し直すことを『**オーバライドする**（*override*）』と表現します。オーバライドについては、次節で詳しく学習します。

Column 12-1 スーパークラスとサブクラス

ここでは、スーパークラスとサブクラスというネーミングについて考えます。

sub とは『部分』という意味で、super とは『部分を含んだ全体・完全』という意味です。

“資産”の量という観点では、スーパークラスはサブクラスの《部分》ですから、sub や super のニュアンスとは反対です。少々紛らわしいため、間違えないようにしましょう。

ちなみに、プログラミング言語 C++ では、サブクラス／スーパークラスと呼ばずに、派生クラス／基底クラスと呼んでいます。

クラス階層

ここまでの例は、あるクラスから別のクラスを派生する例でした。派生したクラスからさらに派生を行うこともできます。そのような例を示したのが **Fig.12-4** です。

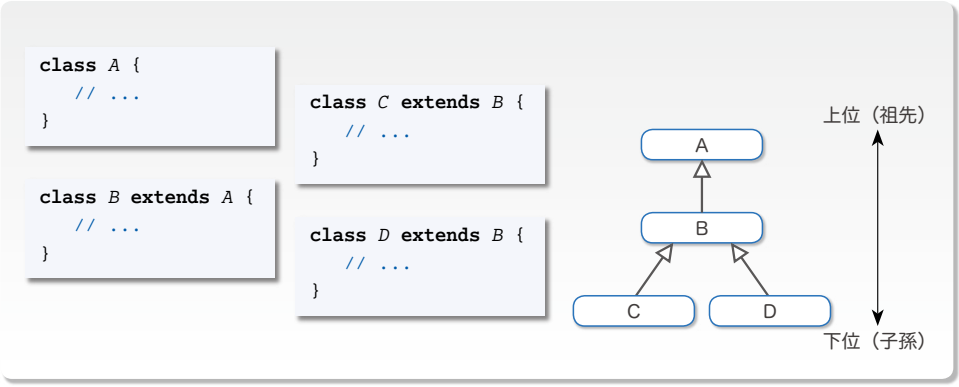


Fig.12-4　クラスの派生

クラス A からクラス B を派生し、クラス B からクラス C とクラス D を派生しています。クラス B は A の**子**です。そして、クラス C と D は、B の**子**であると同時に、A の**孫**です。すべてのクラスに《**血縁関係**》があるわけです。

親を含めた上側のクラスを**祖先**、子を含めた下側のクラスを**子孫**と呼ぶことにすると、上位クラス・下位クラスは、以下のように定義されます。

- **上位クラス**（*super class*）… 祖先クラス（親・お爺さん・曾お爺さん… ）のこと。
- **下位クラス**（*sub class*）… 子孫クラス（子・孫・曾孫… ）のこと。
- **直接上位クラス**（*direct super class*）… 親クラスのこと。
- **直接下位クラス**（*direct sub class*）… 子クラスのこと。

しかし、この表現は、少し分かりにくいだけでなく紛らわしいこともあるため、これ以降は、**Table 12-1** のように表現することにします。

Table 12-1　スーパー／上位クラス と サブ／下位クラス（本書の定義）

名称	定義
スーパークラス	派生の元になったクラス（親）のこと。
サブクラス	派生によって作られたクラス（子）のこと。
上位クラス	親を含めた祖先クラス（親・お爺さん・曾お爺さん…）のこと。
下位クラス	子を含めた子孫クラス（子・孫・曾孫）のこと。
間接上位クラス	親を除いた祖先クラス（お爺さん・曾お爺さん…）のこと。
間接下位クラス	子を除いた子孫クラス（孫・曾孫）のこと。

▶ 紛らわしい理由の一つが、本来の文法用語としての“スーパー”が**親を含めた祖先**を意味するのに対して、キーワードとしての“**super**”が**親のみ**を指すことです。キーワード **super** は、親クラス（直接上位クラス）への参照で、**super()** は親クラスのコンストラクタの呼出しです。親より上の世代のスーパークラスへの参照や、それらのクラスのコンストラクタの呼出しではありません。

Java では、複数のクラスから派生を行う多重継承がサポートされていません。そのため、**Fig.12-5** に示すクラスはコンパイルエラーとなります。

多重継承では、異なるスーパークラスに同一名のフィールドやメソッドなどが含まれる際の内部処理が非常に複雑となります。その多重継承をあえてサポートしないのは、Java が簡潔さを目指していることの一つの現れです。

Fig.12-5　Javaは多重継承をサポートしない

▶ C++ では多重継承がサポートされています。多重継承のサポートは、言語の仕様を大きくし、コンパイラに負担をかけますが、忌み嫌うべきものではありません。C++ の標準ライブラリでは、入力ストリームと出力ストリームから多重継承によって入出力ストリームを作り出すという、素晴らしいお手本が示されています。

Column 12-2　クラス階層図における矢印の向き

クラス階層図における矢印が《サブクラス → スーパークラス》すなわち《子 → 親》となっているのには理由があります。以下の宣言を例に考えましょう。

```
class Derived extends Base { /*…*/ }
```

この宣言の網かけ部である“**extends** *Base*”は、『私は *Base* を親にします。』という宣言です。このことは、親であるクラス *Base* の知らないところで勝手に子供が作られてしまうことを意味しています。

子供（サブクラス）は親（スーパークラス）を知っているのですが、親（スーパークラス）は子供（サブクラス）のことを知りません。そもそも子供がいるのか、いるのであれば何人いるのか、といった情報を、親はもつことができないのです。

人間の親が、自分の作った子供を知っているのとは逆ですね。

スーパークラス側で「このクラスを私の子供にしますよ。」といった宣言はできません。サブクラス側で行うのは「このクラスを私の親にしますよ。」という宣言ですから、矢印の向きは《サブクラス → スーパークラス》となるのです。

Object クラス

Fig.12-6 を見てください。これは、**Fig.12-4**（p.398）に示したクラスの宣言と、クラス階層図をより厳密に書き直したものです。この図には、**Object** というクラスが登場します。クラス *A* のように、**extends** を付けずに宣言されたクラスは、**Object** クラスのサブクラスとなります。

▶ **Object** クラスは、**java.lang** パッケージに所属します。このクラスについては、**Column 12-6**（p.420）で解説します。

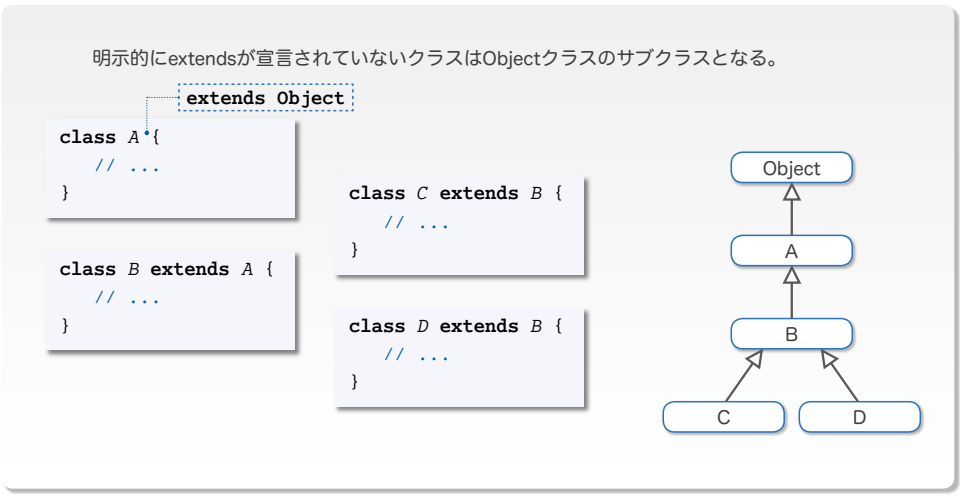


Fig.12-6　クラスの派生

したがって、前章までに作成してきた **extends** を伴わないクラス（*Account* や *Car* や *Day* などのクラス）は、**Object** クラスのサブクラスだったのです。

そして、**Object** 以外のすべてのクラスは、**Ojbect** クラスの下位クラスとなります。図の例であれば、クラス *A* は **Object** の子で、クラス *B* は孫です。

すべてのクラスは、**Object** という共通の祖先をもった《**親戚クラス**》というわけです。

▶ そのため、何の関係もないように見えるクラス *Account* や *Car* や *Day* は、**Object** という共通の親をもった“兄弟”ということになります。

重要　明示的な派生の宣言をしないクラスは Object クラスのサブクラスとなる。Java のすべてのクラスは、Object クラスの下位クラスである。

なお、**Object** クラスは、上品な言葉ではありませんが、《**親玉クラス**》と呼ばれることもあります。

差分プログラミング

本章の最初の話題に戻りましょう。銀行口座クラスに対して、定期預金を追加する例を検討していましたね。別のクラスとして作り直すのではなく、派生によってクラスを作ることにしましょう。それが **List 12-5** に示すクラス *TimeAccount* です。

List 12-5 account2/TimeAccount.java

```
// 定期預金付き銀行口座クラス【第１版】

class TimeAccount extends Account {
   private long timeBalance;            // 預金残高（定期預金）

   // コンストラクタ
   TimeAccount(String name, String no, long balance, long timeBalance) {
      super(name, no, balance);         // クラスAccountのコンストラクタの呼出し
      this.timeBalance = timeBalance;   // 預金残高（定期預金）
   }

   // 定期預金残高を調べる
   long getTimeBalance() {
      return timeBalance;
   }

   // 定期預金を解約して全額を普通預金に移す
   void cancel(long k) {
      deposit(timeBalance);
      timeBalance = 0;
   }
}
```

▶ 本プログラムのコンパイル・実行には、同一ディレクトリ上に銀行口座クラス第２版（p.277）を必要とします。

本クラスでは、フィールド・コンストラクタ・二つのメソッドのみを宣言します。それ以外のフィールドとメソッドは、スーパークラス *Account* から継承していますから、新たに定義し直す必要はありません。

*

継承のメリットの一つは、“既存プログラムに対する必要最低限の追加・修正だけで新しいプログラムが完成する”という**差分プログラミング**（*incremental programming*）が行えることです。プログラム開発時の効率アップや保守性の向上が図れます。

重要 互換性のない“似て非なる”クラスの作成を検討するより先に、“継承”による解決の可能性を吟味せよ。

▶ ２次元座標クラスから３次元座標クラスを派生する **List 12-2**（p.392）も、差分プログラミングの例でした。もっとも、継承が効果を発揮するのは、《差分プログラミング》ではなく、次節で学習する《多相性》です。

is-A の関係とインスタンスへの参照

TimeAccount は、*Account* の子供であって、*Account* 家（*Account* 一族）に属していると考えられます。このことは **is-A の関係**と呼ばれ、以下のように表現します。

> ***TimeAccount* は一種の *Account* である。**

この関係の逆は成立しないことに注意しましょう。*Account* は一種の *TimeAccount* ではありません。なお、is-A は、kind-of-A とも呼ばれます。

＊

is-A の関係が、どのようにプログラム上で活用できるのかを、二つのプログラムを通じて学習します。まずは、**List 12-6** のプログラムです。

▶ 黒網部は、コメントアウトしていなければコンパイルエラーとなる箇所です。

List 12-6 account2/TimeAccountTester1.java

```
// is-Aの関係とインスタンスへの参照（その１）

class TimeAccountTester1 {

   public static void main(String[] args) {
      Account adachi = new Account("足立幸一", "123456", 1000);
      TimeAccount nakata = new TimeAccount("仲田真二", "654321", 200, 500);

      Account x;      // クラス型変数は …
      x = adachi;        // 自分自身の型のインスタンスを参照できる（当たり前）  1
      x = nakata;        // 下位クラス型のインスタンスも参照できる！  2

      System.out.println("xの預金残高：" + x.getBalance());

      TimeAccount y; // クラス型変数は …
//    y = adachi;        // 上位クラス型のインスタンスは参照できない！  3
      y = nakata;        // 自分自身の型のインスタンスを参照できる（当たり前）  4

      System.out.println("yの預金残高：" + y.getBalance());
      System.out.println("yの定期預金残高：" + y.getTimeBalance());
   }
}
```

実行結果

```
xの預金残高：200
yの預金残高：200
yの定期預金残高：500
```

main メソッドの冒頭で、以下に示す二つのインスタンスを生成しています。

> *adachi* … 銀行口座クラス *Account* 型のインスタンス
> *nakata* … 定期預金付き銀行口座クラス *TimeAccount* 型のインスタンス

その後で宣言されている変数 *x* は *Account* 型のクラス型変数で、変数 *y* は *TimeAccount* 型のクラス型変数です。

1〜4では、*x* と *y* に対して、*adachi* や *nakata* のインスタンスへの参照を代入しています。これらについて、**Fig.12-7** を見ながら理解していきましょう。

1・4　*Account* 型の変数 *x* が同一型の *adachi* インスタンスを参照して、*TimeAccount* 型の変数 *y* が同一型の *nakata* インスタンスを参照しています。何も問題はありません。

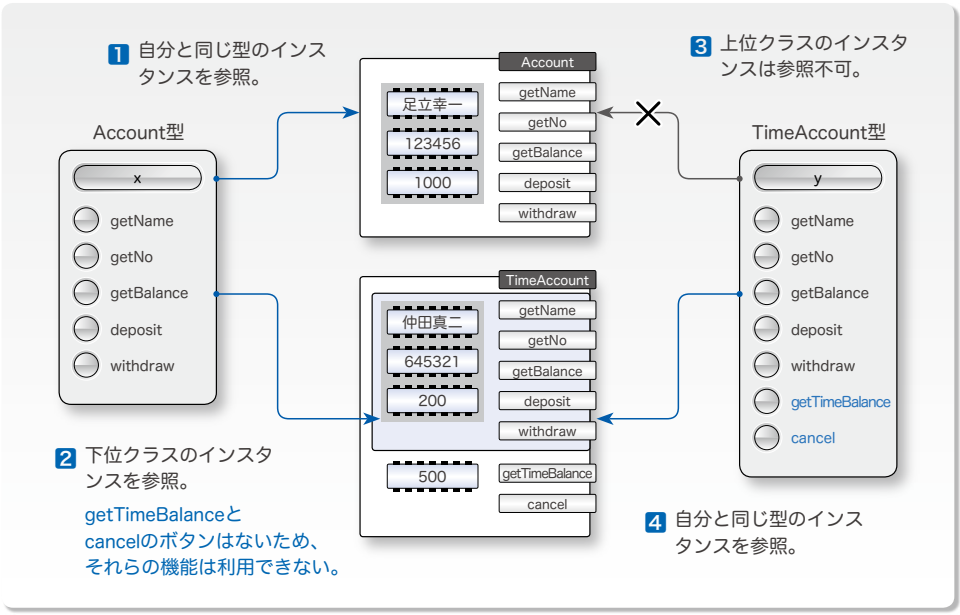


Fig.12-7 スーパークラス／派生クラスのインスタンスへの参照

2 *Account* 型の変数 *x* が、（一種の *Account* である）下位クラス *TimeAccount* 型のインスタンス *nakata* を参照しています。

図に示すように、サブクラスであるクラス *TimeAccount* の中に含まれているクラス *Account* の部分を、*x* が参照していると理解しましょう。*TimeAccount* 型のインスタンスを、*Account* 型のリモコンで操作できることになります。

▶ *Account* 型リモコンには、*TimeAccount* 特有の *getTimeBalance* と *cancel* ボタンがありませんので、それらの機能は使えなくなります。すなわち、*TimeAccount* 型インスタンスを、*Account* 型として操作できる状態になっているのです。

3 これは、ちょうど**2**と逆の関係です。*TimeAccount* 型の変数 *y* は、スーパークラスである *Account* 型の *adachi* インスタンスを指すことは**できません**。

仮に変数 *y* が *adachi* を指すことができたらどうなるかを考えてみましょう。

リモコン *y* の**定期預金残高を調べるボタン** *getTimeBalance* を実行すること、すなわち *y.getTimeBalance()* の呼出しが可能になってしまいます。しかし、そのようなことは許されません。というのも、リモコン *y* の参照先である *adachi* は、定期預金をもたないクラス *Account* 型のインスタンスだからです。

重要 スーパークラス型の変数はサブクラスのインスタンスを参照できるが、サブクラス型の変数はスーパークラスのインスタンスを参照できない。

▶ 明示的なキャスト演算子を適用すれば、参照は可能です。p.412 で学習します。

次に考えるのは、**List 12-7** に示すプログラムです。

プログラム網かけ部のメソッド *compBalance* は、本章の冒頭（p.389）で検討したものです。このメソッドは、二つの口座 *a* と *b* の普通預金の預金残高を比較して、その結果に応じて、1，-1，0 のいずれかの値を返却します。

List 12-7 account2/TimeAccountTester2.java

```
// is-Aの関係とインスタンスへの参照（メソッドの引数で検証）
class TimeAccountTester2 {
   // どちらの預金残高が多いか          TimeAccount試作版（p.388）ではエラー／第１版では動作
   static int compBalance(Account a, Account b) {
      if (a.getBalance() > b.getBalance())          // aのほうが多い
         return 1;
      else if (a.getBalance() < b.getBalance())     // bのほうが多い
         return -1;
      return 0;                                     // aとbは同じ
   }

   public static void main(String[] args) {
      Account adachi = new Account("足立幸一", "123456", 1000);
      TimeAccount nakata = new TimeAccount("仲田真二", "654321",  200, 500);

      switch (compBalance(adachi, nakata)) {
       case  0 : System.out.println("足立君と中田君の預金残高は同じ。");  break;
       case  1 : System.out.println("足立君のほうが預金残高が多い。");  break;
       case -1 : System.out.println("仲田君のほうが預金残高が多い。");  break;
      }
   }
}
```

実行結果

```
足立君のほうが預金残高が多い。
```

仮引数 *a* と *b* の型は *Account* です。**main** メソッドでは、それらに対して、*Account* 型インスタンスへの参照と、*TimeAccount* 型インスタンスへの参照を渡しています。

これがうまくいくのは、*Account* 型の変数が、*Account* 型インスタンスと、*TimeAccount* 型インスタンスのいずれをも参照できるためですね。

▶ もし、仮引数 *a* と *b* の型が *TimeAccount* となっていたら、**main** メソッドから *TimeAccount* インスタンスへの参照は渡せますが、*Account* インスタンスへの参照を渡せなくなってしまいます。

実行結果からも、預金残高の比較が正しく行われていることが分かります。

重要 メソッドのクラス型引数に対しては、そのクラス型のインスタンスへの参照だけでなく、そのクラスの下位クラス型のインスタンスへの参照を渡すことができる。

□ **演習 12-1**

総走行距離を表すフィールドと、その値を調べるメソッドを追加した自動車クラスを作成せよ。自動車クラス *Car* 第２版（p.324）から派生すること。

Column 12-3　Object 型の引数と配列

メソッドの仮引数がクラス型であれば、そのクラスの下位クラス型のインスタンスへの参照も受け取れることが分かりました。

Object クラスは、すべてのクラスの最上位クラスですから、**仮引数の型が Object であれば、あらゆるクラス型のインスタンスへの参照を受け取れる**ことになります。実際にプログラムで確認してみましょう。それが、**List 12C-1** のプログラムです。

List 12C-1　Chap12/ToString.java

```
// toStringが返却する文字列を表示するメソッド（すべてのクラス型に対応）
class X {                                          ← Objectの子
   public String toString() {
      return "Class X";
   }
}

class Y extends X {                                ← Objectの孫
   public String toString() {
      return "Class Y";
   }
}

public class ToString {
   //--- toStringメソッドが返却する文字列を表示 ---//
   static void print(Object obj) {
      System.out.println(obj);
   }

   public static void main(String[] args) {
      X x = new X();
      Y y = new Y();
      int[] c = new int[5];                        ← 配列もObjectの子
      print(x);
      print(y);
      print(c);
   }
}
```

実行結果

```
Class X
Class Y
[I@ca0b6
```

クラス *X* は、暗黙の内に **Object** クラスから派生し、クラス *Y* は、クラス *X* から派生しています。

メソッド *print* が受け取る仮引数 *obj* の型は **Object** 型です。そのため、あらゆるクラス型のインスタンスへの参照を、*obj* に受け取ることができます。このメソッドが行うのは、*obj* に対して toString メソッドを起動することによって得られる文字列を表示することです。

main メソッドに着目しましょう。*X* 型のインスタンス *x* と、*Y* 型のインスタンス *y* と、**int** 型の配列 *c* を生成し、それらのオブジェクトへの参照をメソッド *print* に渡しています。

変数 *x* に対しては、クラス *X* の toString メソッドが返却する文字列 "Class X" が表示され、変数 *y* に対しては、クラス *Y* の toString メソッドが返却する文字列 "Class Y" が表示されます。

＊

ここで着目すべきことは、**Object 型引数に対して、配列への参照を渡せること**です。実は、配列は、プログラムの内部で、実質的にクラスと同等なものとして扱われます。すなわち、配列は **Object** クラスの下位クラス（一種の **Object**）なのです。配列クラスは、toString メソッドや、要素数を表す **final int** 型のフィールド length などをもっています。

配列を出力すると特殊な文字列が表示されることを **List 6-15**（p.206）で確認していました。実は、配列クラスの toString メソッドが呼び出されていたのです。

12-2 多相性

本節では、《クラスの派生》の真の価値を引き出す多相性について学習します。

メソッドのオーバライド

List 12-8 のプログラムを見てください。このプログラムには、二つのクラスが定義されています。クラス *Pet* と、それから派生したクラス *RobotPet* です。

まずは、これらのクラスを理解しましょう。

■ クラス Pet（ペット）

- **フィールド**

  name … ペットの名前です。

  masterName … 飼い主の名前です。

- **コンストラクタ**

  Pet … ペットと飼い主の名前を設定します。

- **メソッド**

  getName … ペットの名前を調べるメソッド（*name* のゲッタ）です。

  getMasterName … 飼い主の名前を調べるメソッド（*masterName* のゲッタ）です。

  introduce … 自己紹介をするメソッドです（**Fig.12-8** a）。

a クラスPet

```
■僕の名前は++++です！
■ご主人様は****です！
```

b クラスRobotPet

```
◇私はロボット。名前は++++。
◇ご主人様は****。
```

Fig.12-8 メソッドintroduce

■ クラス RobotPet（ロボット型ペット）

- **フィールド**

  クラス *Pet* のフィールドを継承しています。

- **コンストラクタ**

  RobotPet … ペットと飼い主の名前を設定します。設定作業は、**super(...)** を呼び出すことによって、スーパークラス *Pet* のコンストラクタに任せます。

- **メソッド**

  introduce … 自己紹介をするメソッドです（図b）。クラス *Pet* のものを継承せずに、上書きしています。

  work … 家事をするメソッドです。家事の種類（掃除／洗濯／炊事）は、引数で 0，1，2 の値として指定します。

クラス *RobotPet* とクラス *Pet* のメソッドの関係をまとめると、以下のようになります。

- そのまま継承 … *getName*，*getMasterName*
- 上書き … *introduce*
- 新規追加 … *work*

List 12-8 pet/Pet.java

```
// ペットクラス

class Pet {
    private String name;          // ペットの名前
    private String masterName;    // 飼い主の名前

    // コンストラクタ
    public Pet(String name, String masterName) {
        this.name = name;                 // ペットの名前
        this.masterName = masterName;     // 飼い主の名前
    }

    // ペットの名前を調べる
    public String getName() { return name; }

    // 飼い主の名前を調べる
    public String getMasterName() { return masterName; }

    // 自己紹介
    public void introduce() {
        System.out.println("■僕の名前は" + name + "です！");
        System.out.println("■ご主人様は" + masterName + "です！");
    }
}

class RobotPet extends Pet {
    // コンストラクタ
    public RobotPet(String name, String masterName) {
        super(name, masterName);      // スーパークラスのコンストラクタ
    }

    // 自己紹介
    public void introduce() {
        System.out.println("◇私はロボット。名前は" + getName() + "。");
        System.out.println("◇ご主人様は" + getMasterName() + "。");
    }

    // 家事をする
    public void work(int sw) {
        switch (sw) {
         case 0: System.out.println("掃除します。"); break;
         case 1: System.out.println("洗濯します。"); break;
         case 2: System.out.println("炊事します。"); break;
        }
    }
}
```

サブクラスに継承される

オーバライド（上書き）する

新規追加 … RobotPet専用メソッド

メソッド *introduce* のように、スーパークラスのメソッドと同形式のメソッドに、サブクラスで別の定義を与えることを、『**オーバライドする**（*override*）』と表現します。

この場合は、次のようになっているわけです。

> クラス *RobotPet* のメソッド *introduce* は、クラス *Pet* のメソッド *introduce* をオーバライドする。

▶ override には、『決定済みのことをくつがえす』といったニュアンスがあります。スーパークラスのメソッドを無効にして、新しいメソッドを上書きするのです。

多相性

二つのクラスをテストするプログラムを **List 12-9** に示します。

List 12-9 pet/PetTester1.java

```
// ペットクラスの利用例（多相性の検証）
class PetTester1 {

    public static void main(String[] args) {
1       Pet kurt = new Pet("Kurt", "アイ");
        kurt.introduce();
        System.out.println();

2       RobotPet r2d2 = new RobotPet("R2D2", "ルーク");
        r2d2.introduce();
        System.out.println();

3       Pet p = r2d2;
        p.introduce();
    }
}
```

実行結果

```
■僕の名前はKurtです！
■ご主人様はアイです！

◇私はロボット。名前はR2D2。
◇ご主人様はルーク。

◇私はロボット。名前はR2D2。
◇ご主人様はルーク。
```


1 クラス *Pet* のインスタンス *kurt* を生成して、自己紹介を行わせます。呼び出されるのは、クラス *Pet* に所属するメソッド *introduce* です。

2 クラス *RobotPet* のインスタンス *r2d2* を生成して、自己紹介を行わせます。呼び出されるのは、クラス *RobotPet* に所属するメソッド *introduce* です。

3 *Pet* 型の変数 *p* が、*RobotPet* 型のインスタンスを参照するように初期化しています。変数と参照先の型が異なる点で、**1**や**2**とは異なります。

▶ is-A の関係によって、“ペット”を参照する変数は、ペットを参照できるのはもちろん、一種のペットである“ロボット型ペット”も参照できるのでしたね。

ここで、メソッド呼出し *p.introduce()* に着目しましょう。このメソッド呼出しは、以下に示すⒶ方式とⒷ方式のどちらに解釈されるのでしょうか？

Ⓐ ペット Pet 用の自己紹介メソッドが呼び出される。

変数 *p* の型が *Pet* であるため、*Pet* 型の自己紹介用メソッド *introduce* が呼び出される。すなわち、右のように表示される。

```
■僕の名前はR2D2です！
■ご主人様はルークです！
```

Ⓑ ロボット型ペット RobotPet 用の自己紹介メソッドが呼び出される。

参照先のインスタンスが *RobotPet* 型であるため、*RobotPet* 型の自己紹介用メソッド *introduce* が呼び出される。すなわち、右のように表示される。

```
◇私はロボット。名前はR2D2。
◇ご主人様はルーク。
```

プログラムが以下のようになっているとして、コンパイラである javac の立場に立って両方式を検討しましょう。

```
if (sw == 1)
   p = kurt;       // pはPet型インスタンスを参照する
else
   p = r2d2;       // pはRobotPet型インスタンスを参照する

p.introduce();     // pの参照先は、PetとRobotPetのどちら？
```

ここで、*sw* は **int** 型の変数であるとします。当然、その値は、プログラムを実行するたびに変わる可能性があります。

そのため、網かけ部が実行される際の *p* の参照先が、*Pet* 型の *kurt* なのか、それとも *RobotPet* 型の *r2d2* なのかは、プログラムのコンパイル時ではなく、実行時に決定されることを見落としてはいけません。

■ Ⓐ方式

変数 *p* が、*kurt* を参照しているか、*r2d2* を参照しているかにかかわらず、*Pet* 型のメソッド *introduce* を呼び出すコードを生成することになります。

そのため、コンパイル作業は容易であり、生成するコードは単純になります。

■ Ⓑ方式

生成するコードは、次のようになります。

> *sw* が 1 のとき　… クラス *Pet* 型のメソッド *introduce* を呼び出す。
> そうでないとき … クラス *RobotPet* 型のメソッド *introduce* を呼び出す。

すなわち、プログラムの実行時に呼び出すメソッドを切りかえるようなコードを生成しなければなりません。

そのため、コンパイル作業は面倒であり、生成するコードは複雑になります。

▶ ソースプログラムの表面上は単なる『メソッド呼出し』ですが、クラスファイル中のコードは、呼び出すメソッドを条件によって切りかえるという複雑なものになります。そのため、Ⓐ方式に比べると、ほんの少しだけとはいえ、プログラムの実行も遅くなります。

*

以上の考察から、以下のことが分かります。

Ⓐ方式では呼び出すメソッドがコンパイル時に決定する。
⇨ リモコンの型のメソッドが呼び出される。

Ⓑ方式では呼び出すメソッドが実行時に決定する。
⇨ リモコンが“現在”参照している回路の型のメソッドが呼び出される。

実際に生成されるコードは、Ⓑ方式です。実行結果がそのことを示しています。

クラス型変数が、派生関係にある様々なクラス型のインスタンスを参照できることを、**多相性＝ポリモーフィズム**（*polymorphism*）と呼びます。

▶ poly は『多くの』、morph は『形態』という意味です。多相性は、『多態性』『多様性』『同名異型』『ポリモルフィズム』などとも呼ばれます。

多相性が絡んだメソッド呼出しでは、プログラムの実行時に呼び出すメソッドが決定されます（**Fig.12-9**）。そのメリットは、以下のとおりです。

・異なるクラス型のインスタンスに対して同一のメッセージを送ることができる。
・メッセージを受け取ったインスタンスは自分自身の型が何であるかを知っており、適切な行動を起こす。

Ⓐ方式は、呼び出すべきメソッドをコンパイル時*に決定できるため、その呼出しメカニズムは、**静的結合**（*static binding*）や**早期結合**（*early binding*）と呼ばれます。

一方、Java で採用されているⒷ方式は、呼び出すべきメソッドが実行時に決定されるため、その呼出しメカニズムは、**動的結合**（*dynamic binding*）や**遅延結合**（*late binding*）と呼ばれます。

▶ binding には**束縛**という訳語が当てられることもあります。たとえば、動的結合は**動的束縛**や**遅延束縛**とも呼ばれます。

重要 メソッド呼出しでは、動的結合が行われる。

▶ C++ では、関数（メソッド）呼出しは、原則としてⒶ方式である静的結合です。特別に宣言された仮想関数のみが、動的結合となります。

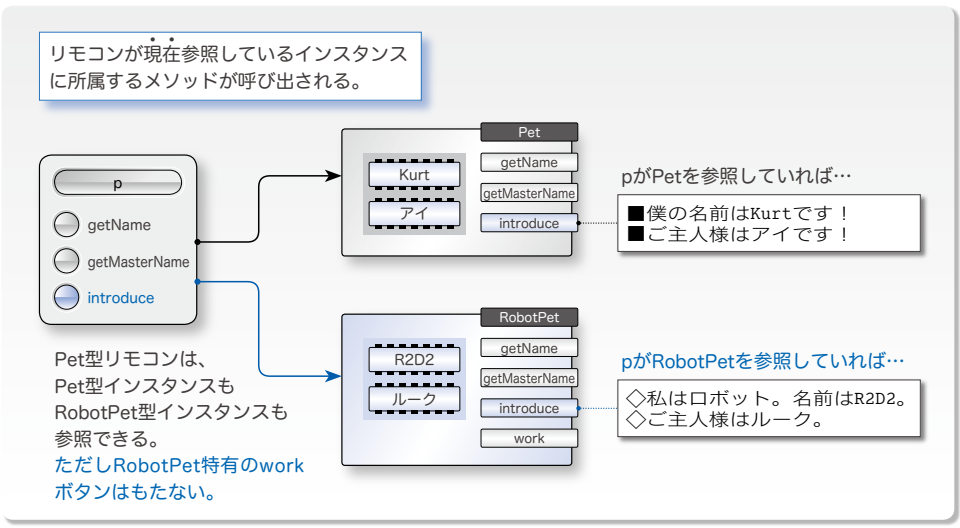


Fig.12-9 多相性と動的結合

動的結合をメソッドの引数に応用したプログラム例を、**List 12-10** に示します。

List 12-10 pet/PetTester2.java

```
// ペットクラスの利用例（メソッドの引数で多相性を検証）
class PetTester2 {

   // pが参照するインスタンスに自己紹介させる
   static void intro(Pet p) {
      p.introduce();
   }

   public static void main(String[] args) {
      Pet[] a = {
         new Pet("Kurt", "アイ"),
         new RobotPet("R2D2", "ルーク"),
         new Pet("マイケル", "英男"),
      };

      for (Pet p : a) {
         intro(p);                    // pが参照するインスタンスに自己紹介させる
         System.out.println();
      }
   }
}
```

実行結果

```
■僕の名前はKurtです！
■ご主人様はアイです！

◇私はロボット。名前はR2D2。
◇ご主人様はルーク。

■僕の名前はマイケルです！
■ご主人様は英男です！
```

メソッド *intro* の仮引数 *p* は *Pet* 型です。このメソッドが行うのは、*p* に対してメソッド *introduce* を起動することだけです。もちろん、仮引数 *p* には、*Pet* クラス型のインスタンスへの参照だけでなく、*Pet* クラスの下位クラスである *RobotPet* 型のインスタンスへの参照も受け取ることができます。

main メソッドでは、*Pet* 型のインスタンスと *RobotPet* 型のインスタンスが混在した配列を生成し、それらのインスタンスへの参照をメソッド *intro* に対して渡しています。

Pet 型のインスタンスである a[0] と a[2] に対しては、*Pet* 型のメソッド *introduce* が呼び出され、*RobotPet* 型のインスタンスである a[1] に対しては、*RobotPet* 型のメソッド *introduce* が呼び出されていることが、実行結果からも確認できます。

オブジェクト指向の三大要素

第 8 章から、クラスについて少しずつ学習してきました。そして、本章のここまでで、《継承》と《多相性》について学習しました。

以下の三つは、**オブジェクト指向の三大要素**と呼ばれます。

- クラス
- 継承
- 多相性

したがって、本書のこれまでの内容をマスターしていれば、**オブジェクト指向の基礎は身に付いている**ことになります。

参照型のキャスト

引き続き、クラス *Pet* とクラス *RobotPet* を考えます。以下に示すように、スーパークラス型の変数はサブクラスのインスタンスを参照できるのでしたね。

```
Pet p = new RobotPet("R2D2", "ルーク");
```

このとき、*RobotPet* 型への参照が、*Pet* 型への参照へと暗黙の内にキャストされていることに気付きましたか。ここで行われる型変換は、**参照型の拡大変換**（*widening reference conversion*）または**アップキャスト**（*up cast*）と呼ばれます（**Fig.12-10** a）。

もちろん、キャスト演算子を明示的に適用しても構いません。その場合、以下のようになります。

```
Pet p = (Pet)new RobotPet("R2D2", "ルーク");
```

▶ 拡大変換が暗黙の内に行われるのは、基本型の拡大変換（p.166）の場合と同じです。

一方、サブクラス型の変数はスーパークラスのインスタンスを参照できないのでしたね（p.403）。もっとも、以下のように**キャスト演算子を明示的に適用すれば、型変換は可能です。**

```
RobotPet r1 = new Pet("Kurt", "アイ");                    // エラー
RobotPet r2 = (RobotPet)new Pet("Kurt", "アイ");          // ＯＫ！
```

ここで行われる型変換は、**参照型の縮小変換**（*narrowing reference conversion*）あるいは**ダウンキャスト**（*down cast*）と呼ばれます（図b）。

▶ 参照型の拡大変換／縮小変換という名称は、Java の文法用語で、アップキャスト／ダウンキャストは、一般的なプログラミングの用語です。
なお、アップ／ダウンは、変換先がクラス階層の上位／下位であることに由来します。

r2 の参照先はロボットではないペットです。そのため、以下に示す家事の命令は、プログラムの実行時にエラーとなります。

```
r2.work(0);        // 実行時エラー
```

不用意にダウンキャストを行って、下位クラス型の変数に、上位クラス型のインスタンスを参照させるようなことは、原則として避けるべきです。

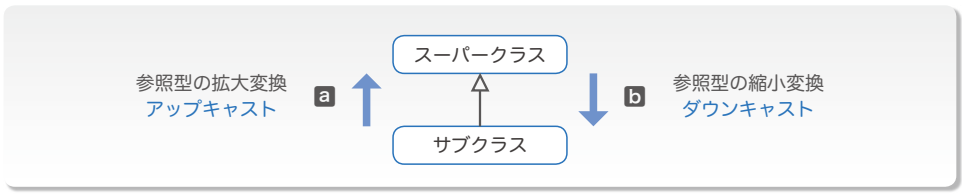


Fig.12-10 アップキャストとダウンキャスト

instanceof 演算子

クラス型の変数は、そのクラス型のインスタンスだけでなく、上位クラスのインスタンスを参照することも、下位クラスのインスタンスを参照することもできることが分かりました。

そうすると、変数が参照しているのが、どのクラスなのかを調べることが必要となることもあります。そのようなプログラム例を **List 12-11** に示します。

List 12-11 pet/PetInstanceOf.java

```
// instanceof演算子の利用例

class PetInstanceOf {

   public static void main(String[] args) {
      Pet[] a = {
         new Pet("Kurt", "アイ"),
         new RobotPet("R2D2", "ルーク"),
         new Pet("マイケル", "英男"),
      };

      for (int i = 0; i < a.length; i++) {
         System.out.println("a[" + i + "] ");
         if (a[i] instanceof RobotPet)     // a[i]がロボットであれば…
            ((RobotPet)a[i]).work(0);      //   家事（掃除）
         else                              // そうでなければ…
            a[i].introduce();              //   自己紹介
      }
   }
}
```

実行結果

```
a[0]
■僕の名前はKurtです！
■ご主人様はアイです！
a[1]
掃除します。
a[2]
■僕の名前はマイケルです！
■ご主人様は英男です！
```

網かけ部に着目しましょう。初登場の **instanceof** 演算子を利用しています。これは、以下の形式で利用する、一種の関係演算子です（**Table 12-2**）。

クラス型変数名 **instnaceof** クラス名

本プログラムでは、*a*[*i*] の参照先がクラス *RobotPet* のインスタンスであれば、家事を命じ、そうでなければ自己紹介を命じるようになっています。なお、家事を命じる際は、*a*[*i*] を *RobotPet* 型への参照型にダウンキャストした上で行う必要があります。

Table 12-2 instanceof演算子

x **instanceof** *t*	変数 *x* が型 *t* に暗黙の内にキャストできる下位クラスであれば **true** を、そうでなければ **false** を生成

▶ なお、本プログラムの **if** 文を以下のようにすることはできません。*a*[*i*] の参照先が *Pet* であっても *RobotPet* であっても、式 *a*[*i*] **instanceof** *Pet* を評価した値が **true** となってしまうからです。

```
if (a[i] instanceof Pet)      // Petを含めてPetの下位クラスであればtrueとなる
   a[i].introduce();          // すなわちa[i]がPetでもRobotPetでも実行される
else
   ((RobotPet)a[i]).work(0);  // 実行されない
```

@Override アナテイション

以下に示すプログラム部分を見てください。これは、クラス *RobotPet* と、その自己紹介のメソッドです。*introduce* とすべきメソッド名を、*introduction* に間違えてしまっていることに注意しましょう。

```
class RobotPet {
   // … 中略 …
   public void introduction() {                // 自己紹介
      System.out.println("◇私はロボット。名前は" + getName() + "。");
      System.out.println("◇ご主人様は" + getMasterName() + "。");
   }
   // … 中略 …
}
```

私たち人間にとっては、単なる綴り間違いです。しかし、頭の固いコンパイラは、『クラス *RobotPet* では、メソッド *introduction* が新規に宣言されている』とみなします。

そのため、スーパークラスで定義された *introduce* はそのまま継承され、ここで定義された *introduction* は、新しく追加されたメソッドとして扱われます。

▶ その場合、**List 12-9** の *p.introduce()* では、クラス *RobotPet* ではなく、クラス *Pet* の自己紹介メソッドが呼び出されることになります。

*

このような人為的なミスを防ぐのに有効なのが、**アナテイション**（*annotation*）です。第 1 章では、プログラムの読み手に伝えるべきことがらをコメントとして記入することを学習しましたね。コメントの対象は、プログラムの作成者を含めた人間です。

一方、アナテイションは、もう少し高度な注釈です。**私たち人間だけでなく、コンパイラにも読ませる注釈です。**

重要 人間とコンパイラの両方に伝えるべきコメントは、アナテイションとして記述せよ。

メソッドのオーバライドの際に利用するのが **@Override** アナテイションです。使い方は簡単です。以下のように、メソッド宣言の名前の前に **@Override** を付けるだけです。

```
class RobotPet {
   // … 中略 …
   @Override public void introduction() {          // 自己紹介
      System.out.println("◇私はロボット。名前は" + getName() + "。");
      System.out.println("◇ご主人様は" + getMasterName() + "。");
   }
   // … 中略 …
}
```

このアナテイションは、人間とコンパイラに対して、以下のことを表明します。

> これから宣言するのは、上位クラスのメソッドをオーバライドするメソッドです。本クラスで新しく追加するメソッドではありませんよ。

この場合、スーパークラス *Pet* にメソッド *introduction* がないため、コンパイラは以下のエラーを発します。

> メソッドはそのスーパークラスのメソッドをオーバライドしません。

▶ これは、コンパイラが表示するメッセージです。直訳調のため、意味が分かりにくいですね。『メソッド introduction をオーバライドすると宣言されていますが、スーパークラスには、そのような名前のメソッドはありません。』という警告である、と理解しましょう。

これで、私たちプログラマは、メソッド名のタイプミスに気付いて、プログラムを修正できることになります。

以下のように修正すると、正しくコンパイルできるプログラムとなります。

```
class RobotPet {
   // … 中略 …
   @Override public void introduce() {              // 自己紹介
      System.out.println("◇私はロボット。名前は" + getName() + "。");
      System.out.println("◇ご主人様は" + getMasterName() + "。");
   }
   // … 中略 …
}
```

重要 スーパークラスのメソッドをオーバライドするメソッドには、**@Override** アナテイションを付けて宣言するとよい。

▶ annotation は、『注釈』あるいは『注解』といった意味の語句です。多くのテキストでは、『アノテーション』と表記されているようです（ちなみに、発音は ǽnətéiʃən です）。

□ 演習 12-2

定期預金付き銀行口座クラス型変数 *a*, *b* の普通預金と定期預金残高の合計額を比較した結果を返却するメソッド *compBalance* を作成せよ。

static int *compBalance*(*TimeAccount a*, *TimeAccount b*)

合計額を比較して、*a* のほうが多ければ 1、等しければ 0、*b* のほうが多ければ -1 を返却すること。もし *a* や *b* の参照先が、定期預金をもたない *Account* 型のインスタンスであれば、普通預金の金額を比較の対象とすること。

Column 12-4　@Deplicate アナテイション

ここで学習した **@Override** 以外にも、いくつかのアナテイションが標準で用意されています（アナテイションは、自作することもできます）。

クラスやメソッドに改良を重ねるうちに、『よりよいクラスを作成した』『クラスの内部的な仕様変更などによって、このメソッドは使うべきでなくなった』といった状況が生じることがあります。そのような際に便利なのが **@Deplicate** アナテイションです。

利用が推奨されないクラスやメソッドの前に **@Deplicate** を付けておきます。そうすると、それを利用しようとするプログラムのコンパイル時に警告が発生されます。

12-3 継承とアクセス性

クラスの派生において、フィールドやメソッドは継承されるものの、コンストラクタは継承されないのでしたね。本節では、クラスの派生において、どの資産が継承されて、どの資産が継承されないのか、また、それらのアクセスがどのようになるのかをきちんと学習します。

メンバ

12-1 節では、フィールドやメソッドが継承される一方で、コンストラクタが継承されないことを学習しました。クラスの派生において、何が継承されて、何が継承されないのかを明確に知っておく必要があります。

クラスの派生で継承されるのは、クラスの**メンバ**（*member*）に限られることになっています。クラスのメンバを以下に示します。

- フィールド
- メソッド
- クラス
- インタフェース

▶ ここでの『クラス』と『インタフェース（第 14 章）』は、通常のクラスやインタフェースのことでなく、クラスの中で宣言されるクラスやインタフェースのことです（『入門編』の範囲を越えるため、本書では学習しません）。

スーパークラスのメンバは、原則としてそのまま継承されます。ただし、非公開アクセス性をもつメンバ、すなわち **private** 宣言されたメンバは、継承されません。

重要 非公開メンバは継承されない。

もし、サブクラスからスーパークラスの非公開（**private**）メンバを自由にアクセスできるとしたらどうなるでしょう。クラスの派生を行うだけで、スーパークラスの非公開部にアクセスできることになってしまいます。これでは、情報隠蔽どころではありません。

▶ なお、**private** メンバが継承されないといっても、そのメンバが消滅するわけではありません。プログラム上からはアクセスできなくなるものの、内部的には存在しています。

*

メンバではない、クラスの資産には、以下に示すものがあります。

- インスタンス初期化子
- 静的初期化子
- コンストラクタ

これらの資産は継承されません。

重要 派生において、メンバではない、インスタンス初期化子・静的初期化子・コンストラクタは継承されない。

final なクラスとメソッド

final 付きで宣言されたクラスやメンバは、派生において特別な扱いを受けます。

final クラス

final クラスから派生を行うことはできないことになっています。すなわち、**final** クラスをスーパークラスとするクラスを作ることはできないのです。

たとえば、文字列を表す **String** クラスは、**final** クラスです。したがって以下のように、**String** クラスを拡張したクラスを作ることはできません。

```
class DeluxeString extends String {    // エラー
    // ...
}
```

このことは、以下の原則を示しています。

重要 拡張すべきでない（勝手にサブクラスを作られると困る）クラスは、**final** クラスとして宣言せよ。

final メソッド

final メソッドは、サブクラスでオーバライドすることができません。以下のように、メソッドの先頭に **final** を付けて宣言します。

```
final void f() { /*…*/ }
```

重要 サブクラスでオーバライドされるべきでないメソッドは、**final** メソッドとして宣言せよ。

なお、**final** クラスのメソッドは、すべて **final** メソッドとなります。

▶ p.43 でも解説したように、**final** には『最後の』という意味があるのでしたね。**final** クラスや **final** メソッドは、『最終決定版であり、もはや拡張したり上書きしたりできない』というニュアンスがあります。

オーバライドとメソッドのアクセス性

メンバのアクセス性について、前章で学習しました（p.382）。メソッドをオーバライドする際は、アクセス性と関連する、以下の規則を必ず知っておかなければなりません。

重要 メソッドをオーバライドする際は、上位クラスのメソッドと同等もしくは弱いアクセス制限をもつ修飾子を与えなければならない。

アクセス制限の強さ／弱さの関係を示したのが **Fig.12-11** です。修飾子のアクセス制限が最も弱いのが、公開（public）アクセス制限をもつ *m1* であり、最も強いのが非公開（private）アクセス制限をもつ *m4* です。

弱い（ゆるい）
強い（きつい）

```
public class A {
    public    void m1() { }  // 公開（public）
    protected void m2() { }  // 限定公開（protected）
              void m3() { }  // パッケージ（デフォルト・修飾子無し）
    private   void m4() { }  // 非公開（private）
}
```

Fig.12-11 メソッドのアクセス制限

下位クラスでメソッドをオーバライドする際は、上位クラスのものよりも強いアクセス制限の修飾子を与えることはできません。その規則をまとめたのが **Table 12-3** です。

Table 12-3 オーバライドするメソッドに与えることのできるアクセス制限（修飾子）

A ＼ B	公開	限定公開	パッケージ	非公開
公開	○	×	×	×
限定公開	○	○	×	×
パッケージ	○	○	○	×
非公開	×	×	×	×

※A … スーパークラスにおけるメソッドのアクセス制限（修飾子）。
　B … サブクラスでオーバライドするメソッドのアクセス制限（修飾子）。

たとえば、クラス A から派生したサブクラスでメソッド *m1* をオーバライドする際は、必ず宣言に **public** を付けなければなりません。**public** を付けなければコンパイルエラーとなります。

また、メソッド *m2* をオーバライドする際は、必ず **public** あるいは **protected** を付ける必要があります。

▶ 派生元スーパークラスと派生したサブクラスのそれぞれが **public** であるかどうかとは関係なく、ここに示した規則に基づいて、**public** や **protected** などのアクセス修飾子を与えます。

なお、非公開メソッドはそもそも継承されないため、当然オーバライドすることはできません。

ただし、クラス *A* から派生したサブクラスでメソッド *m4* を定義することができます。というのも、以下の規則があるからです。

> **重要** 非公開メソッドを、下位クラスで同一シグネチャ・同一返却型のメソッドとして定義しても、オーバライドではなく、たまたま同じ仕様の無関係なメソッドとなる。

*

第 9 章では、『文字列表現を返すメソッド `toString` を定義する際は **`public`** メソッドとするように』という指針を示していました（p.319）。

Java のすべてのクラスの親玉である **`Object`** クラスでは、`toString` が **`public`** メソッドとして定義されています（**Column 12-6**：次ページ）。したがって、`toString` クラスをオーバライドする際は、必ず **`public`** を付ける必要があるのです。

> **重要** メソッド **`String`** `toString()` は、**`public`** 修飾子を付けて定義しなければならない。

*

なお、スーパークラスのクラスメソッドを、インスタンスメソッドとしてオーバライドすることはできません。したがって、以下のプログラムはエラーとなります。

```
class A {
   static void f() { /* … */ }
}

class B extends A {
   void f() { /* … */ }              // コンパイルエラー
}
```


Column 12-5　宣言における修飾子の順序

クラスやフィールドなどの宣言では、アナテイションや **`public`** や **`final`** などの修飾子で属性を指定します。修飾子が複数の場合は、どんな順序で指定しても構わないのですが、**Table 12C-1** の順で指定することが推奨されています（この表は、本書で学習しない修飾子も含んでいます）。

Table 12C-1　宣言に与える修飾子（推奨される順序）

クラス	アナテイション **`public protected private abstract static final strictfp`**
フィールド	アナテイション **`public protected private static final transient volatile`**
メソッド	アナテイション **`public protected private abstract static final synchronized native strictfp`**
インタフェース	アナテイション **`public protected private abstract static strictfp`**

Column 12-6　Object クラス

Java の全クラスの親玉であるクラス **Object** の定義を、**List 12C-2** に示します。

本書では学習しない技術も使われていますので、このプログラムを完全に理解する必要はありません。いくつかの重要な点を理解していきましょう。

List 12C-2　API/Object.java

```
// Objectクラス
package java.lang;
public class Object {
   static {
      registerNatives();
   }
   public final native Class<?> getClass();
   public native int hashCode();                                          1
   public boolean equals(Object obj) {
      return (this == obj);                                               2
   }
   protected native Object clone() throws CloneNotSupportedException;
   public String toString() {
      return getClass().getName() + "@" + Integer.toHexString(hashCode()); 3
   }
   public final native void notify();
   public final native void notifyAll();
   public final native void wait(long timeout) throws InterruptedException;
   public final void wait(long timeout, int nanos) throws InterruptedException {
      if (timeout < 0) {
         throw new IllegalArgumentException("timeout value is negative");
      }
      if (nanos < 0 || nanos > 999999) {
         throw new IllegalArgumentException(
                  "nanosecond timeout value out of range");
      }
      if (nanos >= 500000 || (nanos != 0 && timeout == 0)) {
         timeout++;
      }
      wait(timeout);
   }
   public final void wait() throws InterruptedException {
      wait(0);
   }
   protected void finalize() throws Throwable { }
}
```

これは定義の一例です。Javaのバージョンやプラットフォームによって、定義は異なります。

① java.lang パッケージに所属する

Object クラスは、**java.lang** パッケージに所属します。そのため、Java のすべてのプログラムにおいて、明示的に型インポートすることなく単純名で表せます。

② native メソッド

getClass など、いくつかのメソッドが **native** 付きで宣言されています。このように宣言されたメソッドは、MS-Windows、Mac OS-X、Linux などのプラットフォーム（環境）に依存する部分を実現するための特殊なメソッドです。一般には、Java 以外の言語で記述されて実現されます。

③ hashCode メソッドとハッシュ値

すべてのクラス型のインスタンスは、**ハッシュ値**と呼ばれる **int** 型の整数値を計算できることになっています。ハッシュ値を返却するのが、hashCode メソッドです（1）。

ハッシュ値とは、個々のインスタンスを区別する《識別番号》のようなものです。計算方法は任意ですが、同一状態の（全フィールドの値が同じである）インスタンスには同一のハッシュ値を与えて、異なる状態のインスタンスには異なるハッシュ値を与えるように計算するのが一般的です。

たとえば、第９章の日付クラス *Day* に対しては、hashCode メソッドを以下のように定義することができます。

```
public int hashCode() {
   return (year * 372) + (month * 31) + date;
}
```

同じ日付（フィールド *year*, *month*, *date* の値がすべて等しい日付）のインスタンスのハッシュ値は同じ値となり、異なる日付のインスタンスのハッシュ値は異なる値となりますね。

▶ 西暦１年１月１日からの経過日数を厳密に求める式としてもよいのですが、計算に時間がかかってしまいます。日付クラスに限らず、上記のように素早く計算できる簡易的な式で実現するのが一般的です（計算式中の 372 は、12 × 31 の値です）。

④ equals メソッドとインスタンスの等価性

equals メソッド（2）は、参照先のインスタンスが“等しい”かどうかを判定するメソッドです。等しければ **true** を、そうでなければ **false** を返します。

このメソッドで行う判定は、**ハッシュ値との整合性がとれていることが原則です。***a*.equals(*b*) が **true** となる場合は、*a* と *b* のハッシュ値（*a*.hashCode() と *b*.hashCode() の返却値）が同じ値となり、**false** となる場合は、*a* と *b* のハッシュ値が異なっている必要があります。**equals メソッドを定義する際は、それに合わせて hashCode メソッドも定義しなければなりません。**

第９章の日付クラス *Day* で equals メソッドを定義せずに、ニセモノの *equalTo* メソッドを定義していたのは、hashCode メソッドをオーバライドしていなかったからです（それだけでなく、**Object** クラスや派生について学習していなかったという理由もあります）。

日付クラスの equals メソッドの定義例を以下に示します。

```
public boolean equals(Object obj) {
   if (this == obj)            // 比較対象が自分自身であれば…
      return true;
   if (obj instanceof Day) {   // objがDayクラス（の下位クラス）型であれば…
      Day d = (Day)obj;
      return (year == d.year && month == d.month && date == d.date) ? true
                                                                    : false;
   }
   return false;
}
```

ここに示した hashCode メソッドを追加して、メソッド *equalTo* の代わりに、equals メソッドを定義して改良した日付クラス【第５版】は、ホームページからダウンロードできるソースプログラム（p.5）のディレクトリ day5 に入っています。

⑤ toString メソッド

Object クラスの toString メソッドは、"クラス名@ハッシュ値" を返却します（3）。

自作のクラスでこのメソッドをオーバライドする際は、クラスの特性やインスタンスの状態を表す、適切な文字列を返却するように定義します。

まとめ

- クラスの**派生**によって、既存クラスの資産を**継承**したクラスを簡単に作ることができる。派生元のクラスを**スーパー（上位）クラス**と呼び、派生によって作られたクラスを**サブ（下位）クラス**と呼ぶ。派生は、クラスに《血縁関係》を与える。

- 明示的な派生の宣言をしないクラスは、`Object` クラスのサブクラスとなる。そのため、Java のすべてのクラスは、`Object` クラスの下位クラスである。

- クラスの派生において、コンストラクタは継承されない。

- コンストラクタの先頭では、`super(...)` によってスーパークラスのコンストラクタを呼び出せる。明示的に呼び出さなければ、スーパークラスの“引数を受け取らないコンストラクタ”の呼出し `super()` が、コンパイラによって自動的に挿入される。

- コンストラクタを 1 個も定義しないクラスには、`super()` のみを実行する**デフォルトコンストラクタ**が、コンパイラによって自動的に定義される。スーパークラスが“引数を受け取らないコンストラクタ”をもっていなければ、コンパイルエラーとなる。

- スーパークラスの非公開でないメンバは、“`super.` メンバ名”によってアクセスできる。

- クラス *B* がクラス *A* の下位クラスであるとき、『クラス *B* は**一種の** *A* である。』と表現し、これを、is-A の関係と呼ぶ。

- スーパークラス型の変数がサブクラス型のインスタンスを参照可能であることを利用すると、**多相性**を実現できる。多相性が絡んだメソッド呼出しでは、呼び出すべきメソッドがプログラム実行時に決定される**動的結合**が行われる。

- サブクラス型の変数は、キャストしない限りスーパークラス型のインスタンスを参照できない。

- オーバライドするメソッドには、上位クラスのものと同等もしくは弱いアクセス制限をもつ修飾子を与えなければならない。そうでなければ、コンパイルエラーとなる。

- スーパークラスのメソッドを**オーバライド**する（スーパークラスとは異なる定義を与えて上書きする）メソッドには、**@Override アナテイション**を付けて宣言するとよい。アナテイションは、人間とコンパイラの両方に伝えるコメントである。

Objectクラスのサブクラス。

Chap12/Member.java

```
//--- 会員クラス ---//
public class Member {
    private String name;     // 名前
    private int no;          // 会員番号
    private int age;         // 年齢

    public Member(String name, int no, int age) {
        this.name = name;  this.no = no;  this.age = age;
    }

    public String getName() {
        return name;
    }

    public void print() {
        System.out.println("No." + no + "：" + name +
                           "（" + age + "歳）");
    }
}
```

継承される

クラスMemberのサブクラス。

Chap12/SpecialMember.java

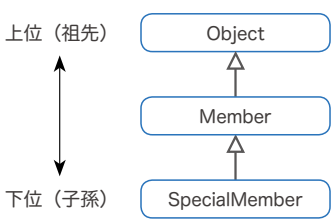
```
//--- 優待会員クラス ---//
public class SpecialMember extends Member {
    private String privilege;     // 特典

    public SpecialMember(String name, int no, int age, String privilege) {
        super(name, no, age);  this.privilege = privilege;
    }

    @Override public void print() {
        super.print();
        System.out.println("特典：" + privilege);
    }
}
```

スーパークラスのコンストラクタ／メソッドの呼び出し

オーバライド（上書き）



すべてのクラスの上位クラス。
java.langパッケージに所属する。

MemberはObjectから派生。
スーパークラスはObject。SpecialMemberはサブクラス。

SpecialMemberはMemberから派生。
スーパークラスはMember。

Chap12/MemberTester.java

```
//--- 会員クラスのテスト ---//
public class MemberTester {

    public static void main(String[] args) {
        Member[] m = {
            new Member("橋口", 101, 27),
            new SpecialMember("黒木", 102, 31, "会費無料"),
            new SpecialMember("松野", 103, 52, "会費半額免除"),
        };

        for (Member k : m) {
            k.print();
            System.out.println();
        }
    }
}
```

Member型変数は、Memberだけでなく
SpecialMemberも参照可能。

動的結合：参照先のクラス型の
メソッドが呼び出される。

実行結果

```
No.101：橋口（27歳）
No.102：黒木（31歳）
特典：会費無料
No.103：松野（52歳）
特典：会費半額免除
```