■　コンストラクタ

●コンストラクタ

クラスの中には、フィールドとメソッドのほかにコンストラクタを記述することが出来る。

コンストラクタは、**オブジェクトを生成**したときに自動的に呼び出され、主に**初期化処理用**に使われる。

クラス名(引数リスト)｛ //　コンストラクタは、メソッドと同様に引数を渡すことができる

コンストラクタの処理

：

｝

＊コンストラクタ名は**クラス名と同じ**にする必要がある（大文字小文字も含めて）

＊コンストラクタは、**戻り値を指定することはできない**。

●コンストラクタの定義を省略すると？

クラス内にコンストラクタをひとつも定義しなかった場合には、**引数のない空の**コンストラクタが用意され、自動的に呼び出されることになっている。そのようなコンストラクタを、「**デフォルトコンストラクタ**」という。

※自分でコンストラクタを定義した場合には、デフォルトコンストラクタが用意されないので注意

■　オーバーロード

●メソッドのオーバーロードとは

**同じクラスの中に、同じ名前を持つメソッド**（コンストラクタも含む）を2つ以上定義すること。ただし、メソッドをオーバーロードするときには、各メソッドの**引数の型、個数が異なるように**しなければならない。

※戻り値だけが異なる同名のメソッドはオーバーロードできない。

似たような複数の処理をオーバーロードしておくことにより、そのクラスを利用する場合に、1つのメソッド名を覚えて使うだけで、指定した引数の型、個数に応じた処理が自動的に呼び出されるようになる。

●同じクラスの別のコンストラクタを呼び出す

あるコンストラクタ内で、別のコンストラクタを呼び出したい場合、自分自身のオブジェクトを表す**this**キーワードを使用する。また、コンストラクタ（メソッド）内で、フィールドと同じ名前のローカル変数を使用したい場合、フィールドとローカル変数を区別するために、フィールドにも**this**キーワードをつける。

**<<参考　継承とコンストラクタ>>**

●スーパークラスのコンストラクタを指定する

スーパークラスに引数を持つコンストラクタがある場合、呼び出されるコンストラクタを明示的に指定したい時がある。このとき、サブクラスのコンストラクタの**先頭で、super**というキーワードを使って呼び出すことが出来る。

※スーパークラスの引数のないコンストラクタは、super()で呼び出すこともできるが、**省略しても自動的**に呼び出される。

※super又は　thisの呼び出しは、どちらかをサブクラスのコンストラクタ内の**先頭行に記述しないとエラーになる**。

●サブクラスのコンストラクタ内では

サブクラスのコンストラクタ内で特に何も指定されていない限り、サブクラスのコンストラクタ内の先頭で、スーパークラスの引数のないコンストラクタが呼び出されることになっている。（**コンストラクタは継承されないため**）

**<<参考　継承を使わずにあるクラスの機能を追加する　～委譲（delegation）～>>**

あるクラスの実装を、継承を使わずに他のクラスに任せること。

特定のクラスの機能を、自分が作るクラスにも持たせたい場合に、継承を使わずフィールドとしてそのクラスを持ち、そのクラスのメソッドを呼び出すメソッドを持たせること。つまり、自分が持つべき機能を他のクラスに「委譲」する。

コンストラクタは**継承されない**

■アクセス修飾子

Javaでは、カプセル化（データ隠蔽）などを行うために、以下のようなアクセス修飾子を使ってアクセスレベルを指定する。

|  |  |
| --- | --- |
| キーワード | 説　明 |
| **public** | どこからでも呼び出せる。 |
| **protected** | 同じパッケージか、そのサブクラスからしか呼び出せない。 |
| **なし** | 同じパッケージ内からしか呼び出せない。 |
| **private** | 同じクラスの中からしか呼び出せない。 |

■オーバーライド

●メソッドのオーバーライドとは

あるクラスが、別のクラスと**継承関係**にあるとき、スーパークラスとまったく同じメソッド名、引数（同じ**シグネチャ**）を持つメソッドを定義することが出来る。これをメソッドの　オーバーライドという。

※**戻り値のデータ型も同じ**でなければならない。

※アクセスレベルは、オーバーライドするほうが**より広くなければならない**。

■　抽象クラス

●抽象クラスとは

処理内容が定義されていないメソッド（**抽象メソッド**）を持つクラスである。

抽象クラスの**オブジェクトを作成（インスタンス化）**することはできない。

●抽象クラスを利用する

・抽象クラスを利用するには、抽象クラスを**継承**したサブクラスを作成する。

・そのサブクラス内で抽象クラスの抽象メソッドを**オーバーライド**して、メソッドの処理を具象化する。

■インターフェース

・具体的な処理が書かれていないメソッドの型だけを宣言している特別なクラスのようなもの。

・オブジェクトを作成すること（インスタンス化）は不可。

＜インターフェースの宣言＞

public **interface** インターフェース名 {

型名　定数フィールド名　＝　値;

戻り値の型　メソッド名(引数リスト);

}

* + - 通常、インターフェースのメンバには、何の修飾子もつけないが、何もつけなくても、フィールドには、**public　static　final（定数）**、メソッドには、**public abstract（抽象メソッド）**という修飾子をつけているのと同じことになる。

●インターフェースを実装する。

インターフェースは、抽象メソッドの中身を記述し、implementsキーワードを使ってクラスを定義する。

これを**実装する（implementation）**という。

* ポリモーフィズム

ポリモーフィズムとは

プログラミング言語の持つ性質の一つで、ある関数やメソッドなどが、引数や返り値の数やデータ型などの異なる複数の実装を持ち、呼び出し時に使い分けるようにできること。

--IT用語辞典--

●参照型変数の暗黙的型変換

**スーパークラス**の変数で**サブクラスのオブジェクト**を扱うことができる。（参照型変数の暗黙的型変換）

上記の例では、サブクラス（Dog、Cat）のオブジェクトは、スーパークラス（Animal）の変数で扱うことが出来る。

●動的束縛とは

最終的に呼び出されるメソッドが呼び出し先の**オブジェクトの種類**により**実行時に動的**に決まること。（dynamic binding）

●ポリモーフィズムとは。

同じ**メッセージ（メソッド）**　に対して**異なるオブジェクトが異なる動作**をすること．

実際の動作は，それぞれのオブジェクトによって異なるが，外部からは同じ**インターフェイス（クラス）**で扱えるようになる．

「**一つのインターフェイス，複数の実装**」と表現される

◆静的フィールドと静的メソッド

●静的フィールド（クラス変数）

クラス変数を宣言するためには、変数の前に「**static**」を付ける。こうすることにより、staticがついたフィールドは、クラスのすべてのインスタンスに共有され、その実体が1つだけのものとなる。宣言の仕方から、クラス変数は「static（静的）フィールド」とも呼ばれる。

・アクセス方法

クラス変数は、クラスで共有されるものなので、「**クラス名.フィールド名**」という指定方法でアクセスができる。（インスタンス化する必要はない）

＜例＞

　　Math.PI　　//（円周率）など

●静的フィールド（クラス変数）の注意点

static変数にアクセスする場合、**this**　キーワードをつけることはできない。

●静的メソッド

クラス変数と同じように、クラス共有のメソッドというものも使いたい場面がある。ここでも、キーワードstaticを使う。staticを付けて宣言されたメソッドは、「クラスメソッド」もしくは「staticメソッド」と呼ばれ、「**クラス名.メソッド名**」でほかのクラスから呼び出すことができる。インスタンスを使って呼び出す通常のメソッドは、「インスタンスメソッド」と呼ぶ。

　　＜例＞

Math.pow(2,3);　　//２の３乗の計算

●静的メソッドの注意点

クラスメソッド（例えばmainメソッド）内からは、　**クラス変数しかアクセスできない**。（クラスメソッドはインスタンス化されなくても実行される場合があるため）

<<参考　SOLID原則>>

オブジェクト指向プログラミングの分野において、SOLID（ソリッド）とは、ソフトウェア設計の5つの原則を記憶するための頭字語である。これらの原則は、ソフトウェアをより理解しやすく、より柔軟に、よりメンテナナンス性の高いものにするために考案されたものである。これらの原則は、アメリカのソフトウェアエンジニアでインストラクタでもあるRobert C. Martin（英語版）により提唱された多数のソフトウェア設計の原則の一部を集めたものとなっている

**単一責任の原則**（Single responsibility principle）

1つのクラスは1つだけの責任を持たなければならない。すなわち、ソフトウェアの仕様の一部分を変更したときには、それにより影響を受ける仕様は、そのクラスの仕様でなければならない。

**開放閉鎖の原則**（Open–closed principle）

「ソフトウェアのエンティティは（中略）拡張に対して開かれていなければならないが、変更に対しては閉じていなければならない。」

**リスコフの置換原則**（Liskov substitution principle）

「プログラムの中にある任意のオブジェクトは、プログラムの正しさを変化させることなく、そのサブクラスのオブジェクトと置換できなければならない。」詳しくは契約プログラミングも参照。

**インターフェイス分離の原則**（Interface segregation principle）（英語版）

「汎用的な目的のインターフェイスが1つだけあるよりも、特定のクライアント向けのインターフェイスが多数あった方がよりよい。」

**依存性逆転の原則**（Dependency inversion principle）

「具体ではなく、抽象に依存しなければならない」

ウィキペディア（Wikipedia）

ソフトウエア開発を行う場合、そのソフトウエアが完成して終わりということはありません。その後の維持管理が必ずついて回ります。その中で、機能追加や変更が発生した場合、いかに早く、ミスなくできるのかが重要です。そのため、設計段階からリリース後の追加変更を見越して設計していくことが大切です。

基本情報の勉強の中でもモジュール結合度、モジュール強度などでよい設計とはどのようなものかを勉強したと思いますが、基本的な考え方は、それと同じで、いかに「疎」な結合にして独立性を高めるかということが非常に重要です。SOLID原則はその一つの指針となる考え方です。今後、自分たちで開発設計を行う上で、迷いが生じるときの参考になるものであると考えます。

さらに、「クリーンアーキテクチャ」の中でもSOLID原則が紹介されているので、興味のある人は、勉強してみてください。