第20章　継承

目次

- 継承
- protected修飾子
- オーバーライド
- Objectクラスの継承

継承とは

メンバ(フィールドやメソッド)を、
他のクラスに引き継がせる機能のこと。

- 継承元のクラス:スーパークラス(基底クラス)
- 継承先のクラス:サブクラス(派生クラス)

サブクラスは、スーパークラス内で定義されたメンバに対して、
自分のクラスのメンバであるかのようにアクセスできる。

継承とは

サブクラスの宣言を行う場合、
extendsという予約語の後ろに
スーパークラス名を指定する。

継承とは

```
class サブクラス名 extends スーパークラス名 {
    サブクラスに追加するフィールド
    サブクラスのコンストラクタ(引数リスト){
            ・・・
    }

    戻り値の型 サブクラスに追加するメソッド(引数リスト) {
            ・・・
    }
}
```

継承とはメンバ(フィールドやメソッド)を他のクラスに引き継がせる機能のことです。
継承するには、extendsの後ろにスーパークラス名を指定します。

【Sample2001 サブクラスのオブジェクトを作成する】を作成しましょう

Sample2001のポイント

Sample2001クラスのmain()メソッド内では、
SmartPhone2001クラスのオブジェクトを生成し、
そのオブジェクトからsetFeeメソッド、
showFeeメソッドを呼び出している。

```
// スマートフォンの料金設定するメソッドを呼びだす
    sp.setFee(5000);
 // スマートフォンの料金表示するメソッドを呼びだす
    sp.showFee();
```

Sample2001のポイント

SmartPhone2001クラス内にはsetFeeメソッド、showFeeメソッドは定義されていない。

```
public class SmartPhone2001 extends Phone2001 {
    public void smartPhoneFunction() {
    }
}
```

SmartPhone2001クラスは、Phone2001クラスと継承関係にあるため、Phone2001クラスを自分のメンバとして利用できる。

```
sp.setFee(5000);
sp.showFee();
```

Sample2001のポイント

サブクラス内にはサブクラス独自のメンバのみを
記述すればよく、スーパークラスと同内容の
メンバを定義する必要はない。

継承を活用することで…
- 冗長性を抑えたプログラムが作成できる。
- 作業の効率化やソースコード全体の可読性向上につながる。

闇雲な継承はNG

継承の条件：
「〇〇は××だ（〇〇は××の一種だ）」と言えること。

この関係を「is-a関係」と言う。
「is-a関係」が成り立たない場合、継承を行うべきではない。

（サンプルコードの場合）
〇「スマートフォンは携帯電話だ」

闇雲な継承はNG

下記のSample2001とPhone2001には
共通の「状態」「機能」がなく、
「サンプルは携帯電話だ」が成り立たない。
「is-a関係」が成り立っていないので、継承は避けるべき。

```
public class Sample2001 extends Phone2001
```

「is-a関係」が成り立たない場合、継承を行うべきではありません。

暗黙的なスーパークラスのコンストラクタ呼び出し

継承では、スーパークラスのコンストラクタを呼び出すことが可能。

コンストラクタを呼び出すパターン

- サブクラスのオブジェクト生成時、暗黙的に引数なしのコンストラクタ（またはデフォルトコンストラクタ）が呼び出される。
- サブクラスのコンストラクタ内で「super()」という処理を実行することで、スーパークラス内の特定のコンストラクタが呼び出される。

【Sample2002 コンストラクタを呼び出す】を作成しましょう

Sample2002のポイント

Sample2002クラスにて、SmartPhone2002クラスのオブジェクト生成したとき、Phone2002クラスとSmartPhone2002クラスのコンストラクタが順に呼び出されている。

```
Phoneクラスのコンストラクタが呼ばれました。
SmartPhoneクラスのコンストラクタが呼ばれました。
```

Sample2002のポイント

サブクラスのオブジェクト生成時には、
以下の順でコンストラクタが実行される。

1.暗黙的にスーパークラスの引数なしのコンストラクタが実行
2.サブクラスのコンストラクタが実行

明示的なスーパークラスのコンストラクタ呼び出し

スーパークラスで引数ありのコンストラクタが定義されている場合、
明示的にコンストラクタを呼び出すことができる。
明示的に呼び出すには、サブクラスのコンストラクタの
先頭行に下記構文に従って処理を記述する。

```
super(引数リスト);
```

明示的なスーパークラスのコンストラクタ呼び出し

super()は、スーパークラスのコンストラクタを指す。
この処理が実行されると、スーパークラスの
コンストラクタが呼び出される。
super()に引数を指定することで、
引数ありのコンストラクタが呼び出される。
super()は必ずコンストラクタの先頭行に記述する。
以外の場所に記述するとコンパイルエラーが発生する。

【Sample2003 super()を呼び出す】を作成しましょう

protected修飾子

スーパークラスのフィールドをprivateメンバにした場合…
- 他のクラスからそのメンバはアクセスできない
- サブクラスからもアクセスできない。

サブクラスからメンバにアクセスしたい場合、
protected修飾子を指定する。
サブクラスからもアクセスできるようになる。

【Sample2004 protectedメンバにアクセスする】を作成しましょう

Sample2004のポイント

スーパークラス内のprotectedメンバは、
サブクラスからアクセスできる。

SmartPhone2004クラスでは、
スーパークラスのフィールドdataにアクセスしている。

```
data *= 2;
```

protected修飾子が付与されたメンバは、
サブクラスからアクセスできます。

オーバーライドされたメソッドの利用

オーバーライド:
スーパークラスで定義されたメソッドと
同じシグネチャのメソッドをサブクラスで再定義すること。

シグネチャ:
メソッド名・引数の型・個数・順番の組合せを指す言葉。

オーバーライドされたメソッドの利用

オーバーライドすることで、サブクラス内で、
スーパークラスで定義されたメソッドと
同じメソッド名、引数の型、個数、順番である
メソッドを定義できる。

サブクラス内でオーバーライドしたメソッドは、
スーパークラス内のメソッドと異なる処理内容を定義できる。

オーバーライドとは、サブクラス内でスーパークラスで
定義しているメソッドと同じ名前で再定義することです。

オーバーライドできない場合

1.オーバーライド元のメソッドにfinalが指定されている
　処理内容を変更させたくないメソッドには
　スーパークラス側 のメソッドにfinalをつける。

2.staticメソッド
　オーバーライドできるのはインスタンスメソッドのみ。

【Sample2005 メソッドのオーバーライド】を作成しましょう

Sample2005のポイント

サブクラス内でオーバーライドされたメソッドがある場合、
サブクラスのオブジェクトからは優先的にそのメソッドが
呼び出される。

```
携帯電話を生成しました。
携帯電話の料金は、5000円です。
データ通信量は、2.0GBです。
携帯電話を生成しました。
スマートフォンを生成しました。
液晶画面のサイズは5インチです。
```

オーバーライドされた
showメソッド()

明示的なスーパークラスのメソッド呼出し

オーバーライドしてもスーパークラスで定義したメソッドが
使えなくなるわけではなく、呼び出して使用することができる。
スーパークラスのメンバを呼び出す際は、
「super.」というキーワードを使って、下記のように記述する。

```
super.メンバ名
```

明示的なスーパークラスのメソッド呼出し

SmartPhone2005クラスのshow()メソッドに
下記の下線部の処理を追記する。

```
public void show() {
    System.out.println(″液晶画面のサイズは、″ + inch+ ″インチです。″);
super.show();
    System.out.println(″購入したスマートフォンの色は黒です。″);
}
```

明示的なスーパークラスのメソッド呼出し

SmartPhone2005クラスのshow()メソッド内で
「super.show();」と記述すると、
スーパークラス（Phoneクラス）のshow()メソッドが呼び出される。

スマートフォンを生成しました。
液晶画面のサイズは5インチです。
携帯電話の料金は、5000円です。
データ通信量は、2.0GBです。
購入したスマートフォンの色は黒です。

Phoneクラスの
showメソッド()

明示的なスーパークラスのメソッド呼出し

「super.」を用いることでメソッドを再利用し、
可読性を上げることができる。

サブクラスで再定義している、
スーパークラスと同じ名前のメンバにアクセスする場合は、super.をつけます。

変数の型によるメソッド呼び出しへの影響

サブクラスから生成されたオブジェクトは、
スーパークラスの型の変数に代入できる。
（例）SmartPhone2005クラスのオブジェクトを
　　Phone2005型の変数に代入

```
// 携帯電話クラスのオブジェクトを生成
Phone2005 p = new SmartPhone2005();
```

変数の型によるメソッド呼び出しへの影響

サブクラスから生成したオブジェクトを、
スーパークラスの型の変数に代入した場合、
変数を通してオブジェクトから以下のメンバを呼び出せる。

- スーパークラスからオーバーライドしたメソッド
- スーパークラスで定義されたメソッド
  （アクセス可能な制限の場合）
- スーパークラスで定義されたフィールド
  （アクセス可能な制限の場合）

サブクラス内で独自に定義されたフィールド、
メソッドにはアクセスできない。

オーバーライドを活用するメリット

メソッドの作成者はスーパークラスのメソッドを
再利用でき、コードの重複を防ぐことができる。

将来の変更があった場合も、共通部分は
スーパークラスに定義しておくことで柔軟に編集できる。

オーバーライドを活用するメリット

オーバーライドされたメソッドを呼び出すと、
そのメソッドがスーパークラスとサブクラスの
どちらのオブジェクトに属するかで処理内容が変わる。

1つの名前のメソッドを呼び出した際に状況に応じて
振る舞いが変わること：多態性

多態性を活用することで、把握しておくべきメソッドが
最小限に抑えられ、メソッドの管理が容易になる。

【Sample2006 スーパークラスの配列を利用する】を作成しましょう

Sample2006のポイント

スーパークラスの型の配列を作り、スーパークラスと
サブクラスのオブジェクトをそれぞれ代入している。

```
Phone2006[] phones = new Phone2006[2];
phones[0]= new Phone2006();
phones[1]= new SmartPhone2006();
```

サブクラスのオブジェクトは
スーパークラス型の変数に代入できる。
Phone2006クラス型の配列1つで、Phone2006クラスと
SmartPhone2006クラスのオブジェクトをまとめて管理できている。

Sample2006のポイント

処理の最後では、拡張for文でスーパークラスである
Phone2006クラス型の変数にPhone2006と
SmartPhone2006のオブジェクトを代入し、
show()メソッドを実行している。

```
for (Phone2006 phone : phones) {
phone.show();
}
```

Sample2006のポイント

実行結果では、Phone2006クラスのshow()メソッドと
SmartPhone2006クラスのshow()メソッドが実行されている。

```
携帯電話の料金は、5000円です。
データ通信量は、2.0GBです。
購入したスマートフォンの色は黒です。
液晶画面のサイズは5インチです。
携帯電話の料金は、8000円です。
データ通信量は、5.0GBです。
```

オーバーライドされたメソッドを実行する場合、
変数のクラスではなく、変数に代入されている
オブジェクトのクラスによって、実行されるメソッドが決まる。

final修飾子

オーバーライド元のメソッドにfinal修飾子が指定されている場合、
オーバーライドすることができない。(コンパイルエラーとなる)

スーパークラス側のメソッドにfinal修飾子を指定することで、
オーバーライドによる意図せぬ変更を防ぐことができる。

final修飾子でコンパイルエラーとなる例

```
public class Phone{
    public final void show() {
        System.out.println("携帯電話の料金は、" + fee + "円です。");
        System.out.println("データ通信量は、" + data + "GBです。");
    }
}
```

```
public class SmartPhone extends Phone {
    public void show() {
        System.out.println("液晶画面のサイズは" + inch + "インチです。");
    }
}
```

showメソッド()にfinal修飾子が指定されているため、コンパイルエラーとなる

final修飾子をクラスに指定した場合

そのクラスは継承禁止となる。

（例）Mainクラスにfinal修飾子を指定

```
public final class Main {
    public static void main(String[] args) {
        // 処理
    }
}
```

final修飾子を変数に指定した場合

その変数は「定数」となる。宣言と同時に初期値が
代入された後は、値を書き換えることができない。
（例）定数PIにfinal修飾子を指定

```
final double PI = 3.14;
```

final修飾子を変数に指定した場合

final修飾子の指定された定数PIの値を
途中で変更しようととすると、コンパイルエラーとなる。

```
public class Main {
    public static void main(String[] args) {
        final double PI = 3.14;  //定数として円周率を宣言
        System.out.println("円周率を変更します。");
    PI = 3;
}
}
```

定数名を記述する場合の命名ルール

- 変数名は全て大文字する。
- 複数単語になる場合、単語と単語を「_」でつなぐ

Object クラスとは

全てのクラスの継承元にあたるクラス。
Javaの全てのクラスは、Objectクラスを
暗黙的に継承している。

Object クラスとは

Objectクラスはextendsを記述しなくても継承される。
したがって、全てのクラスは・・・
・Objectクラスのメンバを呼び出せる
・Objectクラスのメソッドをオーバーライドできる

継承のルール上、サブクラスを他のクラスが継承できる。
その場合、サブクラスを継承したサブクラスは、
大元のスーパークラスまでの一連のクラスの
メンバを継承できる。

Object クラスとは

Objectクラスを継承したクラスが
他の継承関係にあるという場合でも、
一番下の階層のサブクラスは
Objectクラスのメンバを継承でき、
かつオーバーライドできる。

全てのクラスは、Objectクラスを継承しています。

Objectクラスの代表的なメソッド

メソッド名	機能
`String toString()`	オブジェクトが表す文字列を返す
`boolean equals(Object obj)`	オブジェクトが引数と同じものであるかどうかを調べる

toString()メソッド

文字列化する（Stringクラス型の文字列に変換する）」ためのメソッド。

代表的な利用場所はprintln()メソッド、
print()メソッドなどの標準出力処理

```
//「12」という文字列が出力される
System.out.println(12);
```

toString()メソッド

println()メソッドは、処理中にObjectクラスから
継承したtoString()メソッドを呼び出している。

1. 引数で受け取った値をtoString()メソッドに渡して、
   文字列に変換。
2. 変換後の文字列を戻り値として受け取って、
   コンソール上に出力。

```
//「12」という文字列が出力される
System.out.println(12);
```

toString()メソッド

変換された文字列の形式は、渡された値の型に応じて変わる。

文字型や数値型の場合：
元の値と同形式の内容をString型の文字列に変換。

toString()メソッド

参照型（クラス型）の場合：

「クラス名＠数値」といった形式の文字列（ハッシュコード）に変換。

```
// オブジェクトを生成
Phone phone = new Phone();
// ハッシュコードが出力される
System.out.println(phone);
```

```
Phone@1ab633e57
```

toString()メソッド

ハッシュコードの「@」より左側：
　　参照先のオブジェクトを生成したクラス
「@」より右側：
　　オブジェクトの識別番号（ハッシュ値）
　　ハッシュ値は16進数の数値形式で出力される。

```
Phone@1ab633e57
```

【Sample2007 toString()メソッドのオーバーライド】を作成しましょう

toString()メソッドはオーバーライドすることで、
柔軟に出力形式を変更できる。

Sample2007のポイント

Phone2007クラスでは、toString()メソッドを
オーバーライドして、料金とデータ通信量を含む
文字列を戻り値として返している。

```
public String toString() {
        String str = "料金:" + fee + "データ通信量:" + data;
        return str;
    }
```

Sample2007のポイント

Sample2007クラスでは、Phone2007クラスのオブジェクトを参照している変数をprintln()メソッドの引数に渡している。

```
Phone2007 p = new Phone2007();
System.out.println(p);
```

結果、オーバーライドしたtoString()メソッドで指定された形式の文字列が出力される。

```
携帯電話を購入しました。
料金は5000円でデータ通信量は7.3GBです。
料金:5000データ通信量:7.3
```

Sample2007のポイント

toString()メソッドをオーバーライドすれば、
ハッシュコードではなく、任意の形式の文字列を
出力させられる。

toString()メソッドにより、様々な型の値をString型の文字列に変換できます。

equals()メソッド

2つの変数が同じオブジェクトを参照しているかを判定する。
同じオブジェクトを参照している場合trueを、
そうでない場合にはfalseを戻り値として返す。

【Sample2008 equals()メソッド】を作成しましょう

Sample2008のポイント

Sample2008クラスでは、携帯電話クラスのオブジェクトを2つ作成。

変数phone1とphone2：
　　異なるオブジェクトを参照している。
変数phone3：
　　phone1と同じオブジェクトを参照している。

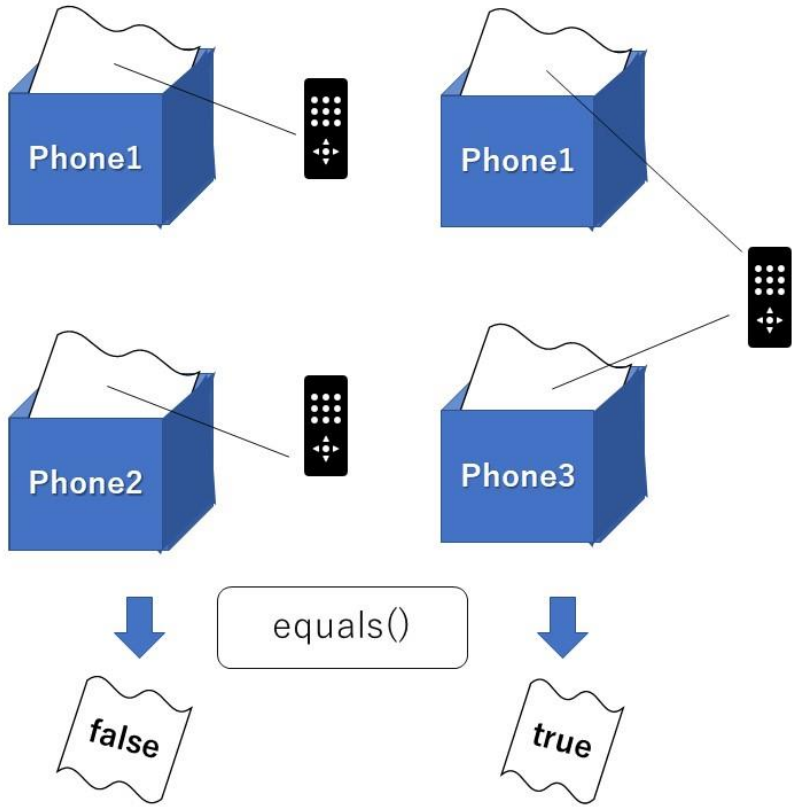
```
Phone2008 phone1 = new Phone2008();
Phone2008 phone2 = new Phone2008();
Phone2008 phone3 = phone1;
```

Sample2008のポイント

phone1からequals()メソッドを呼び出し。
引数としてphone2を指定した場合：
　　phone1とは異なるオブジェクトを参照しているため、false。
引数としてphone3を指定した場合：
　　phone1と同じオブジェクトを参照しているため、true。

```
phone1とphone2が同じオブジェクトか調べた結果:false
phone1とphone3が同じオブジェクトか調べた結果:true
```

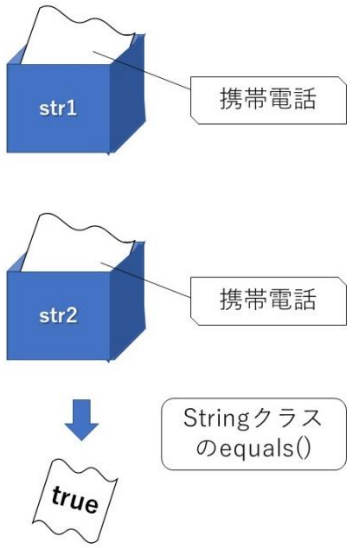
Sample2008のポイント



equals()メソッドは、2つの変数が参照しているオブジェクトが同じ場合にtrueを、異なる場合はfalseを戻り値として返します。

Stringクラスのequals()メソッド

一部のクラスは独自の処理を実行できるように
equals()メソッドをオーバーライドしている。
Stringクラスでオーバーライドされたequals()メソッドは、
2つのオブジェクトに保存された文字列が同じ内容であるかを
判定する。



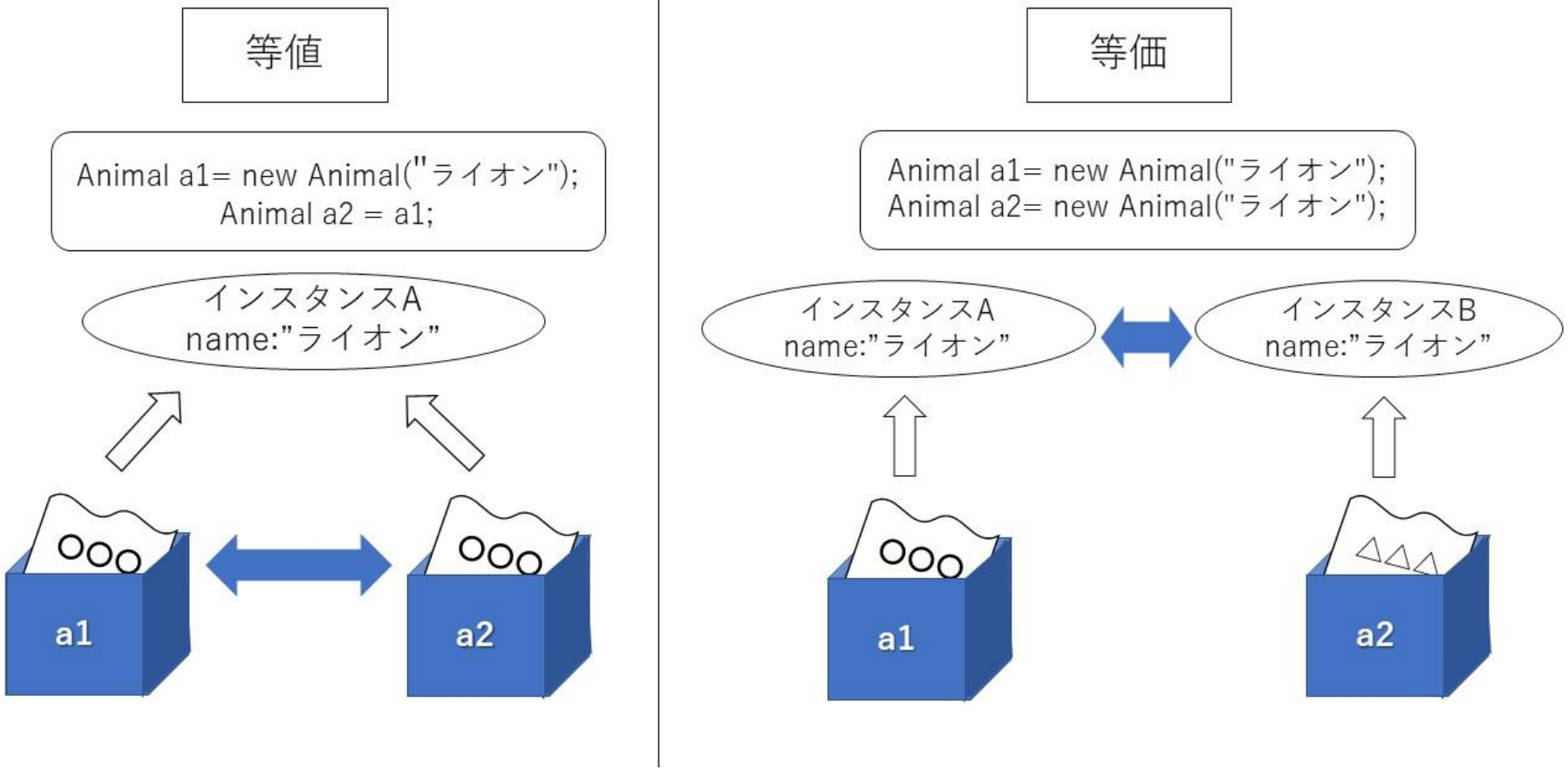
等値と等価の違い

「==」を使った判定と、この章で扱った「equals()メソッド」を使った判定では、意味が異なる。

== ： 等値（同一のオブジェクトを参照しているか）を判定。

equals()メソッド：
　　等価（変数に保存された値が等しいか）を判定。

等値と等価の違い



章のまとめ

- スーパークラスからサブクラスを拡張できます。
- サブクラス内で、スーパークラスが定義しているメソッドと同じ名前で再定義することを、オーバーライドといいます。
- super()を記述すると、スーパークラスのどのコンストラクタを呼び出すかを自分で指定できます。
- サブクラスからスーパークラスの同じ名前のメンバにアクセスする場合は、super.をつけます。
- 全てのクラスは、Objectクラスを継承しています。