1 Программирование объектов: введение

В этой главе

- ✓ Работа с объектами
- ✓ Модульное тестирование
- ✓ Динамические массивы

Прежде чем приступить к изучению стилей проектирования, давайте обсудим некоторые фундаментальные вопросы программирования с применением объектов. Мы кратко пройдемся по самым важным понятиям и поясним распространенную терминологию, с которой продолжим знакомиться и в последующих главах.

Мы рассмотрим:

- *Классы и объекты* — создание объектов на основе классов, использование конструктора, статические и объектные методы, статические методы фабрик для создания новых экземпляров и выдача исключений в конструкторе (раздел 1.1).
- *Состояние* — объявление свойств private и public, задание им значений, константы, а также изменяемые и неизменяемые состояния (раздел 1.2).
- *Поведение* — private- и public-методы, передача значений в качестве аргументов и исключения `NullPointerException` (раздел 1.3).

- *Зависимости* — их инициализация, обнаружение и передача в конструктор в качестве аргумента (раздел 1.4).
- *Наследование* — интерфейсы, абстрактные классы, переопределение и финальные классы (раздел 1.5).
- *Полиморфизм* — один интерфейс, разное поведение (раздел 1.6).
- *Композиция* — присвоение объектов свойствам и создание более сложных объектов (раздел 1.7).
- *Организация классов* (раздел 1.8).
- *Оператор возврата и исключения* — возвращение значения из метода, исключения в методе, обработка исключений, объявление собственных классов исключений (раздел 1.9).
- *Модульное тестирование* — паттерн Arrange-Act-Assert, тестирование на сбои и использование дублеров тестов для замещения зависимостей (раздел 1.10).
- *Динамические массивы* — их использование для создания списков или маппингов (раздел 1.11).

Если вы знакомы с этими темами, можете смело пропускать эту главу и переходить к главе 2. Если только некоторые темы представляют для вас интерес, просмотрите соответствующие разделы. Если же вы начинающий объектно-ориентированный программист, я рекомендую прочесть главу целиком.

1.1. КЛАССЫ И ОБЪЕКТЫ

Поведение объекта в среде исполнения задается объявлением класса. На основе класса можно создавать любое количество объектов. В следующем листинге показан простой класс, у которого нет состояния или поведения, но который можно инстанцировать.

Листинг 1.1. Минимально жизнеспособный класс

```
class Foo
{
    // Здесь ничего нет
}

object1 = new Foo();
object2 = new Foo();

object1 == object2 // false
```

Два экземпляра одного класса нельзя считать равными

После того как вы создадите экземпляр, можно будет вызывать его методы.

Листинг 1.2. Вызов метода экземпляра

```
class Foo
{
    public function someMethod(): void
    {
        // Что-нибудь сделать
    }
}

object1 = new Foo();
object1.someMethod();
```

Обычный метод, такой как `someMethod()`, можно вызвать только на *экземпляре* класса. Такой метод называется *объектным методом*. Можно также задать методы, вызываемые без наличия экземпляра. Они называются *статическими методами*.

Листинг 1.3. Объявление статического метода

```
class Foo
{
    public function anObjectMethod(): void
    {
        // ...
    }

    public static function aStaticMethod(): void
    {
        // ...
    }
}

object1 = new Foo();
object1.anObjectMethod();

Foo.aStaticMethod();
```

anObjectMethod() можно вызвать только на экземпляре SomeClass

aStaticMethod() можно вызвать без экземпляра

Кроме объектных и статических методов, класс может иметь специальный метод — *конструктор*. Этот метод вызывается до того, как объекту присваивается ссылка. Если необходимо подготовить объект к использованию, это можно сделать в конструкторе.

Листинг 1.4. Объявление метода-конструктора

```
class Foo
{
    public function __construct()
    {
        // Подготовить объект
    }
}

object1 = new Foo();
```

__construct будет явно вызван перед тем, как object1 будет присвоен экземпляр Foo

Можно помешать созданию объекта, вызвав *исключения* в конструкторе, как показано в листинге ниже. Подробнее об исключениях см. в разделе 1.9.

Листинг 1.5. Обработка исключения в конструкторе

```
class Foo
{
    public function __construct()
    {
        throw new RuntimeException();
    }
}

try {
    object1 = new Foo();
} catch (RuntimeException exception) {
    // `object1` будет неопределен
}
```

Инстанцировать Foo невозможно, так как его конструктор всегда будет выдавать исключение

Стандартный способ инстанцирования объекта класса — использование оператора `new`, как вы уже, наверно, заметили. Также можно задать статичный *фабричный метод* класса, который будет возвращать новый экземпляр этого класса.

Листинг 1.6. Объявление статичного фабричного метода

```
class Foo
{
    public static function create(): Foo
    {
        return new Foo();
    }
}

object1 = Foo.create();
object2 = Foo.create();
```

Метод `create()` нужно объявлять статичным, так как он вызывается на классе, а не на экземпляре класса.

1.2. СОСТОЯНИЕ

Объект может хранить данные. Эти данные могут содержаться в *свойствах*. Свойства будут иметь *имя* и *тип*; им можно присвоить значения в любой момент после создания объекта. Обычно свойства задаются в конструкторе.

Листинг 1.7. Объявление свойств и присвоение им значений

```
class Foo
{
    private int someNumber;
```

```
    private string someString;

    public function __construct()
    {
        this.someNumber = 10;
        this.someString = 'Hello, world!';
    }
}

object1 = Foo.create();
```

После инициализации свойствам someNumber и someString будут присвоены значения 10 и Hello, world! соответственно

Данные, содержащиеся в объектах, также называют *состоянием* объекта. Если эти данные необходимо жестко закрепить, как в предыдущем листинге, лучше присвоить значение изначально или объявить для него константу.

Листинг 1.8. Объявление констант

```
class Foo
{
    private const int someNumber = 10;
    private someString = 'Hello, world!';
}
```

В языке программирования может быть другой синтаксис. Например, в Java вы увидите final private int someNumber

С другой стороны, если значение свойства должно быть *переменным*, можно позволить клиенту передавать значение для аргумента конструктора. Добавляя параметр в конструктор, вы вынуждаете клиентов предоставлять значение при создании экземпляра класса.

Листинг 1.9. Добавление аргумента в конструктор

```
class Foo
{
    private int someNumber;

    public function __construct(int initialNumber)
    {
        this.someNumber = initialNumber;
    }
}

object1 = new Foo(); // не работает
object2 = new Foo(20);
```

Невозможно создать экземпляр Foo без предоставления значения для аргумента initialNumber

Должно сработать, здесь новому экземпляру класса Foo присваивается начальное значение 20

Если присвоить свойствам `someNumber` и `someString` модификатор `private`, они станут доступными только внутри экземпляров класса `Foo`. Это называется *инкапсуляцией*. Существуют и другие модификаторы для свойств: `protected` (см. раздел 1.5) и `public`. Присваивая свойству модификатор `public`, вы предоставляете любому клиенту доступ к нему.

Листинг 1.10. Объявление свойства с модификатором public

```
class Foo
{
    public const int someNumber;

    public string someString;

    // ...
}

object1 = new Foo();
number = object1.someNumber;
object2.someString = 'Cliché';
```

Так как свойство someNumber задано как константа, его значение менять нельзя, но хотя бы можно получить

someString не константа, но это свойство задано с модификатором public, поэтому здесь можно менять его значение

МОДИФИКАТОРОМ ПО УМОЛЧАНИЮ ДЛЯ СВОЙСТВ ДОЛЖЕН БЫТЬ PRIVATE

В общем случае предпочтительно использовать модификатор `private`. Ограничение доступа к данным объекта помогает скрывать детали его реализации. Так можно гарантировать, что клиентам не придется полагаться на специфичные данные объекта и они будут общаться с объектом через объявленные публичные методы (подробнее о методах см. в разделе 1.3). Мы разберем эту тему подробнее в следующих главах, например в разделах 6.3 и 9.8.

Инкапсуляция свойств (а также методов, см. раздел 1.3) основана на классах. Это означает, что если свойство имеет модификатор `private`, то к этому свойству имеют доступ все экземпляры класса, включая тот экземпляр, которому оно присвоено.

Листинг 1.11. Обращение к свойству private другого экземпляра

```
class Foo
{
    private int someNumber;

    // ...

    public function getSomeNumber(): int
    {
        return this.someNumber;
    }

    public function getSomeNumberFrom(Foo other): int
    {
        return other.someNumber;
    }
}

object1 = new Foo();
object2 = new Foo();

object2.getSomeNumberFrom(object1);
```

Foo, конечно же, имеет доступ к собственному свойству someNumber

Foo также имеет доступ к свойству someNumber других экземпляров Foo

Здесь вернется значение свойства someNumber

Объект называется *изменяемым*, когда значение свойства объекта может меняться в течение всего времени существования объекта. Если свойства объекта не могут меняться после инициализации, то объект считается *неизменяемым*. Листинг ниже иллюстрирует оба случая.

Листинг 1.12. Изменяемые и неизменяемые объекты

```
class Mutable
{
    private int someNumber;

    public function __construct(int initialNumber)
    {
        this.someNumber = initialNumber;
    }

    public function increase(): void
    {
        this.someNumber = this.someNumber + 1;
    }
}

class Immutable
{
    private int someNumber;

    public function __construct(int initialNumber)
    {
        this.someNumber = initialNumber;
    }

    public function increase(): Immutable
    {
        return new Immutable(someNumber + 1);
    }
}

object1 = new Mutable(10);
object1.increase();

object2 = new Immutable(10);
object2 = object2.increase();
```

Вызов increase() у Mutable изменит состояние объекта, модифицируя значение свойства someNumber

Вызов increase() у immutable не изменит состояния object2. Вместо этого мы получим новый экземпляр с увеличенным значением someNumber

В разделе 4.4 мы подробнее рассмотрим изменяемые объекты и узнаем, как делать их неизменяемыми.

1.3. ПОВЕДЕНИЕ

Кроме состояния объект обладает поведением, которым клиенты могут пользоваться. Это поведение объекта задается методами класса. Методы с модификатором

`public` доступны клиентам объекта. Их можно вызывать в любое время после создания самого объекта.

Некоторые методы возвращают сущности тому, кто его вызывает. В таком случае в качестве *возвращаемого типа* будет объявлен тип сущности. Некоторые методы ничего не возвращают. В таком случае возвращаемым типом будет `void`.

Листинг 1.13. Поведение объекта задается публичными методами

```
class Foo
{
    public function someMethod(): int
    {
        return /* ... */;
    }

    public function someOtherMethod(): void
    {
        // ...
    }
}

object1 = new Foo();
value = object1.someMethod();

object1.someOtherMethod();
```

someMethod() возвращает целое число, которое можно присвоить переменной

someOtherMethod() не возвращает ничего конкретного, поэтому клиент не сможет ничего присвоить переменной

Класс также может содержать объявления методов с модификатором `private`. Это работает так же, как и для приватных переменных. Любой экземпляр данного класса может вызывать приватные методы на любых других экземплярах одного и того же класса, включая себя самого. Тем не менее приватные методы часто используются для выполнения мелких шагов в большом процессе.

Листинг 1.14. Приватные методы

```
class Foo
{
    public function someMethod(): int
    {
        value = this.stepOne();

        return this.stepTwo(value);
    }

    private function stepOne(): int
    {
        // ...
    }

    private function stepTwo(int value): int
    {
        // ...
    }
}
```

ПРОВЕРЯЙТЕ НАЛИЧИЕ АРГУМЕНТОВ СО ЗНАЧЕНИЕМ NULL

В некоторых языках клиенты могут передавать значение `null` в качестве аргумента, даже если тип параметра был явно задан. Так что в примере листинга 1.15 аргумент для `bar` может оказаться `null`, даже если его тип явно указан — `Bar`. Попытка вызывать метод `doSomething()` на объекте `bar` приведет к выдаче исключения `NullPoinerException`. Поэтому всегда нужно проверять наличие аргументов со значением `null` либо обращать внимание на предупреждения компилятора или статического анализатора о потенциальных исключениях `NullPointerException`.

Вымышленный язык программирования, используемый в этой книге, *не позволяет* передавать `null` в качестве аргумента. Если мы хотим разрешить передачу `null`, это необходимо задать явно при помощи вопросительного знака (`?`) после объявления параметра метода. Таким же образом это работает и для типов свойств и возвращаемых типов:

```
class Foo
{
    private string? foo;

    private function someOtherMethod(Bar? bar): Baz?
    {
        // ...
    }
}
```

Так же как параметры конструкторов, можно задавать параметры методам. Клиенту придется предоставить специфичное значение в качестве аргумента при вызове метода. Сам метод может использовать значение для дальнейших действий: передать его вспомогательным объектам или использовать его для изменения значения свойства.

Листинг 1.15. Несколько способов применения аргументов метода

```
class Foo
{
    private int number;

    public function setNumber(int newNumber): void
    {
        this.number = newNumber;
    }

    private function multiply(int other): int
    {
        return this.number * other;
    }
```

Здесь аргумент newNumber станет новым значением свойства number

В этом случае аргумент other будет умножаться на текущее значение свойства number

```
    private function someOtherMethod(Bar bar): void
    {
        bar.doSomething();
    }
}
```

Здесь в качестве аргумента передается другой объект, и Foo даже может вызвать его метод

1.4. ЗАВИСИМОСТИ

Если объекту `Foo` нужен объект `Bar` для выполнения задачи, то `Bar` называют зависимостью `Foo`. Существуют три способа проверить, что у `Foo` есть доступ к зависимости `Bar`.

- `Foo` может самостоятельно инициализировать `Bar`.
- `Foo` может получить экземпляр `Bar` из известного источника.
- `Foo` может получить экземпляр `Bar` во время вызова конструктора.

В листинге ниже проиллюстрирован каждый из этих трех вариантов.

Листинг 1.16. Различные способы получения доступа Foo к экземпляру Logger

```
class Foo
{
    public function someMethod(): void
    {
        logger = new Logger();
        logger.debug('...');
    }
}
```

Foo инстанцирует Logger при необходимости

```
class Foo
{
    public function someMethod(): void
    {
        logger = ServiceLocator.getLogger();
        logger.debug('...');
    }
}
```

Foo получает Logger из известного источника

```
class Foo
{
    private Logger logger;

    public function __construct(Logger logger)
    {
        this.logger = logger;
    }

    public function someMethod(): void
    {
        this.logger.debug('...');
    }
}
```

Foo получает экземпляр Logger, предоставленный в качестве аргумента конструктора

Как обращаться с зависимостями, мы рассмотрим подробнее в разделе 2.2. Сейчас же достаточно знать, что получение зависимостей из известного источника называется *локализацией сервисов*, а получение зависимостей в качестве аргументов конструктора — *внедрением зависимостей*.

1.5. НАСЛЕДОВАНИЕ

Можно также задавать только часть класса и позволить другим классам его расширять. Например, у вас может быть класс без свойств и методов, только с сигнатурами методов. Такие классы обычно называют *интерфейсами*, и многие ООП-языки позволяют их таковыми объявлять. Класс затем использует интерфейс и предоставляет фактические реализации методов, определенных интерфейсом.

Листинг 1.17. Реализации Bar и Baz интерфейса Foo

```
interface Foo
{
    public function foo(): void;
}

class Bar implements Foo
{
}

class Baz implements Foo
{
    public foo(): void
    {
        // ...
    }
}
```

Интерфейс Foo объявляет метод foo(), но не предоставляет его реализацию

Класс Bar неверно реализует интерфейс Foo, так как в нем нет реализации метода foo()

Класс Baz правильно реализует интерфейс Foo, так как в нем представлена реализация метода foo()

В интерфейсах нельзя задать конкретную реализацию, но она может задаваться в *абстрактных классах*. Они позволяют предоставлять реализацию методов и сигнатуры методов. Абстрактный класс нельзя инстанцировать, но его можно расширить в другом классе, в котором будут реализовываться абстрактные методы.

Листинг 1.18. Baz расширяет абстрактный класс Foo

```
abstract class Foo
{
    abstract public function foo(): void;

    public function bar(): void
    {
        // ...
    }
}
```

Метод foo() абстрактный и должен быть реализован расширяющим классом

Класс Foo задает реализацию методу bar()

```
class Baz extends Foo
{
    public function foo(): void
    {
    }
}
```

Baz — корректная реализация класса Foo, так как в нем представлена реализация ранее объявленного метода foo()

И наконец, в классе могут быть заданы реализации всех его методов, а другие классы могут их расширять и переопределять.

Листинг 1.19. Класс Bar расширяет класс Foo и частично меняет его поведение

```
class Foo
{
    public function bar(): void
    {
        // сделать что-то
    }
}

class Bar extends Foo
{
    public function bar(): void
    {
        // сделать что-то еще
    }
}
```

Foo — обычный класс, без абстрактных методов

Bar расширяет Foo, который теперь является родительским классом. Bar может менять поведение его метода bar()

Foo — обычный класс, без абстрактных методов

Классы, которые расширяют другие классы, получают доступ к родительским методам с модификаторами доступа `public` и `protected`.

Листинг 1.20. Доступ к методам с модификаторами доступа public и protected

```
class Foo
{
    public function foo(): void
    {
        // сделать что-то
    }

    protected function bar(): void
    {
    }

    private function baz(): void
    {
    }
}

class Bar extends Foo
{
    public function someMethod(): void
```

```
    {
        $this->foo();   <-- foo() доступен, так как это метод с модификатором public
        $this->bar();   <-- bar() доступен, так как это метод с модификатором protected
        //$this->baz();   <-- baz() недоступен, так как это приватный метод
    }
}
```

Подклассы могут также переопределять родительские методы с модификаторами `public` и `protected`.

Листинг 1.21. Переопределение методов с модификаторами public и protected

```
class Foo
{
    public function foo(): void
    {
        // сделать что-то
    }

    protected function bar(): void
    {
    }

    private function baz(): void
    {
    }
}

class Bar extends Foo
{
    public function foo(): void   <-- Метод foo() можно переопределить, так как это метод с модификатором public
    {
        // ...
    }

    protected function bar(): void   <-- Метод bar() можно переопределить, так как это метод с модификатором protected
    {
        // ...
    }

    private function baz(): void   <-- Метод baz() нельзя переопределять, так как это приватный метод
    {
        // не работает
    }
}
```

В этой книге наследованию уделено не очень много внимания, несмотря на то что это крайне важный аспект ООП. На практике наследование зачастую приводит к усложнению дизайна. В книге мы в основном будем использовать наследование в двух случаях:

- при объявлении интерфейсов для зависимостей;
- при объявлении иерархии объектов, как, например, при объявлении кастомных исключений, которые расширяют встроенные классы исключений.

В большинстве случаев старайтесь не допускать расширений из классов. Для этого используйте ключевое слово `final` перед названием класса. Подробнее см. в разделе 9.7.

Листинг 1.22. Класс Bar нельзя расширить

```
final class Bar
{
    // ...
}

class Baz extends Bar // не работает
{
    // ...
}
```

Bar обозначен ключевым словом final, поэтому его запрещено расширять

1.6. ПОЛИМОРФИЗМ

Полиморфизм — это одна из основополагающих концепций ООП. Полиморфизм означает, что если для параметра задан определенный класс в качестве типа, то любой объект, являющийся экземпляром этого класса, можно передавать в качестве допустимого аргумента. Например, в следующем листинге любой экземпляр `Foo` можно передать в качестве аргумента методу `bar()`:

Листинг 1.23. Любой экземпляр Foo будет допустим для метода bar()

```
class Foo
{
    // ...
}

final class Bar
{
    public function bar(Foo foo): void
    {
        foo.someMethod();
    }
}
```

Так как экземпляры Foo могут иметь различную конфигурацию или внутреннее состояние, то теоретически каждый из них может вести себя по-своему. Это значит, что можно влиять на поведение `bar()`, не изменяя сам метод `bar()`.