Урок 1

Обобщения

Понятие обобщения. Обобщенные классы, методы и интерфейсы. Наследование обобщенных классов. Ограничения при работе с обобщениями.

Небольшая задача

Представим что перед нами поставлена задача создать класс, который позволит хранить в себе один объект любого типа. В таком случае мы можем создать класс SimpleBox, у которого будет единственное поле типа Object, в которое можно будет записать объект абсолютно любого типа. Далее для проверки работы нашего класса напишем немного кода в классе BoxDemoApp.

```
public class SimpleBox {
    private Object obj;

    public Object getObj() {
        return obj;
    }

    public void setObj(Object obj) {
        this.obj = obj;
    }

    public SimpleBox(Object obj) {
        this.obj = obj;
    }
}

public class BoxDemoApp {
    public static void main(String[] args) {
        SimpleBox intBox1 = new SimpleBox(20);
        SimpleBox intBox2 = new SimpleBox(30);

        if (intBox1.getObj() instanceof Integer && intBox2.getObj() instanceof
Integer) {
            int sum = (Integer)intBox1.getObj() + (Integer)intBox2.getObj();
            System.out.println("sum = " + sum);
        } else {
            System.out.println("Содержимое коробок отличается по типу");
        }

        // вызвали какой-нибудь метод, которому отдали intBox1
        // и этот метод кладет в коробку String
        intBox1.setObj("Java");

        // продолжаем наш код, и при выполнении получим ClassCastException
        int secondSum = (Integer)intBox1.getObj() + (Integer)intBox2.getObj();
    }
}
```

В первых строках кода метода main() мы создаем две коробки intBox1 и intBox2, в которые при инициализации складываем значения 20 и 30. Представим что нам надо вытащить из коробок числа и получить их сумму. Для того чтобы вытащить из коробки число можно применить метод getObj(), но он вернет нам не int а Object, следовательно нужно еще добавить приведение к типу Integer. Во-вторых, чтобы не получить ClassCastException, перед строками с приведением типов желательно делать проверку (instanceof) на то, что в коробках лежит именно то, что мы ожидаем. В результате мы получим сумму чисел из коробок и напечатаем ее в консоль.

Представим что в коде мы вызываем некий метод, и отдаем туда ссылку на intBox1. Выполняемый метод может положить в нашу коробку все что угодно, например, строку Java. И если после этого, мы попытаемся сложить содержимое наших коробок, забыв при этом сделать instanceof, то получим ClassCastException.

Подводя итог, можно выделить три проблемы при использовании такого подхода:

- Каждый раз, когда мы хотим вытащить данные из нашей универсальной коробки, нам необходимо выполнять приведение типов;
- Чтобы не получить ClassCastException, перед каждым приведением типов, необходимо делать проверку типов данных с помощью instanceof;
- Если мы где-то будем применять приведение типов, и забудем прописать instanceof, то появится вероятность появления ClassCastException в этой части кода.

Для того, чтобы решить эти проблемы, в Java применяются обобщения.

Понятие обобщения

Обобщения – это параметризованные типы, которые позволяют объявлять обобщенные классы, интерфейсы и методы, где тип данных, которыми они оперируют, указан в виде параметра. Обобщения добавляют в Java безопасность типов и делают управление проще. Исключается необходимость применять явные приведения типов, так как благодаря обобщениям все приведения выполняются неявно, в автоматическом режиме.

Чтобы понять почему нам больше не надо делать явное приведение типов, и откуда берется какая-то безопасность, рассмотрим простой пример обобщенного класса.

```
public class TestGeneric<T> {
    private T obj;

    public TestGeneric(T obj) {
        this.obj = obj;
    }

    public T getObj() {
        return obj;
    }

    public void showType() {
        System.out.println("Тип T: " + obj.getClass().getName());
    }
}

public class GenericsDemoApp {
    public static void main(String args[]) {
        TestGeneric<String> genStr = new TestGeneric<>("Hello");
        genStr.showType();
        System.out.println("genStr.getObject(): " + genStr.getObj());
        TestGeneric<Integer> genInt = new TestGeneric<>(140);
        genInt.showType();
        System.out.println("genInt.getObject(): " + genInt.getObj());
        int valueFromGenInt = genInt.getObj();
        String valueFromGenString = getStr.getObj();
```

```
        // genInt.setObj("Java"); // Ошибка компиляции !!!
        }
}
```

Важно!

- При получении значений из genInt и genStr не требуется преобразование типов, genInt.getObj() сразу возвращает Integer, а genStr.getObj() - String. То есть приведение типов выполняется неявно и автоматически.
- Если объект создан как TestGeneric<Integer> genInt, то мы не сможем записать в него строку: genInt.setObj("Java"). При попытке написать такую строку кода, мы получим ошибку на этапе компиляции. То есть обобщения отслеживают корректность используемых типов данных.

Эти две особенности использования обобщений и приводят к повышению безопасности типов данных, и упрощению работы при написании кода.

Результат выполнения:

```
Тип T: java.lang.String
genStr.getObject(): Hello
Тип T: java.lang.Integer
genInt.getObject(): 140
```

В объявлении класса public class TestGeneric<T>, T представляет собой имя параметра типа, на место которого при создании объекта класса TestGeneric будет подставлен конкретный тип данных. У объекта genStr из примера выше T = String, а для genInt T = Integer. То есть используя обобщения мы можем создавать переменные типы данных.

Важно! Обобщения работают только с ссылочными типами данных. Для работы с примитивами надо будет использовать классы-обертки.

Буква T в объявлении обобщенного класса не является обязательной, вы можете вместо нее использовать любую другую букву (E, N, V, …), это всего лишь имя переменного типа.

Ссылка на одну специфическую версию обобщенного типа не обладает совместимостью с другой версией того же обобщенного типа. Например:

```
TestGeneric<String> genStr = new TestGeneric<>("Hello");
TestGeneric<Integer> genInt = new TestGeneric<>(140);
genInt = genStr; // Ошибка
```

Объекты genInt и genStr принадлежат классу TestGeneric<T>, но представляют собой ссылки на разные типы – ведь типы их параметров отличаются.

Обобщенный класс с несколькими параметрами типа

Для обобщенного типа можно объявлять более одного параметра, используя список, разделенный запятыми:

```
public class TwoGen<T, V> {
   private T obj1;
   private V obj2;

   public TwoGen(T obj1, V obj2) {
       this.obj1 = obj1;
       this.obj2 = obj2;
   }

   public void showTypes() {
       System.out.println("Тип T: " + obj1.getClass().getName());
       System.out.println("Тип V: " + obj2.getClass().getName());
   }

   public T getObj1() {
       return obj1;
   }

   public V getObj2() {
       return obj2;
   }
}

public class SimpleGenApp {
  public static void main(String args[]) {
    TwoGen<Integer, String> twoGenObj = new TwoGen<Integer, String>(555,
"Hello");
    twoGenObj.showTypes();
    int intValue = twoGenObj.getObj1();
    String strValue = twoGenObj.getObj2();
    System.out.println(intValue);
    System.out.println(strValue);
  }
}
```

Ограниченные типы

В примерах, рассмотренных выше, параметры типов можно было заменить любыми типами классов. Но иногда существует необходимость ограничить набор этих перечень типов.

Создадим обобщенный класс, который содержит в себе массив (мы предполагаем что это будет массивом чисел любого типа) и метод, возвращающий среднее значение этого массива.

```
public class Stats<T> {
   private T[] nums;

   public Stats(T... nums) {
       this.nums = nums;
   }

   public double avg() {
       double sum = 0.0;
       for(int i = 0; i < nums.length; i++) {
           sum += nums[i]; // Ошибка
       }
       return sum / nums.length;
   }
}
```

В методе avg() мы создаем временную double переменную sum и пытаемся с помощью нее сложить все числа, лежащие в массиве nums. Но первая проблема заключается в том, что Java видит что мы пытаемся складывать объекты (хотя мы собираемся хранить в этом массиве только числа), и конечно же выдает ошибку. Вторая же проблема заключается в том, что если даже это будет массив Integer, то у каждого объекта при суммировании надо будет вызывать метод doubleValue(), который будет преобразовывать каждый объект Integer в примитив double. Давайте попробуем разобраться с двумя этими проблемами.

При работе с обобщениями будем использовать ограниченные типы. Когда указывается параметр типа, можно создать ограничение сверху, которое укажет суперкласс, от которого должны быть унаследованы все аргументы типов. Для этого используется ключевое слово extends:

```
<T extends суперкласс>
```

Важно! В роли ограничителя сверху может выступать не только класс, но и один или несколько интерфейсов. Для указания нескольких элементов используется оператор &.
<T extends Cat>
<T extends Animal & Serializable>
<T extends Serializable>
<T extends Cloneable & Serializable>
Обратите внимание, что даже если вы ограничиваете T интерфейсом, все равно используется ключевое слово extends.
Если в качестве ограничителя используется класс и интерфейс, то класс должен быть указан первым.

Это означает, что параметр T может быть заменен только самим суперклассом или его подклассами. Он объявляет включающую верхнюю границу. Можно использовать ограничение сверху, чтобы исправить класс Stats, указав класс Number как верхнюю границу используемого параметра типа. Посмотрим что нам это даст.

```
public class Stats<T extends Number> {
   private T[] nums;

   public Stats(T... nums) {
       this.nums = nums;
   }

   public double avg() {
       double sum = 0.0;
       for(int i = 0; i < nums.length; i++) {
           // У nums[i] появился метод doubleValue() из класса Number
           // который позволяет любой числовой объект привести к double
           sum += nums[i].doubleValue();
       }
       return sum / nums.length;
   }
}

public class StatsDemoApp {
  public static void main(String args[]) {
    Stats<Integer> intStats = new Stats<Integer>(1, 2, 3, 4, 5);
    System.out.println("Ср. знач. intStats равно " + intStats.avg());
    Stats<Double> doubleStats = new Stats<Double>(1.0, 2.0, 3.0, 4.0, 5.0);
    System.out.println("Ср. знач. doubleStats равно " + doubleStats.avg());
    // Это не скомпилируется, потому что String не является подклассом Number
    // Stats<String> strStats = new Stats<>("1", "2", "3", "4", "5");
    // System.out.println("Ср. знач. strStats равно " + strStats.avg());
  }
}
```

Объявление public class Stats<T extends Number> сообщает компилятору, что все объекты типа T являются подклассами класса Number, и поэтому могут вызывать метод doubleValue(), как и любой другой из класса Number. Ограничивая параметр T, мы предотвращаем создание нечисловых объектов класса Stats.

Использование метасимвольных аргументов

Давайте попробуем добавить в класс Stats метод sameAvg(), который будет проверять равенство средних значений массивов двух объектов типа Stats, независимо от того, какого типа числовые значения в них содержатся. Допустим в одном будет new int[] { 1, 2, 3 }, а во втором new double { 1.0, 2.0, 3.0 };

Метод sameAvg() на вход должен принимать объект типа Stats и сравнивать его среднее значение со средним значением вызывающего объекта. Код применения этого метода может выглядеть вот так:

```
public class StatsDemoApp {
   public static void main(String args[]) {
      Stats<Integer> intStats = new Stats<>(1, 2, 3, 4, 5);
      Stats<Double> doubleStats = new Stats<>(1.0, 2.0, 3.0, 4.0, 5.0);
      if (intStats.sameAvg(doubleStats)) {
         System.out.println("Средние значения равны");
      } else {
         System.out.println("Средние значения не равны");
      }
   }
}
```

Класс Stats является обобщенным, и при написании метода sameAvg() возникает вопрос: какой тип указать для аргумента Stats? Попробуем использовать обобщенный тип T.

```
public class Stats<T extends Number> {
   // ...
   public boolean sameAvg(Stats<T> another) {
      return Math.abs(this.avg() - another.avg()) < 0.0001;
   }
   // ...
}
```

Заметка. Чтобы не столкнуться с ошибкой округления при сравнении двух дробный чисел, мы сравниваем средние значения в пределах дельты 0.0001

Такой код будет работать только с объектом класса Stats, тип которого совпадает с вызывающим объектом. Если вызывающий объект имеет тип Stats<Integer>, то параметр another обязательно должен принадлежать к аналогичному типу.

```
public class StatsDemoApp {
   public static void main(String args[]) {
      Stats<Integer> intStats1 = new Stats<>(1, 2, 3, 4, 5);
      Stats<Integer> intStats2 = new Stats<>(2, 1, 3, 4, 5);
      Stats<Double> doubleStats = new Stats<>(1.0, 2.0, 3.0, 4.0, 5.0);
      System.out.println(intStats1.sameAvg(intStats2)); // Так работает
      // System.out.println(intStats1.sameAvg(doubleStats)); // Ошибка
      // (T = Integer) != (T = Double)
   }
}
```

Чтобы создать обобщенную версию метода sameAvg(), следует использовать метасимвольные аргументы. Это средство обобщений Java, которое обозначается символом «?» и представляет собой неизвестный тип.

```
public class Stats<T extends Number> {
   // ...
   public boolean sameAvg(Stats<?> another) {
      return Math.abs(this.avg() - another.avg()) < 0.0001;
   }
   // ...
}
```

Stats<?> соответствует любому объекту класса Stats. Можно сравнивать средние значения любых двух объектов этого класса.

```
public class Stats<T extends Number> {
   private T[] nums;

   public Stats(T[] nums) {
       this.nums = nums;
   }

   public double avg() {
       double sum = 0.0;
       for (int i = 0; i < nums.length; i++) {
           sum += nums[i].doubleValue();
       }
       return sum / nums.length;
   }

   public boolean sameAvg(Stats<?> another) {
       return Math.abs(this.avg() - another.avg()) < 0.0001;
   }
}

public class WildcardDemoApp {
  public static void main(String args[]) {
    Stats<Integer> iStats = new Stats<>(1, 2, 3, 4, 5);
    System.out.println("Среднее iStats = " + iStats.avg());

    Stats<Double> dStats = new Stats<>(1.1, 2.2, 3.3, 4.4, 5.5);
    System.out.println("Среднее dStats = " + dStats.avg());

    Stats<Float> fStats = new Stats<>(1.0f, 2.0f, 3.0f, 4.0f, 5.0f);
    System.out.println("Среднее fStats = " + fStats.avg());

    System.out.print("Средние iStats и dStats ");
    if (iStats.sameAvg(dStats)) {
      System.out.println("равны");
    } else {
      System.out.println("отличаются");
    }

    System.out.print("Средние iStats и fStats");
    if (iStats.sameAvg(fStats)) {
      System.out.println("равны");
```

```
    } else {
      System.out.println("отличаются");
    }
  }
}
```

Результат работы программы:

```
Среднее iStats = 3.0
Среднее dStats = 3.3
Среднее fStats = 3.0
Средние iStats и dStats отличаются
Средние iStats и fStats равны
```

Метасимвольный аргумент не влияет на тип создаваемого объекта класса Stats. Это зависит от extends в объявлении класса Stats.

Ограничения

Ограничения на статические члены

Никакой статический член не может использовать тип параметра, объявленный в его классе.

```
public class WrongGenericClass<T> {
  static T data; // Неверно, нельзя создать статические переменные типа T
  static T getData() { return data; } // Неверно, ни один статический метод не
может использовать T
}
```

Нельзя объявить статические члены, использующие обобщенный тип. Но можно объявлять обобщенные статические методы, определяющие их собственные параметры типа.

Ограничения обобщенных исключений

Обобщенный класс не может расширять класс Throwable. Значит, создать обобщенные классы исключений невозможно.

Практическое задание

1. Написать метод, который меняет два элемента массива местами (массив может быть любого ссылочного типа);
2. Написать метод, который преобразует массив в ArrayList;
3. Задача:
    a. Даны классы Fruit, Apple extends Fruit, Orange extends Fruit;
    b. Класс Box, в который можно складывать фрукты. Коробки условно сортируются по типу фрукта, поэтому в одну коробку нельзя сложить и яблоки, и апельсины;
    c. Для хранения фруктов внутри коробки можно использовать ArrayList;
    d. Сделать метод getWeight(), который высчитывает вес коробки. Задать вес одного фрукта и их количество: вес яблока – 1.0f, апельсина – 1.5f (единицы измерения не важны);
    e. Внутри класса Box сделать метод compare(), который позволяет сравнить текущую коробку с той, которую подадут в compare() в качестве параметра. true – если их массы равны, false в противоположном случае. Можно сравнивать коробки с яблоками и апельсинами;
    f. Написать метод, который позволяет пересыпать фрукты из текущей коробки в другую. Помним про сортировку фруктов: нельзя яблоки высыпать в коробку с апельсинами. Соответственно, в текущей коробке фруктов не остается, а в другую перекидываются объекты, которые были в первой;
    g. Не забываем про метод добавления фрукта в коробку.

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл. Java. Библиотека профессионала. Том 1. Основы;
2. Стив Макконнелл. Совершенный код;
3. Брюс Эккель. Философия Java;
4. Герберт Шилдт. Java 8: Полное руководство;
5. Герберт Шилдт. Java 8: Руководство для начинающих.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Герберт Шилдт. Java. Полное руководство // 8-е изд.: Пер. с англ. – М.: Вильямс, 2012. – 1 376 с.
2. Герберт Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. – М.: Вильямс, 2015. – 720 с.