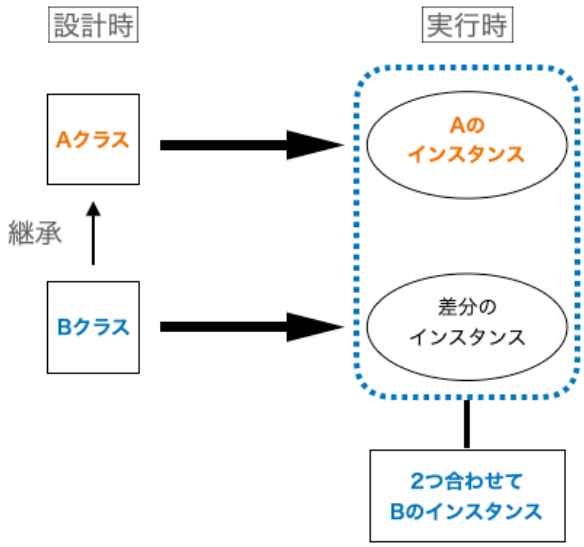
7-8.クラスの継承

継承のイメージ

継承は、あるクラスの内容を読み込み、別のクラスでも使用できるようにする機能です。
読み込み先のクラスを **スーパークラス** 、拡張したクラスを **サブクラス** などと呼びます。



Aクラスを継承するBクラスがあるとします。

Aをインスタンス化した時は、もちろんAの内容だけのインスタンスが生成されます。
Bをインスタンス化した時を考えると、Aの内容とBの内容を持つインスタンスとなります。
この時Bの内容は、単純にAとは別の追加要素もありますし、Aのメソッドをオーバーライドした修正要素などもあるはずです。なのでAとの **差分要素** とも呼びます。

継承されない要素

次の2つの要素は、継承しても引き継がれません。

- privateなフィールドやメソッド
- コンストラクタ

アクセス修飾子についておさらいしておきましょう。

アクセス修飾子	**概要**
public	すべてのクラスからアクセスできる
protected	現在のクラスとサブクラスからアクセスできる
なし(デフォルト)	現在のクラスと同じパッケージのクラスからアクセスできる
private	現在のクラスからだけアクセスできる

アクセス修飾子の中で`private`は、それが書いてあるクラスからしかアクセスできないという、かなり限定的な指定を行います。
サブクラスでも読み込めないので注意しましょう。 ?

コンストラクタも、何もしなければ読み込まれないので注意が必要です。

```
class A {
    A(String val) {
        // any code
    }
}

class B extends A { }

public class Main {
    public static void main(String[] args) {
        B b = new B("hello"); // コンパイルエラー
    }
}
```

上記コードではBのインスタンスを作り、Aにある引数が1つのコンストラクタを実行しようとしていますが、BにはAのコンストラクタが引き継がれていないため、コンパイルエラーとなります。

暗黙的に読み込まれるsuper();

次のようなコードの出力結果を考えてみて下さい。

```
class A{
    A(){
      System.out.println("Aクラス");
    }
  }

  class B extends A{
    B(){
      System.out.println("Bクラス");
    }
  }

  class C extends B{
    C(){
      System.out.println("Cクラス");
    }
  }

public class Main {
    public static void main(String[] args) {
        C c = new C();
    }
}
```

エントリーポイントのmainメソッドで、Cクラスのインスタンス化をし、コンストラクタを起動しています。

こちらの実行結果は以下となります。

```
//出力結果
クラスA
クラスB
クラスC
```

一見するとあれ？と思うかもしれません。
CクラスからBクラスのコンストラクタにアクセスする記述も無いし・・・

先程、スーパークラスのコンストラクタは継承されないといいましたが、そのコンストラクタを呼び出しをすることは必須になります。
また、呼び出しが明示されていない場合は、暗黙的にコンパイル時に自動で`super()`が呼ばれる仕様となっています。
なので、上記コードは下記コードと同じ意味となります。

```
class A{
    A(){
      super();
      System.out.println("Aクラス");
    }
  }
//※Aクラスは明示的に他のクラスを継承はしていませんが、クラスは基本的に「java.lang.Object」のサブクラスとなっています。
//その為、全てのクラスはjava.lang.Objectのコンストラクタが必ず呼ばれています。

  class B extends A{
    B(){
      super();
      System.out.println("Bクラス");
    }
  }

  class C extends B{
    C(){
      super();
      System.out.println("Cクラス");
    }
  }

public class Main {
    public static void main(String[] args) {
        C c = new C();
    }
}
```

`super()`は、各コンストラクタの一番初めの行に呼び出されますので注意しましょう。
また、`super()`は、スーパークラスにある、引数無しのコンストラクタを呼び出すコードですね。
よって、スーパークラスにコンストラクタがあっても、引数無しのものが無ければコンパイルエラーとなります。

引数があるコンストラクタを呼び出す時には、自分で明示的に記述する必要があります。

```
class A{
    A(String val2){
      System.out.println(val2);
    }
  }

  class B extends A{
    B(String val){
        super("Aを呼び出す");
      System.out.println(val);
    }
  }

  class C extends B{
    C(){
        super("Bを呼び出す");
      System.out.println("Cクラス");
    }
  }

public class Main {
    public static void main(String[] args) {
        C c = new C();
    }
}

//出力結果
Aを呼び出す
Bを呼び出す
Cクラス
```

継承時のフィールド・メソッド参照のルール

次のようなコードを考えます。出力結果はどうなるでしょうか。

```
class A {
    String val = "A";

    void print() {
        System.out.println(val);
    }
}

class B extends A {
    String val = "B";

    void print() {
        System.out.println(val);
    }

}

public class Main {
    public static void main(String[] args) {
        A a = new A(); //Aクラス型の変数aに、Aクラスのインスタンスを格納
        A b = new B(); //Aクラス型の変数bに、Bクラスのインスタンスを格納
        B b2 = new B(); //Bクラス型の変数b2に、Bクラスのインスタンスを格納

        System.out.println(a.val);
        System.out.println(b.val);
        System.out.println(b2.val);
        a.print();
        b.print();
        b2.print();
    }
}
```

AクラスとBクラスにはどちらも同じ名前のフィールドが存在しますが、中身が違います。

mainメソッドにて、AクラスとBクラスのインスタンス化をしていますが、 **それを格納している変数の型に注意しましょう。**

クラスの継承をした状態でどちらのフィールドが使われるかは次のようなルールがあります。

- フィールドを参照した場合は、 **変数のクラス型で宣言された方を使う**
- メソッドを呼び出した場合は、メソッド内の指示に従う

`a.val`は分かりやすいです。
Aクラスのインスタンスは、Bクラスの内容は関係ないので、`A`が出力されます。

`b.val`ですが、 **変数bはAクラス型で宣言されています。**
なので、AにもBにもフィールド`val`はありますが、Aの方が使われるので`A`が出力されます。

`b2.val`ですが、 **変数b2はBクラス型で宣言されています。**
なので、AにもBにもフィールド`val`はありますが、Bの方が使われるので`B`が出力されます。

`a.print();`は、Bクラスは関係無く、Aクラスにあるprintメソッドの実行、`val`の出力を行うので、`A`が出力されます。

`b.print();`は、Bクラスのインスタンスの中に、Aクラスにある`print`メソッドと、Bクラスでオーバーライドしている`print`メソッドを見つけます。
オーバーライドされているメソッドが優先的に実行されるので、 **Bクラスにある`print`メソッドが実行されます。**
Bクラスの中で、フィールドvalを読み込むと、そのクラス内にあるフィールドが読み込まれるので`B`が出力されます。

`b2.print();`は`b.print();`と全く同じ処理の流れになり、`B`が出力されます。

```
//出力結果
A
A
B
A
B
B
```

インスタンスそのものの中身と、それが入っている変数のクラス型は全く別物ですので、上記の少しややこしいルールを把握しておきましょう！

クラス型による扱いの違い

先程のコードにおいて、Aクラスの`print`メソッドを削除すると、Aクラス型の変数に入れたインスタンスでは`print`メソッドが実行できなくなります。

```
class A {
    String val = "A";
    // printメソッド削除
}

class B extends A {
    String val = "B";

    void print() {
        System.out.println(val);
    }

}

public class Main {
    public static void main(String[] args) {
        A a = new A(); // Aクラス型の変数aに、Aクラスのインスタンスを格納
        A b = new B(); // Aクラス型の変数bに、Bクラスのインスタンスを格納
        B b2 = new B(); // Bクラス型の変数b2に、Bクラスのインスタンスを格納

        System.out.println(a.val);
        System.out.println(b.val);
        System.out.println(b2.val);
        a.print(); // コンパイルエラー
        b.print(); // コンパイルエラー
        b2.print(); // 問題無し
    }
}
```

`a.print();`はAクラスのインスタンスが入っており、その中にメソッドは無いので実行不能なことは分かりやすいです。

`b.print();`ですが、変数`b`に入っているのは **Bクラスのインスタンスです。**
ということは、Bクラスの内容と、継承しているAクラスの内容を持っており、`print`メソッドが含まれているはずですね。
しかし、それを使用する(見つける)ことは出来ません。

ここで注意すべきなのは、扱う対象(インスタンス)そのものの内容と、型で指定する「扱う種類」は異なる概念だということです。

インスタンスのメソッドを指定した時、プログラムは変数の **クラス型に沿って** 、そのクラスの中身をまず検索します。
その時、そのクラスの中はそのメソッドが定義されている必要があります。
一つ前のコードのAクラスには、printメソッドが定義されていたため、それを実行することができました。
しかし、今回Aクラスには見当たらないため、実行できない状態になっています。

Aクラスに直接メソッドが定義されていなくても、Aクラスから継承したスーパークラスに定義があれば、実行することは可能です。

```
class AA {
    void print() {
        System.out.println("AAクラスのメソッド");
    }
}

//AAクラスを継承したAクラス
class A extends AA {
    String val = "A";
    // printメソッド削除
}

//Mainクラスの  a.print();  b.print();  はどちらも実行可
```

インスタンスの型変換

次のようなコードを考えます。

```
class A  {
    //any code
}

class B extends A {
    //any code
}

public class Main {
    public static void main(String[] args) {
        B b1 = new B();
        A a1 =b1; //問題無し

        A a2 = new B();
        B b2 = a2; //コンパイルエラー

    }
}
```

Aクラスを継承したBクラスがあります。

パターン①
Bクラス型の変数b1に、Bクラスのインスタンスを格納します。
これをAクラス型の別の変数a1に代入し、Aクラス型として扱うように変更することは可能です。

パターン②
次に、Aクラス型の変数a2に、Bクラスのインスタンスを格納します。
これをBクラス型の別の変数b2に代入することはできず、コンパイルエラーとなります。

クラス型のインスタンスを別のクラス型に変更することを **型変換** といいますが、型変換を行う時、値を扱うクラスを確認し、それらに互換性があるかをチェックします。

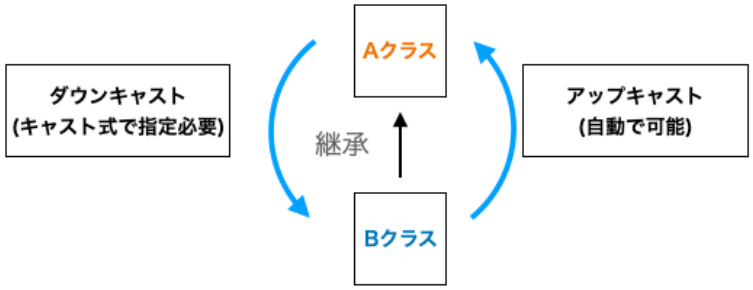
パターン①では、値はBクラス型で扱っていますので、Bクラスをチェックします。
その時、`extends A`という記述を見つけるので、Aクラスと互換性がある、と判断できます。

パターン②では、値はAクラス型で扱っていますので、Aクラスをチェックします。
この時、Aクラスだけを見ても、Bクラスとの関係性がどこにも見当たりません。なので互換性があるとは判断できず、コンパイルエラーとなります。
クラスの定義は何を継承しているかは記述しますが、そのクラスがどこのクラスに継承されているかは特に分からない形になっていますね。(これが1つの弱点でもあります。)

しかし、本来はBクラスのインスタンス型をBクラス型の変数に入れても問題無いはずですね。
このような動作を実現したい時、コンパイラに「この型に変換しても大丈夫」と宣言するために **キャスト式** を記述する方法があります。
`(変換する型名) 変数などの値`　のように記述します。

```
class A  {
    //any code
}

class B extends A {
    //any code
}

public class Main {
    public static void main(String[] args) {
        B b1 = new B();
        A a1 =b1; //問題無し

        //キャスト式で型変換
        A a2 = new B();
        B b2 = (B) a2; //問題無し

    }
}
```

サブクラスをスーパークラス型に変換することを **アップキャスト** 、その逆を **ダウンキャスト** といいますが、
アップキャストは自動で可能、ダウンキャストは指定が必要だということを押さえておきましょう。



課題

満点を取れるまで小テストを受験して下さい。

制限時間: 30 分

評定方法: 最高評点

合格点: 100 / 100

受験件数: 77