

5.4.4 匿名クラスの利用



あるメソッドの中で1回しか使わない、つまり「その場で使い捨てる」クラスを作りたい場合のために匿名クラスを紹介しよう。

GUIに関連するようなアプリケーションなど、一部のプログラムを開発していると、次のようなニーズが生じることがあります。

- ・メソッド中で、独自のクラスを定義してそのインスタンスを使いたい。
- ・ただし、インスタンスの生成は今回1回限りでよく、二度と行わない。

「あるインスタンスを今すぐ1つ欲しい」というだけで、クラスを定義することや、その後に new してインスタンスを生み出すというめんどろな手続きはできるだけ省いてラクしたいものですが、このような場合に活躍するのが匿名クラス (anonymous class) です。

匿名クラスは普通のクラスやほかの2つのインナークラスと比較して、かなり異質な存在です。特に根本的に違うのは、次の点です。

! 匿名クラスの特異性
普通のクラスやメンバクラスは、class キーワードでまず「宣言」し、その後 new で「利用」しますが、匿名クラスは「宣言と利用」を同時に行います。



宣言すると同時にインスタンス化もしちゃうってことですか？

そのとおり。かなり違和感を感じるはずだから、覚悟してコード例を見てほしい。



リスト 5-13 匿名クラスの利用例

```
1 public class Main {
2     public static void main(String[] args) {
3         Pocket<Object> pocket = new Pocket<Object>();
4         System.out.println
5             ("使い捨てのインスタンスを作りpocketに入れます");
6         pocket.put(new Object() {
7             String innerField;
8             void innerMethod() { ... }
9         });
10    }
```

Main.java

メンバを2つ持つ匿名クラスを
宣言すると同時にインスタンス化



うわキモっ… なんだこれ…。

リスト 5-13 の色文字で示した部分が、匿名クラスの宣言兼利用部分です。innerField と innerMethod を持つクラスを宣言すると同時に、その場でインスタンス化しています。このように、メソッド呼び出しや代入式の途中でいきなりクラスの宣言兼利用部分が現れるのが匿名クラスの特徴です (図 5-9)。

```
pocket.put( new Object() {
    String innerField;
    void innerMethod() { ... }
});
```

メンバを2つ持つクラスをその場で定義

さらにその場ですぐに、インスタンスを生成


```
pocket.put( インスタンス );
```

図 5-9 匿名クラスの宣言兼利用



それにしても意味が全然わからないのが「new Object()」という部分です。Object クラスを new しているんですか？

リスト5-13の5行目から8行目の色文字の部分が匿名クラスの中身の宣言であることは推測できると思います。一方、5行目でなぜいきなりObjectクラスが登場したのか意味がわからないという方も多いでしょう。
匿名クラスの宣言兼利用は、次のような構文で行うことになっています。

 匿名クラスの宣言兼利用

```
new 匿名クラスの親クラス指定() {
    匿名クラスの内容(メンバ)定義
}
```

そもそも匿名クラスは、クラス宣言と同時にインスタンス化も行ってしまうため、匿名クラス自身の名前を指定する必要はありません。その代わりに、どのクラスを継承して匿名クラスを作るかということを指定する決まりになっています。



「new 匿名クラス名 extends 匿名クラスの親クラス()」の色文字部分が省略された形だと考えると理解しやすいかもしれない。

5.4.5 インナークラスのまとめ

3種のインナークラスには、それぞれ異なる宣言方法・機能・制約がありました。たいへん混乱しやすいので、表5-1にまとめておきましょう。

表5-1 インナークラスのまとめ

		インナークラス			
		メンバクラス		ローカルクラス	匿名クラス
		static	非static		
宣言場所		クラスブロック内		メソッドブロック内	文内
宣言方法		class クラス名 { ~ }			new 親クラス() { ~ }
利用方法	無関係クラスから	new Outer.Inner()	new o.Inner() *1	利用不可	利用不可
	外部クラスから	new Inner()	new o.Inner() *1	利用不可	利用不可
	取り囲むメソッドから	----	----	new Inner()	利用不可
	宣言したメソッド内	----	----	new Inner()	利用不可
修飾	宣言したその場で		----	new 親クラス() { ~ }	
	アクセシビリティ	public	○	×	×
		protected	○	×	×
		package private	○	×	×
		private	○	○	×
	その他	final	○	○	○(*4)
		abstract	○	○	○(*4)
		extends	○	×	×
		implements	○	×	×
		static	○	○	○
アクセス	外部クラスの	非staticメンバ	×	○	○
		staticメンバ	○	×	×
	取り囲むメソッドの	ローカル変数	----	○	○

- *1 外部クラスのインスタンスが変数oに格納されているとする
- *2 無指定の記述自体はなされるが、package privateという意味ではない
- *3 修飾はできないが、暗黙的にfinalがつけられる
- *4 extendsやimplementsキーワードは利用しない。また複数指定はできない

減りゆくインナークラス活躍の場

本節で紹介した3種類のインナークラスは、実際にはあまり広く活用されていません。これらを利用することによってコードは複雑化しますが、それに見合うメリットが得られる状況が決して多くないからです。

また、このような複雑な構文を利用せずとも済むように、Java言語自体が進化し続けています。たとえば、次の章で紹介するラムダ式という新しい文法を利用すれば、これまで匿名クラスを用いてきたようなコードをより素早くスマートに記述できるようになります。

リスト 5-13 の5行目から8行目の色文字の部分が匿名クラスの中身の宣言であることは推測できると思います。一方、5行目でなぜいきなり Object クラスが登場したのか意味がわからないという方も多いでしょう。

匿名クラスの宣言兼利用は、次のような構文で行うことになっています。



匿名クラスの宣言兼利用

```
new 匿名クラスの親クラス指定 () {
    匿名クラスの内容 (メンバ) 定義
}
```

そもそも匿名クラスは、クラス宣言と同時にインスタンス化も行ってしまうため、匿名クラス自身の名前を指定する必要はありません。その代わりに、どのクラスを継承して匿名クラスを作るかということを指定する決まりになっています。



「new 匿名クラス名 extends 匿名クラスの親クラス ()」の色文字部分が省略された形だと考えると理解しやすいかもしれないね。

5.4.5

インナークラスのまとめ

3種のインナークラスには、それぞれ異なる宣言方法・機能・制約がありました。たいへん混乱しやすいので、表 5-1 にまとめておきましょう。

表 5-1 インナークラスのまとめ

		インナークラス				
		メンバクラス		ローカルクラス	匿名クラス	
		static	非static			
宣言場所		クラスブロック内		メソッドブロック内	文内	
宣言方法		class クラス名 { ~ }			new 親クラス() { ~ }	
利用方法	無関係クラスから	new Outer.Inner()	new o.Inner() *1	利用不可	利用不可	
	外部クラスから	new Inner()	new o.Inner() *1	利用不可	利用不可	
	取り囲むメソッドから	---	---	利用不可	利用不可	
	宣言したメソッド内	---	---	new Inner()	利用不可	
	宣言したその場で	---	---	---	new 親クラス() { ~ }	
修飾	アクセ ス	public	○	×	×	
		protected	○	×	×	
		package private	○	× (*2)	× (*2)	
		private	○	×	×	
	その他	final	○	○	×	
		abstract	○	○	×	
		extends	○	○	○(*4)	
		implements	○	○	○(*4)	
アクセ ス	外部 クラスの	static	○	×	×	
		非staticメンバ	×	○	○	
	取り 囲む メソッドの	staticメンバ	○	○	○	
		ローカル変数	---	---	×	×
		final変数	---	---	○	○

*1 外部クラスのインスタンスが変数oに格納されているとする

*2 無指定の記述自体はなされるが、package privateという意味ではない

*3 修飾はできないが、暗黙的にfinalがつけられる

*4 extendsやimplementsキーワードは利用しない。また複数指定はできない

減りゆくインナークラス活躍の場

本節で紹介した3種類のインナークラスは、実際にはあまり広く活用されていません。これらを利用することによってコードは複雑化しますが、それに見合うメリットが得られる状況が決して多くないからです。

また、このような複雑な構文を利用せずとも済むように、Java 言語自体が進化し続けています。たとえば、次の章で紹介するラムダ式という新しい文法を利用すれば、これまで匿名クラスを用いてきたようなコードをより素早くスマートに記述できるようになります。

5.5 この章のまとめ

型安全

- ・ Java では、取り扱う変数に対して型による制約をかけることで、処理の安全性を向上させている。
- ・ 型を積極的に活用し、データをより厳密に扱うことで不具合を減らせる。

ジェネリクス

- ・ `<~>` 記法を用いることで、型をパラメータとしたジェネリクスを定義できる。
- ・ ジェネリクスを用いたクラスは、利用時に型パラメータを決定する。
- ・ 型パラメータなしのコレクションは特段の理由がない限り利用しない。

列挙型

- ・ `enum` を用いることで、インスタンスを列挙した集合を定義できる。
- ・ 列挙型のインスタンスは、`switch` の分岐に利用できる。

インナークラス

- ・ クラスの内部に定義できるインナークラスには、「メンバクラス」「ローカルクラス」「匿名クラス」の3種類がある。
- ・ インナークラスの種類によって宣言方法、アクセス可能な変数などに違いがある。

5.6 練習問題

練習 5-1

以下の仕様に従った金庫を `StrongBox` クラスとして定義してください。

- ・ 金庫クラスに格納するインスタンスの型は、開発時には未定である。
- ・ 金庫には、1つのインスタンスを保存できる必要がある。
- ・ `put()` でインスタンスを保存し、`get()` でインスタンスを取得できる。
- ・ `get()` で取得する際、キャストを使わなくても格納前の型に代入できる。

練習 5-2

練習 5-1 で作成した `StrongBox` クラスに鍵の種類を示す列挙型 `KeyType` を定義した上で、以下の2つを `StrongBox` クラスの定義に加えてください。

- ・ 鍵の種類を示すフィールド。
- ・ 鍵の種類を受け取るコンストラクタ。

ただし、鍵の種類は以下の4種類に限定されるものとします。

①南京錠 (PADLOCK)	必要施行回数 = 1,024 回
②押ボタン (BUTTON)	必要施行回数 = 10,000 回
③ダイヤル (DIAL)	必要施行回数 = 30,000 回
④指紋認証 (FINGER)	必要施行回数 = 1,000,000 回

なお、金庫は `get()` メソッドが呼び出されるたびに回数をカウントし、各鍵が定める必要施行回数に到達しない限り `null` を返すようにしてください。

5.7 練習問題の解答

練習 5-1 の解答

```

1 public class StrongBox<E> {
2     private E item;
3     public void put(E i) {
4         this.item = i;
5     }
6     public E get() {
7         return this.item;
8     }
9 }

```

StrongBox.java

練習 5-2 の解答

```

1 enum KeyType { PADLOCK, BUTTON, DIAL, FINGER; }

```

KeyType.java

```

1 public class StrongBox<E> {
2     private KeyType keyType;
3     private E item;
4     private long count;
5     public StrongBox(KeyType key) {
6         this.keyType = key;
7     }
8     public void put(E i) {
9         this.item = i;
10    }
11    public E get() {

```

StrongBox.java

```

12    this.count++;
13    switch(this.keyType) {
14        case PADLOCK:
15        if (count < 1024) return null;
16        break;
17        case BUTTON:
18        if (count < 10000) return null;
19        break;
20        case DIAL:
21        if (count < 30000) return null;
22        break;
23        case FINGER:
24        if (count < 1000000) return null;
25        break;
26    }
27    this.count = 0;
28    return this.item;
29 }
30 }

```