Урок 1

Обобщения

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

Понятие обобщения

Обобщения – это параметризованные типы. Они позволяют объявлять классы, интерфейсы и методы (они называются обобщенными), где тип данных, которыми они оперируют, указан в виде параметра.

Язык Java известен тем, что предоставляет возможность создавать в классы, интерфейсы и методы, оперирующие ссылками на тип Object. Это придает им характер обобщенных. Object является суперклассом: ссылка на него может обращаться к объекту любого типа. Но возникает проблема с безопасностью типов.

Именно обобщения добавили в Java безопасность типов и сделали управление проще. Нет необходимости применять явные приведения, чтобы транслировать объекты Object в реальные типы данных. Благодаря обобщениям все приведения выполняются неявно, в автоматическом режиме.

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 GenericsDemo1 {
  public static void main(String args[]) {
    TestGeneric<String> wStr = new TestGeneric<String>("Hello");
    wStr.showType();
    String x = wStr.getObj();
    System.out.println("x: " + x);
    TestGeneric<Integer> wInt = new TestGeneric<Integer>(140);
    wInt.showType();
    Integer x2 = wInt.getObj();
    System.out.println(x2);
  }
}
```

Результат выполнения:

```
Тип T: java.lang.String
x: Hello
Тип T: java.lang.Integer
x2: 140
```

Обратим внимание на объявление класса **public class TestGeneric<T>**, где T – имя параметра типа (заключено в угловые скобки). Здесь будет подставлено имя реального типа. Например, при создании объекта класса **TestGeneric<String>** в имя параметра типа – это String.

Обобщения работают только с ссылочными типами данных!

Когда добавляется объект обобщенного типа, аргумент, переданный в качестве параметра, должен обладать ссылочным типом данных. Для примитивных типов данных можно использовать оболочки типов.

Отличие обобщенных типов в зависимости от аргументов типа

Принимаем во внимание, что ссылка на одну специфическую версию обобщенного типа не обладает совместимостью с другой версией того же обобщенного типа. Например:

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

Объекты **wInt** и **wStr** принадлежат типу **TestGeneric<T>**, но представляют собой ссылки на разные типы – ведь типы их параметров отличаются. Так обобщения обеспечивают безопасность типов и предупреждают ошибки.
Обобщения повышают безопасность типов.

Вопрос: если так же работать с несколькими типами данных можно, просто указав класс Object в качестве типа данных и применяя верные приведения, то в чем параметризации класса TestGeneric?

Ответ: обобщения автоматически гарантируют безопасность типов во всех операциях, где задействован класс TestGeneric. Он исключает явное приведение и ручную проверку типов в программном коде.

Чтобы оценить преимущества работы с обобщениями, рассмотрим программу, которая создает необобщенный аналог класса TestGeneric.

```
public class TestNonGeneric {
  private Object obj;

  public TestNonGeneric(Object o) {
    this.obj = obj;
  }

  public Object getObj() {
    return obj;
  }

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

public class NonGenericDemo {
  public static void main(String args[]) {
    TestNonGeneric iObj;
    iObj = new TestNonGeneric(88);
    iObj.showType();
    int v = (Integer) iObj.getObj();
    System.out.println("значение: " + v ) ;
    TestNonGeneric strObj = new TestNonGeneric("Hello");
    strObj.showType();
    String str = (String) strObj.getObj();
    System.out.println("Значение: " + str);
    iObj = strObj;
    v = (Integer)iObj.getObj(); // Runtime Exception
  }
}
```

Здесь класс TestNonGeneric заменяет все обращения к типу T, обращаясь вместо него к Object. Это позволяет ему хранить объекты любого типа, как это делает и обобщенная версия. Но компилятор Java лишается реальных сведений о типе данных, сохраняемых в объекте класса TestNonGeneric. Это влечет неприятные последствия:

- При извлечении сохраненных данных потребуется явное приведение;
- Ошибки несоответствия типов не выявить до старта выполнения.

```
int v = (Integer)iObj.getObj();
```

Здесь возвращаемым типом метода **getObj()** является тип **Object**. Его надо привести его к типу **Integer**. В версии с обобщением приведение выполняется неявно. Отказываясь от обобщения, мы вынуждены проводить приведение явно – иначе программа не будет компилироваться. Мало того, что это неудобно, еще и риск ошибок существенно повышается.

```
iObj = strObj;
v = (Integer)iObj.getObj();
```

StrObj, который здесь является строкой, а не целым числом, присваивается объекту **iObj**. Следовательно, в **iObj** тоже записывается строка. При попытке присвоить значение **iObj** переменной типа **int,** мы получим **Runtime Exception.** Не применяя обобщения, компилятор Java не может это обнаружить.

Главное преимущество, которое дают обобщения – это создание безопасного кода, где ошибки несоответствия типов перехватываются компилятором. Благодаря обобщениям ошибки времени выполнения преобразуются в ошибки времени компиляции.

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 SimpleGen {
  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. Это во многих случаях удобно, но иногда предпочтительнее ограничить перечень типов, передаваемых в параметрах.

Создадим обобщенный класс, который содержит метод, возвращающий среднее значение массива чисел. Предусмотрим его использование для расчета среднего значения, включая целые числа и числа с плавающей точкой одинарной и двойной точности. Укажем тип числовых данных обобщенно, используя параметр типа.

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

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

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

Метод **avg()** класса **Stats** пытается получить версию типа **double** каждого числа в массиве **nums**, вызывая метод **doubleValue()**. Все числовые классы, включая **Integer** и **Double** – это подклассы Number, который определяет метод **doubleValue()**. Следовательно, этот метод доступен всем числовым классам-оболочкам.

Но компилятор не знает, что необходимо создавать объекты класса **Stats**, используя только числовые типы. При компиляции класса **Stats** возникает ошибка, так как метод **doubleValue()** неизвестен. Нужно довести до сведения компилятора, что в параметре T только числовые типы. Требуется еще и гарантировать, что будут передаваться только они.

В качестве решения Java предлагает ограниченные типы. Когда указывается параметр типа, можно создать ограничение сверху. Оно объявляет суперкласс, от которого должны быть унаследованы все аргументы типов. Для этого при указании параметра типа используется ключевое слово **extends**:

```
<T extends суперкласс>
```

Это означает, что параметр T может быть заменен только 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.

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

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

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

public class StatsDemo {
  public static void main(String args[]) {
    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.avg();
    System.out.println("Среднее значение iob равно " + v ) ;
    Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
    Stats<Double> dob = new Stats<Double>(dnums);
    double w = dob.avg();
    System.out.println("Среднее значение dob равно " + w ) ;
    // Это не скомпилируется, потому что String не является подклассом Number.
    // String strs[] = { "1", "2", "3", "4", "5" };
    // Stats<String> strob = new Stats<String>(strs);
    // double x = strob.avg();
    // System.out.println("Среднее значение strob равно " + v ) ;
  }
}
```

Класс Stats теперь объявлен как **public class Stats<T extends Number>**. Компилятор Java знает, что все объекты типа T могут вызывать метод **doubleValue()**, так как это метод класса Number, которым ограничен тип T.

Ограничивая параметр T, мы предотвращаем создание нечисловых объектов класса **Stats**. Дополнительно к использованию типа класса как ограничения можно применять тип одного или нескольких интерфейсов. В этом случае тип класса должен быть задан первым. Когда ограничение включает тип интерфейса, допустимы только реализующие его аргументы типа. При указании ограничения, имеющего класс и интерфейсы, для их объединения применяется оператор **&**.

```
public class Gen<T extends MyClass & MyInterface>
```

Параметр T ограничен классом **MyClass** и интерфейсом **MyInteface**. Любой тип, переданный параметру T, должен быть подклассом **MyClass** и иметь реализацию интерфейса **MyInteface**.

Использование метасимвольных аргументов

С помощью контроля типов можно получить полезные конструкции. Например, класс **Stats** предполагает, что можно добавить метод по **sameAvg()**. Он определяет, содержат ли два объекта класса **Stats** массивы, порождающие одинаковое среднее значение независимо от того, какого типа числовые значения в них содержатся.

*Если один объект содержит значения типа **double** 1.0, 2.0 и 3.0, а другой — целочисленные значения 2, 1 и 3, то среднее значение у них будет одинаковым.*

Чтобы реализовать метод **sameAvg()**, необходимо передать ему аргумент класса Stats, а затем сравнивать его среднее значение со средним значением вызывающего объекта. Если они равны, возвращается значение **true**.

Необходимый способ вызывать метод **sameAvg()**:

```
Integer inums[] = {1, 2, 3, 4, 5};
Double dnums[] = {1.1, 2.2, 3.3, 4.4, 5.5};
Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);
if (iob.sameAvg(dob)) {
   System.out.println("Средние значения одинаковы");
} else {
   System.out.println("Средние значения отличаются");
}
```

Класс **Stats** является обобщенным, и его метод **avg()** может работать с его объектами любого типа. Но при написании метода **sameAvg()** проблема может возникнуть сразу при попытке объявить параметр типа **Stats**. Это параметризованный тип – какой тип параметра необходимо указать для него? Попробуем решение, в котором T будет использоваться как параметр типа.

```
public boolean sameAvg(Stats<T> another) {
   return Math.abs(this.avg() - another.avg()) < 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 **ob** обязательно должен принадлежать к аналогичному типу.

Чтобы создать обобщенную версию метода **sameAvg()**, следует использовать метасимвольные аргументы. Это средство обобщений Java, которое обозначается символом «?» и представляет собой неизвестный тип. Только применяя метасимвольный аргумент возможно написать работающий метод **sameAvg()**.

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

Здесь часть **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[] o) {
       nums = o;
   }

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

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

public class WildcardDemo {
  public static void main(String args[]) {
    Integer inums[] = {1, 2, 3, 4, 5};
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.avg();
    System.out.println("Среднее для iob равно " + v);
    Double dnums[] = {1.1, 2.2, 3.3, 4.4, 5.5};
    Stats<Double> dob = new Stats<Double>(dnums);
    double w = dob.avg();
    System.out.println("Среднее для dob равно " + w);
    Float fnums[] = {1.0f, 2.0f, 3.0f, 4.0f, 5.0f};
    Stats<Float> fob = new Stats<Float>(fnums);
    double x = fob.avg();
    System.out.println("Среднее для fob равно " + x);
    System.out.print("Средние iob и dob ");
    if (iob.sameAvg(dob)) {
      System.out.println("равны");
    } else {
      System.out.println("отличаются");
    }
    System.out.print("Средние iob и fob ");
    if (iob.sameAvg(fob)) {
      System.out.println("равны.");
    } else {
      System.out.println("отличаются");
    }
  }
}
```

Результат работы программы:

```
Среднее для iob равно 3.0
Среднее для dob равно 3.3
Среднее для fob равно 3.0
Средние iob и dob отличаются.
Средние iob и fob равны.
```

Метасимвольный аргумент не влияет на тип создаваемого объект 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 TestWrong<T> {
  static T obj;                    // Неверно, нельзя создать статические
переменные типа T
  static T getobj() { return obj; } // Неверно, ни один статический метод не
может использовать T
}
```

Нельзя объявить статические члены, использующие тип параметра, объявленный в окружающем классе. 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.

Домашнее 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, Orange;
    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.