Модуль 2. Объектно-ориентированное программирование

Тема 2.1. Классы и объекты

2 часа

Оглавление

2.1. Введение в ООП

2.1.1. Новый принцип создания программ

В связи со все возрастающим объемом программного кода и все увеличивающейся сложностью программ возникла необходимость в технологии, которая позволила бы упростить разработку.

Слишком сложно удерживать в памяти все детали реализации и те неявные связи, при помощи которых одни части программы влияют на другие.

В частности:

- разделить большую задачу на подзадачи, каждую из которых можно было бы выполнять отдельно от других, не опасаясь при этом, что изменение данных в одной подзадаче повлияет на данные другой подзадачи,
- легко использовать в других проектах решения подзадач, которые были реализованы ранее (и, возможно, другими разработчиками),
- перейти от понятия обработки структур данных к понятиям описания объектов — их свойств и действий над ними;
- использовать реализованные ранее объекты для реализации новых объектов, обладающих частичными свойствами старых.

Новая технология получила название - объектно-ориентированное программирование (ООП). В рамках второго Модуля мы изучим базовые принципы ООП и освоим его инструменты, которые дает нам язык Java, проиллюстрируем выше описанные преимущества.

2.1.2. Классы и объекты

Во всех ранее написанных программах мы встречались с ключевым словом class и и уже знаем, что класс - это базовое понятие ООП. Классическое определение говорит, что класс - это абстрактный тип данных. Т.е. так же, как изученные нами ранее int или double, класс является типом.

Но что же значит слово “абстрактный?” Это значит, что класс может описывать понятия на более высоком уровне абстракции. Например, с помощью типа int мы можем описать одно число целого типа, но при этом не можем сказать, что это будет за число: возраст человека, номер дома, страница книги. А ведь программы пишутся для использования в нашей повседневной жизни и существует огромный разрыв между таким примитивным понятием как число и сложными сущностями: “товар в интернет магазине”, “персональная страничка человека в социальной сети”, “счет в банке”, “песня на сайте” и т.д.

С точки зрения ООП каждую такую сущность можно описать, как совокупность

- полей класса - переменных для хранения данных, описывающих класс (те свойства/параметры/характеристики, которые описывают состояние сущности) и
- методов класса - функций для работы с полями класса. Это те действия, которые можно производить с этой сущностью.

Например, класс “Товар в интернет магазине” можно охарактеризовать, как совокупность

- полей: “Наименование товара”, “Цена“, “Производитель”, “Категория товара”, “Оценка

потребителя”, “Фото товара” и т.д.

- методов: “Сделать скидку”, “Задать Наименование товара”, “Поставить Оценку”, “Добавить Фото товара” и т.д.

В итоге, если описать такой класс, мы получаем возможность оперировать всем этим набором, как единое целое, а не отдельными примитивными типами!

Так же, как мы объявляем переменную примитивного типа, мы можем создать переменную класса - объект. **Объект** - это экземпляр класса. Можно привести аналогию: класс - это форма для выпечки печений, а объект - это сами печеньки.

Приведем примеры полей для описанного ранее класса “Товар в интернет магазине”:

- Молоко, 30 руб., Веселый молочник, молочные продукты, …
- Samsung Galaxy S5, 27000 руб., Samsung, телефоны, ...

В идеологии ООП разработчик начинает написание программы с проектирования классов. В процессе создания программы это важнейшая и действительно сложная задача, для решения которой предлагается множество подходов и методик, а также разработаны специальные инструменты для графической иллюстрации структуры классов и их связей между собой (например, диаграммы классов языка графического описания UML).

Три парадигмы, "три кита" объектно-ориентированного программирования это:

- инкапсуляция,
- наследование,
- полиморфизм.

В данной теме мы дадим общее описание этих парадигм. Позже каждая из них будет разобрана отдельной темой более детально.

2.1.3. Инкапсуляция

Инкапсуляция означает, что методы и поля класса, рассматриваются в качестве единого целого. В связи с чем инкапсуляция требует соблюдения следующих принципов:

- Все характеристики, которые описывают состояние объекта, должны храниться в его полях.
- Все действия, которые можно осуществлять с объектом, должны описываться его методами.
- Нельзя извне менять поля объекта. Обратиться к полям объекта можно только при помощи методов. Из-за чего инкапсуляцию часто связывают с понятием “сокрытие данных”.

Такой подход позволяет, с одной стороны, обеспечить правильное функционирование внутренней структуры объекта. С другой стороны — при необходимости изменить режим внутренней работы объекта, не меняя способы его внешнего использования. Достаточно изменить внутреннюю структуру.

Например, объект реального мира — микроволновая печь. Для того, чтобы ею пользоваться,

нужно только складывать в нее блюда и кнопками задавать время работы. Пользователю не нужно знать, в какой момент внутренняя тарелка должна вращаться, какой ток и какое напряжение нужно подавать на элементы внутренней электроники.

Если пользователю позволить менять эти параметры напрямую, он может задать такую их комбинацию, после которой микроволновая печь взорвется или загорится.

С другой стороны, если изготовитель микроволновой печи пришлет мастера, который изменит принцип ее внутренней работы, пользователь этого не почувствует — в его распоряжении все также останется дверца и кнопки, задающие время разогрева.

Микроволновая печь в этом примере (как и инкапсулированный объект) — это "черный ящик" для пользователя. Таким образом ООП дает возможность пользователю пользоваться объектами, не задумываясь об их внутреннем устройстве.

2.1.4. Наследование

Наследование — механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Класс-потомок может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами. Позволяет строить иерархии классов.

Некоторые объекты являются частным случаем более общей категории объектов. Например, авторучка, фломастер и простой карандаш являются частными случаями категории "инструмент для письма". С одной стороны, каждый из объектов обладает отдельными характеристиками. С другой стороны, они имеют общие свойства (например, цвет). Хотелось бы иметь возможность рассматривать их как объект общей категории "инструмент для письма", т.к. для всех этих объектов применимы общие действия (например, “писáть”), но при этом уметь хранить для каждого в отдельности его особенные свойства и применять к ним присущие каждому специфические методы.

Например, для авторучки хотелось бы различать ее тип (шариковая, гелевая, роллер) и уметь менять стержень. Для фломастера — знать перманентный он или нет. А для карандаша — знать, простой он или цветной и уметь его точить.

Используя технологию наследования, говорят, что фломастер, авторучка и карандаш — наследники "инструмента для письма". Абстрактный (в данном случае) объект "инструмент для письма" хранит информацию о цвете и к нему может быть применена операция "писáть".

Еще принято говорить, что "инструмент для письма" является родительским по отношению к фломастерам, ручкам и карандашам.

Каждый из объектов — фломастер, авторучка и карандаш — наследуют от "инструмента для письма" свойство "цвет" и метод "писáть". И при этом для каждого из них мы добавляем свои дополнительные свойства и методы.

2.1.5. Полиморфизм

Полиморфизм - это реализация одинакового по смыслу действия различным способом в зависимости от типа объекта.

Например, в предыдущей теме мы рассмотрели операцию “Писáть”, общую для всех “Инструментов для письма". Теперь рассмотрим другое общее действие, присущее этим объектам - метод "Привести в рабочее состояние". Несмотря на то, что действие общее, каждый наследник реализует его по-своему.

На авторучке нужно щелкнуть кнопку в торце (если считать, что авторучка с убирающимся кончиком). С фломастера нужно снять колпачок (который предохраняет фломастер от высыхания). А с карандашом не нужно ничего делать, если он заточен. В этом заключается идея полиморфного поведения наследников по отношению к родительскому объекту.

При вызове метода "Привести в рабочее состояние" родительского класса “Инструмент для письма” для каждого из объектов компилятор (в нашем примере — человек) сам по классу объекта понимает, какой метод нужно использовать.

Задание 2.1.1.

Придумайте иерархию 3-4 объектов из реальной жизни и опишите инкапсулированные свойства и методы их родительского класса, свойства и методы наследуемых от них классов (отличающиеся от родителя) и полиморфные методы, применимые к ним.

Например, животные: кенгуру, слон, карась. Способ передвижения, способ получения и выращивания потомства.

2.1.6. Описание класса

Ключевое слово class, как начало описания нового типа данных

Научимся создавать собственные классы в языке Java. Прежде всего, нужно написать ключевое слово **`class`**. После этого слово нужно написать имя нового класса. Это имя должно быть создано по общим правилам имен идентификаторов (в т.ч. переменных) в Java.

Для соблюдения конвенции по оформлению кода Java первая буква имени класса должна быть заглавной. После имени класса в фигурных скобках задается описание класса.

Описание класса содержит:

- описание данных, которые хранятся в классе (поля/переменные);
- описание методов, которые обрабатывают эти данные.

Общая форма оформления класса:

```
class Имя_класса {
      тип_поля1 имя_поля1; // имя_переменной1
      тип_поля2 имя_поля2; // имя_переменной2
      тип_результата1 имя_метода1 (параметры_метода1) {
          тело_метода1
      }
```

```
        тип_результата2 имя_метода2 (параметры_метода2) {
            тело_метода2
        }
}
```

Заметим, что создание нового класса означает создание нового типа данных. Например, давайте создадим класс, который описывает прямоугольник:

```
class Rect {
        double width;
        double height;
}
```

Теперь можно создавать переменные (объекты) этого типа:

```
Имя_класса имя_объекта;
```

В нашем примере:

```
Rect rect; // объявили переменную класса Rect под именем rect
```

Заметим, что в этой строчке кода мы только объявили переменную, т.е. сообщили компиллятору, что в переменной rect мы собираемся хранить ссылку на объект - экземпляр класса Rect. Самого объекта в памяти еще нет.

Мы даже можем явно указать, что объектная переменная пока ни на что не ссылается, используя значение null:

```
rect = null;
```

Чтобы создать сам объект, необходимо использовать операцию **new** (мы уже встречались с этой операцией, когда создавали массивы):

```
rect = new Rect();
```

Можно объединить описание и создание объекта:

```
Имя_класса имя_объекта = new Имя_класса();
```

В нашем случае:

```
Rect rect = new Rect();
```

В примере после оператора new стоит Rect() - вызов метода класса без параметров и его имя совпадает с названием класса. Такой метод называется конструктором класса. Конструктор определяет действия, выполняемые при создании объекта.

Если в классе программистом не был задан конструктор, то Java автоматически создаст "конструктор по умолчанию", который проинициализирует все поля создаваемого объекта значениями по умолчанию, принятыми для каждого типа Java.

В соответствии с официальной документацией Java для переменных различных типов определены следующие значения по умолчанию:

Тип	*Значение по умолчанию*
byte, short, int, long, float, double	*0 (в соответствующем формате представления)*
char	'\u0000'
boolean	false
объекты и массивы	null

В следующей теме мы более подробно разберем, что такое конструкторы и как создавать собственные конструкторы.

Почему при создании переменных типа int или float мы не используем операцию new? Потому что в Java они относятся к примитивным типам данных, а у таких типов значения хранятся непосредственно в переменных. Это позволяет языку обеспечить высокую скорость работы.

В переменных объектного типа, о которых сейчас идет речь, хранятся ссылки на динамическую область памяти, где операцией new было выделено место под хранение объекта со всеми его полями и ссылками на методы.

Описание полей класса

Для обращения к полям (и методам) созданного объекта нужно использовать операцию "точка".

Например, для обращения к полю "width" объекта "rect" нужно написать rect.width.

Установим ширину и высоту прямоугольника rect и посчитаем его площадь:

```
rect.width = 10;
rect.height = 15;
System.out.println(rect.width * rect.height);
```

Заметим, что каждый объект имеет собственные значения полей. Данные одного объекта полностью отделены от данных другого объекта (если необходимо, то это можно поменять, используя слово `static`).

Рассмотрим фрагмент:

```
Rect rect1 = new Rect();
rect1.height = 10;
rect1.width = 20;

Rect rect2 = new Rect();
rect2.height = 10;
rect2.width = 20;

System.out.println(rect1 == rect2); // ?
rect2 = rect1;
System.out.println(rect1 == rect2); // ?
```

У каждого из объектов rect1 и rect2 мы можем задать свои собственные значения полей: ширина (width) и высота (height), при этом изменение параметров одного из прямоугольников никак не повлияет на другой.

Если запустить этот фрамент кода, то будет выведено:

```
false
true
```

Почему в первом случае, несмотря на то, что значения полей объектов rect1 и rect2 одинаковы, результат операции сравнения == отрицательный, а во втором случае положительный?

Потому что здесь операция “==” сравнивает не содержимое объектов, а ссылки на объекты в памяти. А как мы уже разбирали, оператор `new` возвращает ссылку на новую область динамической памяти, следовательно rect1 и rect2 сначала хранят ссылки на разные ячейки памяти.

Далее, после операции rect2 = rect1; переменной rect2 мы присвоили адрес объекта, на который ссылается rect1. Таким образом, обе переменные стали ссылаться на один и тот же объект и во втором случае результат сравнения - true.

Метод класса, его аргументы и возвращаемое значение. Описание метода в протоколе класса

Кроме полей, в классе можно также описать методы (действия), которые можно совершать с объектами класса. Синтаксис описания методов мы разбирали в теме 1.9:

```
тип_данных_которые_метод_вернет ИМЯ_МЕТОДА (список_параметров_метода){
	операторы тела метода
}
```

Например, добавим в определение нашего класса Rect метод area, который вычисляет площадь прямоугольника:

```
class Rect {
	double width;
	double height;

	double area() {
```

```
            return width * height;
        }
}
```

У данного метода нет параметров, потому что его параметрами являются ширина и высота самого объекта — прямоугольника.

Для вызова метода, как и для обращения к полю класса, нужно использовать операцию точка. Только для метода необходимо после имени обязательно написать круглые скобки, в которых указываются параметры его вызова.

Даже если методу не требуются параметры, пустые скобки все равно должны быть.

Например, в данном примере:

```
System.out.println(rect1.area());
```

Пример метода с параметром:

```
// Увеличение в указанное число раз размеров прямоугольника
void magnify(double koeff) {
        width *= koeff;
        height *= koeff;
}
```

Обращение к этому методу:

```
rect1.magnify(1.5);
```

Заметим, что для вызова метода, описанного в классе, нужно обязательно указать объект, к которому применяется этот метод (исключение — статические методы, которые будут рассмотрены позже).

Например, рассмотренный метод magnify не возможен без указания того, чьи именно ширину и высоту нужно изменить — ведь у каждого объекта они разные!

Вызов метода класса полезно рассматривать как посылку сообщения объекту. Указанный (перед точкой) объект рассматривается как адресат, которому посылают сообщение — приказ что-то выполнить. Если не указать конкретный объект — адресата нет и неясно, кому адресовать сообщение (метод).

Еще раз уточним различие между классом и объектом:

- Класс — это описание типа данных, некая абстрактная логическая конструкция.
- А объект — это нечто реально существующее. Он занимает место в памяти.

Упражнение 2.1.1

Спроектируйте и реализуйте простейший класс, описывающий рациональную дробь.

2.1.7. Обзор классов-оболочек примитивных типов

Мы уже хорошо знаем примитивные (базовые) типы данных языка Java, , не являющиеся классами. Эти типы данных не вписываются в общую "объектно-ориентированную картину мира" Java.

Для того, чтобы с примитивными типами данных можно было работать так же, как с остальными-объектными, для каждого из них существует свой собственный класс-оболочка. Они инкапсулируют в себе эти простые типы и предоставляют широкий набор методов для работы с ними.

Такой прием достаточно широко применяется. К примеру известная нам уже программа TestBed также содержит классы-оболочки, позволяющие скрыть особенности реализации кода под платформу Android.

Далее рассмотрим классы-оболочки, соответствующие 6 примитивным числовым типам данных. Все эти классы-оболочки являются наследниками абстрактного класса `Number`.

Примитивный тип	**Соответствующий класс-оболочка**
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Приведем некоторые примеры использования методов перечисленных классов-оболочек:

Примеры методов	**Класс-оболочка**	**Описание**

`public static Integer valueOf(int i)`
 	
`public static Double valueOf(double d)`
`...`
	все	Создание объектов соответствующих классов:
		
		`Integer i = Integer.valueOf(50);`
		`Double db = Double.valueOf(50.5);`
		`Long ln = Long.valueOf(50);`
		`Short sh = Short.valueOf((short)50);`
		`Byte bt = Byte.valueOf((byte)50);`

`public int intValue()`
 	
`public double doubleValue()`
`...`
	все	Для объекта возвращает соответствующее значение примитивного типа:
		`Integer i = Integer.valueOf(50);`
		`int d = i.intValue(); //50`

`public String toString()`	все	Результат - строковый десятичный эквивалент вызываемого объекта.

		Пример:
		`int x = 125;`
		`Integer y = x;`
		`String s1 = y.toString(); //"125"`
		`String s2 = Double.toString(2.5); //"2.5"`

`public static int parseInt(String s)`	`Integer`	Переводит строковое представление числа s в int
		`int value = Integer.parseInt("10");`
		`System.out.println(value); //10`

`public static int parseInt(String s, int radix)`	`Integer`	Переводит строковое представление числа s в системе счисления `radix`, в целое число:
		`int value = Integer.parseInt("110",2);`
		`System.out.println(value); //выведет 6 (110 в 2-чной системе счисления)`

`public static double parseDouble(String s)`	`Double`	Переводит строковое представление числа в вещественное:
		`double value1 = Double.parseDouble("256.01");`
		`System.out.println(value1); //256.01`

`public static String toBinaryString(int i)`	`Integer`	Переводит число из 10-чной системы в 2-чную и возвращает в виде строки :
		`int x = 12;`
		`System.out.println(Integer.toBinaryString(x)); //1100`

`public static String toOctalString(int i)`
 	
`public static String toHexString(int i)`
	`Integer`	Аналогичны предыдущему методу, но переводят в 8-чную и 16-чную системы соответственно:
		`int x = 12;`
		`System.out.println(Integer.toOctalString(x)); // 14`
		`System.out.println(Integer.toHexString(x)); // c`

Задание 2.1.2.

Введите с клавиатуры целое число X ($|X| \le 10^9$).

В первых трех строках выведите это число на экран в двоичной, восьмеричной, 16-ричной системах счисления.

В следующих двух строках выведите, поместится ли это число в ячейке типа byte и ячейке типа short ("YES"/"NO").

Запрещается использовать циклы и знания о том, сколько именно байт/бит памяти занимают переменные типа int, byte, short.

Пример ввода:

```
123
```

Пример вывода:

```
1111011
173
7B
YES
YES
```

Пример ввода:

```
40000
```

Пример вывода:

```
1001110001000000
116100
9C40
NO
NO
```

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Ушакову Денису Михайловичу.