Начинаем программировать на языке Java. Часть 2

Дмитрий Рамодин

Разобравшись с классами языка Java и с тем, как они описываются непосредственно в программе, это занятие мы посвятим частному случаю классов — абстрактным классам, а также элементам, родственным классам, называемым интерфейсами. Кроме того, вы познакомитесь с таким замечательным средством языка Java, как обработка исключительных ситуаций.

1. Абстрактные классы

Многие программисты, использующие язык Си++, уже знакомы с понятием абстрактных классов. В языке программирования Java абстрактные классы весьма схожи со своими "родственниками" из Си++ и отличаются лишь деталями реализации. Но тем не менее давайте рассмотрим абстрактные классы Java подробно.

Предположим, вы создаете некую модель бытовой техники и вам требуются отдельные классы для описания каждой детали этого устройства. Логично создать единую электронную форму для занесения в нее сведений о любом аппарате, в которой будет стоять ссылка на некий класс "Бытовое устройство". Теперь вы можете смело ссылаться из этой формы на любой класс, унаследованный от класса "Бытовое устройство". Это может быть "Пылесос", "Электрогриль" и т. п. При этом все ссылки будут работать корректно, несмотря на то, что они могут указывать на совершенно разные классы. Главное, чтобы у них был некий общий предок. Но вот что интересно. Когда мы начинаем пользоваться классами, мы создаем их экземпляры с помощью вызовов new. Остается неясным, зачем создавать экземпляр класса "Бытовое устройство", ведь товара с таким наименованием просто не существует! Какой покупатель станет платить за непонятный агрегат "Бытовое устройство"! Стало быть,

это всего лишь удобный способ задания общего класса-предка. А раз не требуется его реализация, то можно просто создать пустой класс, в котором будут описаны (но не реализованы!) некоторые общие методы для работы с данными внутри класса, например методы "Установить величину напряжения питания" или "Включить устройство". Полученный класс будет называться абстрактным. Все классы — потомки абстрактного класса унаследуют его данные и методы, но должны будут сами предоставить код тех методов, которые класс-предок оставил нереализованными, т. е. абстрактными. Резонно спросить, почему базовый абстрактный класс не реализует методы самостоятельно, а отдает это на откуп своим потомкам. Ответ прост: для каждого устройства потребуется своя собственная методика включения и установки напряжения питания. К примеру, пылесос можно включить нажатием на кнопку, а сушилка для рук запускается автоматически, когда под нее подставляют руки. Такие тонкости может знать только класс самого устройства, а значит, ему и отвечать за реализацию соответствующих методов.

Другой часто приводимый пример — программа рисования геометрических фигур. Есть некоторая программа, задача которой состоит в рисовании точки, круга и квадрата. На ее примере мы и рассмотрим, как создавать абстрактные классы Java. Для начала создадим абстрактный класс Shape — предок всех фигур. Для каждой фигуры потребуются одинаковые данные: цвет (Color) и начальная точка (StartPoint). Чтобы нарисовать фигуру, необходимо создать метод Draw. Как вы, наверное, уже поняли, метод Draw абстрактного класса Shape будет пустым. Непосредственной его реализацией займется класс каждой фигуры. Для объекта класса "Точка" (Point) нужно нарисовать точку, для объекта класса "Круг" (Circle) — круг, а для объекта класса "Квадрат" (Square) - квадрат.

На языке Java абстрактный класс Shape будет описан следующим образом:

```
  // Абстрактный класс "Фигура"
  abstract public class Shape
{
         // Цвет фигуры
        int Color;

        // Начальная точка фигуры
        Coordinates StartPoint;
```

```
        // Нарисовать фигуру
        abstract public void Draw();
}
```

Обратите внимание на модификатор abstract в описании класса Shape и его метода Draw. Этим модификатором необходимо отмечать все абстрактные методы и классы. На тип Coordinates не обращайте внимания. Он взят лишь для примера. Предполагается, что где-то ранее он был описан как тип для задания координат x и y фигуры.

Теперь унаследуем от класса Shape необходимые нам классы фигур.

```
   // Конкретный класс "Точка"
   class Point extends Shape
{
        // Цвет точки
        int Color;

        // Координаты точки
        Coordinates StartPoint;

        // Нарисовать точку
        public void Draw()
        {
        // Здесь рисуется точка
        }
}
   // Конкретный класс "Круг"
   class Circle extends Shape
{
        // Цвет круга
        int Color;

        // Координаты центра круга
        Coordinates StartPoint;

        // Нарисовать круг
        public void Draw()
        {
```

```
        // Здесь рисуется круг
        }
}

   // Конкретный класс "Квадрат"
   class Square extends Shape
{
        // Цвет квадрата
        int Color;

        // Координаты верхнего левого угла
        Coordinates StartPoint;

        // Нарисовать квадрат
        public void Draw()
        {
        // Здесь рисуется квадрат
        }
}
```

Теперь у нас есть все необходимое для рисования фигур. Можно создавать их в оперативной памяти и рисовать на экране:

```
Point point = new Point();
Circle circle = new Circle();
Square square = new Square();
point.Draw();
circle.Draw();
square.Draw();
```

В качестве дополнительного "бесплатного пирожка" мы получили возможность хранить все унаследованные от Shape объекты в одном массиве, не обращая внимания на их тип:

```
// Создаем массив
Shape[] shape = new Shape[3];
shape[0] = new Point();
shape[1] = new Circle();
shape[2] = new Square();
```

Мы даже можем вызывать методы Draw для всех элементов этого массива, не заботясь

об их типе. Руководствуясь идеей полиморфизма, Java сам отследит тип объектов и вызовет корректный метод:

```
shape[0].Draw(); // Вызывает Point.Draw();
shape[1].Draw(); // Вызывает Circle.Draw();
shape[2].Draw(); // Вызывает Square.Draw();
```

Я специально упростил пример, оставив лишь голый шаблон, и не указал конструкторов классов, методы инициализации, методики рисования и т. п. Если хотите, в качестве упражнения можете дописать эту программу до конца.

Два замечания напоследок:

- абстрактный метод не имеет тела, поскольку мы не знаем, что будет в нем;
- если унаследовать класс от абстрактного, но оставить нереализованным хотя бы один его абстрактный метод, то унаследованный класс также будет абстрактным. Чтобы избавиться от "абстрактности", необходимо реализовать код для всех абстрактных методов абстрактного класса-предка.

2. Интерфейсы

Java предоставляет программисту еще одно средство, родственное классам, — интерфейсы. Интерфейс — это набор абстрактных методов, которые не содержат никакого кода. По своему предназначению интерфейсы похожи на абстрактные классы, хотя между ними имеются некоторые существенные различия. Так, например, интерфейсы, в отличие от абстрактных классов, могут быть только public или private. Методы, описанные внутри интерфейсов, всегда доступны (public) и абстрактны (abstract). Данные, декларированные в интерфейсе, изначально имеют атрибуты final, public и static, т. е. неизменяемы. Иногда это удобно, а иногда накладывает серьезные ограничения на применение интерфейсов. Но тут уж ничего не поделаешь — таковы правила языка.

Интерфейсы дают возможность программисту описывать наборы методов, которые должен реализовать класс. К примеру, стандартный интерфейс для создания многопоточных приложений Runnable задается следующим образом:

```
shape[0].Draw(); // Вызывает Point.Draw();
```

```
shape[1].Draw(); // Вызывает Circle.Draw();
shape[2].Draw(); // Вызывает Square.Draw();
```

Данное описание устанавливает прототип для метода Run, необходимого для запуска нового потока выполнения.

Для того чтобы использовать интерфейсы, от них должен быть унаследован класс, который реализует все шаблоны абстрактных методов, определенных в интерфейсе. Это можно сделать, использовав ключевое слово implements. Так, описание класса потока может выглядеть следующим образом:

```
public NewThread implements Runnable
{
public void run()
{
   // Здесь запускается новый поток
   // выполнения
}
}
```

Обратите внимание: ключевое слово implements (реализует) стоит в том месте, где обычно располагается ключевое слово extends, описывающее отношение наследования. Но встречаются и случаи, когда какой-нибудь класс наследует методы другого класса и одновременно реализует какой-нибудь интерфейс:

```
public class MyApplet
extends Applet implements Runnable
```

После такого упрощенного введения позволю себе описать понятия и синтаксис интерфейсов снова, но уже более формально. Итак, как уже было сказано, интерфейс — это набор описаний методов без реализации и констант. Такое средство может понадобиться для организации наследования из любого места иерархии. Описав, к примеру, интерфейс CustomLook с методом CustomPaint для создания элементов интерфейса с новым внешним видом, мы можем создавать по-новому выглядящие элементы на базе стандартных. При этом можно с одинаковым успехом создать на базе интерфейса CustomLook новый вид кнопки или новую строку ввода, и при этом не имеет значения, что кнопка и строка ввода располагаются в разных местах иерархии классов. Главное то, что их объединяет, — необходимость реализовать собственный

метод CustomPaint для нестандартного отображения элемента. В связи с этим отметим следующие случаи применения интерфейсов:

- если различные классы, расположенные в разных местах иерархии, имеют некую общность;
- если несколько классов должны реализовать некий общий набор методов;
- если требуется создать интерфейс без раскрытия деталей реализации класса.

Интерфейсы описываются по такой схеме:

```
public interface CustomLook
{
public abstract void NotifyStartPaint();
public abstract void CustomPaint ();
}
```

После того как интерфейс декларирован, его имя можно использовать наряду со стандартными типами и классами. Возвращаясь к примеру создания элементов пользовательского интерфейса с новым внешним видом, можно сказать, что вы имеете право создавать переменные типа CustomLook. Возникает интересная возможность: вы можете хранить в массиве элементов CustomLook любые классы, унаследованные от него (полиморфизм), и передавать эти классы в качестве параметра типа CustomLook, иначе говоря, приводить их к типу базового интерфейса, не теряя при этом их особенностей. И все это можно проделывать для классов, никак не связанных в рамках иерархии. Разве такое возможно в Си++?

Для облегчения понимания рассмотрим простой пример — создание элементов пользовательского интерфейса нестандартного вида. Сначала уточним задачу. Имеются несколько стандартных элементов интерфейса пользователя: кнопка (OldButton), строка ввода (OldInputLine) и пункт меню (OldMenuItem). Все эти элементы унаследованы от разных классов, никак не связанных между собой. Требуется создать на базе указанных выше элементов новые, отличающиеся по внешнему виду. Для этого нам потребуется, чтобы каждый новый элемент установил метод, отслеживающий начало рисования элемента на экране NotifyStartPaint, и новый метод рисования своего интерфейса CustomPaint. Оформим все новые требования как интерфейс CustomLook:

```
public interface CustomLook
{
   public abstract void NotifyStartPaint();
   public abstract void CustomPaint ();
}
```

На базе интерфейса CustomLook и старых элементов мы создаем новые элементы: кнопку (NewButton), строку ввода (NewInputLine) и пункт меню (NewMenuItem). Вот окончательный вариант каркаса программы:

```
public class NewButton
extends OldButton implements CustomLook
{
public void NotifyStartPaint()
{
// Код для перехвата начала рисования
}
public void CustomPaint ();
}
{
// Код для рисования кнопки нового
// внешнего вида
}
}

public class NewInputLine
extends OldInputLine implements CustomLook
{
public void NotifyStartPaint()
{
// Код для перехвата начала рисования
}
public void CustomPaint ();
}
{
// Код для рисования строки ввода
// нового внешнего вида
}
}
```

```
public class NewMenuItem
extends OldMenuItem implements CustomLook
{
public void NotifyStartPaint()
{
// Код для перехвата начала рисования
}
public void CustomPaint ();
}
{
// Код для рисования пункта меню нового
// внешнего вида
}
}
```

Таким образом, мы получили новые классы, как и раньше, не связанные между собой, но имеющие одинаковую функциональность. Их можно сохранить в массиве элементов типа CustomLook, несмотря на то, что все они имеют разных предков.

Кратко напомним ключевые моменты использования интерфейсов:

- программы, выполненные на языке Java, могут использовать интерфейсы, если нежелательно использование общего предка или добавление новых методов к общему абстрактному классу-предку Object;
- переменные типа какого-либо интерфейса могут содержать ссылки на классы, унаследованные от этого интерфейса;
- недостаточно, чтобы класс реализовал методы интерфейса; кроме того, из описания должно быть ясно, что класс представляет собой реализацию некоего интерфейса, иначе считается, что этот класс не реализует интерфейс;
- если класс, который наследует интерфейс, не полностью реализует набор его методов, он становится абстрактным и к нему применимы все правила для абстрактных классов.

3. Обработка исключительных ситуаций

Если ваша программа нарушит семантические правила языка Java, то виртуальная машина Java (JVM) немедленно отреагирует на это выдачей ошибки под названием *"исключительная ситуация"*. Пример такой ситуации — выход за рамки массива. Она

может возникнуть при попытке обратиться к элементу за пределами границы массива. Некоторые языки программирования никак не "реагируют" на ошибки программиста и позволяют ошибочным программам выполняться. Но Java не относится к таким языкам. И поэтому программа тщательно проверяет все места, где может возникнуть потенциальная ошибка, а при обнаружении ошибки возбуждаются (throw) исключительные ситуации. Если имеются обработчики таких ситуаций, они перехватывают их (catch) и обрабатывают надлежащим образом.

Программы на языке Java могут самостоятельно возбуждать исключительные ситуации, используя для этого оператор throw. В точке метода, где встречается throw, выполнение метода прерывается и управление передается в тот метод, который вызвал ошибочный. Если исключительная ситуация может быть обработана методом, то вызывается его обработчик. Если же это невозможно, то поток управления передается дальше, и так происходит до того момента, когда исключительная ситуация не будет перехвачена или пока ее не перехватит виртуальная машина Java. В последнем случае выполнение программы прерывается и выводится сообщение об ошибке.

В языке Java каждая исключительная ситуация реализуется как экземпляр класса Throwable или его наследников. Когда в программе нужно отследить возможную исключительную ситуацию, в ней устанавливается обработчик (несколько обработчиков). На практике это оформляется в виде так называемого блока try-catch:

```
try{
    // Здесь возможно возбуждение
    // исключительной ситуации
} catch (ТипИсключительнойСитуации)
{
    // Здесь производится обработка
// перехваченной исключительной
// ситуации
}
```

Но не всегда исключительные ситуации — это фатальный сбой. Может, к примеру, оказаться, что программа просто не нашла какой-то файл в каталоге. В этом случае можно перехватить такую ошибку и в обработчик исключительной ситуации вставить оператор вызова диалоговой панели, где пользователь укажет местоположение этого файла. После разрешения этой проблемы программа может быть запущена с того

места, где ее выполнение было прервано.

Обычно все методы, в которых может возникнуть исключительная ситуация, описываются особым образом. Например:

```
static void SomeMethod ()
throws FileNotFoundException {-}
```

В этом описании оператор throws обозначает, что метод потенциально может создать/вызвать исключительную ситуацию FileNotFoundException, поскольку не найден какой-либо файл. Теперь любой вызов этого метода в программе должен быть обрамлен описанием блока try-catch, иначе компилятор выдаст ошибку и не обработает исходный текст вашей программы. Корректное решение проблемы выглядит примерно следующим образом:

```
try{
 ..
 static void SomeMethod ();
 ..
} catch (FileNotFoundException exception){
        // Предпринимаем действия
        // по устранению ошибки
}
```

Конечно, может показаться, что этот способ несколько расточителен и трудоемок. Но зато он гарантирует, что вы обработаете нештатную ситуацию, а не оставите ее "в подарок" пользователю вашей программы.

Существуют и более сложные понятия, например идеология обработки исключений или блоки try-finally. Однако того, что вы прочитали, в большинстве случаев вполне достаточно для повседневной работы.

Мир ПК #01/97