Создание экземпляров параметризованных типов на основе шаблонов (wildcards)

Мы продолжаем цикл наших статей о нововведениях в Java Generics весии J2SE 1.5 (см. №4 за 2003 и №5 за 2004). В предыдущей статье мы лишь немного коснулись темы шаблонов (wildcards). Теперь пришло время более детально изучить данный вопрос. Как следует из названия, мы рассмотрим шаблоны (wildcards) и его виды на примере создания экземпляров параметризированіх типов.

Параметризованные типы и методы (Java Generics) в релизе J2SE 1.5 стали новой особенностью языка программирования Java. Предметом обсуждения в этой статье являются шаблоны в Java Generics. Шаблоны можно использовать для создания экземпляров параметризованных типов. Самый простой шаблон - это вопросительный знак (?). Он используется, например, в объявлении типа (наподобие List<?>).

Для того чтобы выяснить назначение шаблонов, рассмотрим иерархию классов для геометрических фигур с абстрактным суперклассом Shape и несколькими подклассами, такими как Circle, Triangle, Rectangle и т.д. Все классы для фигур имеют метод draw():

```
public abstract class Shape {
  public abstract void draw();
}
public final class Circle {
  ...
  public void draw() { ... }
  ...
}
public final class Triangle {
  ...
  public void draw() { ... }
  ...
}
```

...

Если бы мы захотели реализовать для заданной коллекции фигур метод drawAllShapes(), каким образом можно было объявить этот метод? Рассмотрим первый подход:

```
public void drawAllShapes(
  Collection<Shape> c) {
  for (Shape s : c) {
    s.draw();
  }
}
```

Обратите внимание на используемый здесь своеобразный синтаксис цикла for-loop , который является другим нововведением в Java 1.5. Такое выражение как:

```
for (Shape s : c) { s.draw(); }
```

является краткой формой записи для цикла:

```
for (Iterator _i = c.iterator();
  _i.hasNext(); ) { Shape s =
  _i.next(); s.draw(); }
```

Сам метод можно вызвать таким образом:

```
Collection<Shape> c =
  new LinkedList<Shape>();
... fill the collection with
  shapes ...
drawAllShapes(c);
```

Однако следующая форма вызова становится недопустимой:

```
Collection<Triangle> c =
  new LinkedList<Triangle>();
... fill the collection with
  triangles ...
drawAllShapes(c);
```

Хотя треугольники и являются фигурами, имеющими метод draw(), нельзя применительно к коллекции треугольников как к аргументу вызвать метод drawAllShapes(), т.к. параметр метода объявлен как имеющий тип Collection<Shape>, а соответствующий аргумент – типа Collection<Triangle>. Эти типы несовместимы друг с другом. В общем, нет отношения совместимости присваивания между различными экземплярами одного и того же параметризованного типа. Collection<Triangle> не может быть присвоено переменной типа Collection<Shape>, с чем не всегда можно интуитивно согласиться.

Можно предполагать совместимость по присваиванию, в частности, применительно к коллекциям, поскольку они подобны массивам, а встроенные типы массивов с различными типами элементов совместимы по присваиванию, если для типов элементов существует отношение супертип/подтип. Например, метод, объявленный как принимающий параметр типа Shape[], должен принимать аргумент типа Triangle[]. Такой тип отношения, известный как *ковариантность*, предполагает совместимость по присваиванию массива подтипа с массивом супертипа, в данном случае Triangle[] с Shape[]. Подобных ковариантных отношений не существует для экземпляров параметризованных типов.

Глава 1. Знакомство с Wildcards

1.1 Виды шаблона

Чтобы восполнить отсутствие ковариантности для экземпляров параметризованных типов, разработчики языка изобрели шаблоны (групповые символы) wildcards. Рассмотрим объявление метода drawAllShapes() с применением такого шаблона:

```
public void drawAllShapes(
  Collection<? extends Shape> c)
  {
  for (Shape s : c) {
    s.draw();
  }
}
```

Выражение ? extends Shape является примером шаблона wildcard. Оно символизирует семейство типов, а именно, все типы, являющиеся подтипами Shape—включая сам тип Shape. Следовательно, реализация Collection<? extends Shape> означает семейство типов – экземпляров класса Collection для типа элемента из семейства подтипов Shape.

Данное объявление параметра позволяет нам вызвать метод drawAllShapes() как ранее, предоставляя в распоряжение метода коллекцию фигур:

```
Collection<Shape> c =
  new LinkedList<Shape>();
... fill the collection with
  shapes ...
drawAllShapes(c);
```

Кроме того, можно вызвать метод с коллекцией, содержащей подтипы Shape:

```
Collection<Triangle> c =
  new LinkedList<Triangle>();
... fill the collection with
  triangles ...
drawAllShapes(c);
```

В сущности, шаблоны используются преимущественно для определения относительно “мягких” API; то есть, позволяют объявлять методы, для которых возможен широкий диапазон типов аргументов— более широкий, чем это было бы возможным без шаблонов.

Выражение ? extends Shape – только один из вариантов шаблона. Всего есть три различных вида: ? extends Type – шаблон “верхнего предела”, обозначающий семейство типов, которые являются подтипами Type; ? super Type – шаблон “нижнего предела”, т.е. для семейства супертипов Type; ? – неограниченный шаблон для обозначения всех типов.

Шаблоны можно применять для инициализации параметризованных типов, например, можно формировать такие типы как Collection<? extends Shape>, Comparable<? super Triangle>, и LinkedList<?>. Результатами будут семейства типов: Collection<? extends Shape> представляет собой множество типов коллекций, тип элементов которых является подтипом от Shape. Подобно этому, Comparable<? super Triangle> обозначает реализации интерфейса Comparable, делающие возможным сравнение объектов-супертипов Triangle. LinkedList<?> включает в себя все реализации параметризованного типа LinkedList, независимо от типа элемента.

Шаблоны можно использовать для инициализации параметризованных типов, но не для методов. Рассмотрим параметризованный метод <T extends Comparable<T>> T max(Collection<T> c). Как правило, мы не задаем тип, используемый для инициализации; а просто вызываем параметризованный метод как обычный не параметризованный, а заботу о правильном выборе типа аргумента берет на себя компилятор. При желании мы можем отключить автоматический выбор типа и задать его явно. Затем надо вызвать метод max() как <Shape> max(someShapeCollection). Однако, шаблон не должен присутствовать в явном задании параметра типа, т.е. нельзя вызвать метод как <? extends Shape> max(someShapeCollection). Шаблоны можно использовать только для инициализации параметризованных типов, а не для параметризованных методов.

Реализации параметризованных типов на основе шаблонов могут использоваться как аргументы и возвращаемые типы для методов, в качестве аргументов типа для других параметризованных типов, а также для объявления переменных. Они *не могут* использоваться для создания объектов или массивов. Wildcard-экземпляры недопустимы в выражении new. Такие экземпляры не являются типами, а представляют собой метки -заполнители для членов из семейства типов. В известном смысле wildcard-экземпляр подобен интерфейсу: возможно объявление переменных интерфейсных типов, но невозможно создание объектов этих типов. Созданные объекты должны принадлежать типу класса, реализующему интерфейс. Мы можем, подобно экземплярам шаблона, объявлять переменные типов на основе шаблонов, но не можем создавать объекты таких типов; созданные объекты должны относиться к конкретному экземпляру семейства, обозначенного в шаблоне. Вот пример:

```
LinkedList<?> list1 =
  new LinkedList<String>();
  // fine
LinkedList<?> list2 =
  new LinkedList<?>();
  // error: cannot create
  // objects of a wildcard type

LinkedList<? extends Shape>
  list3 =
  new LinkedList<Triangle>();
  // fine
LinkedList<? extends Shape>
  list4 =
  new LinkedList<String>();
  // error: does not belong to
  // the family
```

1.2 Ограничения доступа

Ссылочные переменные или параметры wildcard-типов ссылаются на объекты конкретных типов. Доступ к объекту по ссылке посредством ссылочной переменной или параметра шаблонного типа ограничен. Ограничения могут различаться в зависимости от вида используемого шаблона.

Шаблон ? запрещает вызов методов, имеющих аргументы типа, обозначенного в шаблоне. Например, шаблон List<?> означает произвольный экземпляр параметризованного типа List; и тип элемента неизвестен, он может быть любого типа. Если мы попытаемся добавить элемент в список из элементов неизвестного типа, то не существует способа определения “правильности” типа добавляемого элемента. Для иллюстрирования вышесказанного приведем следующий

пример. Ссылочная переменная типа List<?> ссылается на объект типа, являющегося конкретным экземпляром параметризованного типа List:

```
List<?> list =
  new LinkedList<String>();
```

Ссылочная переменная list ссылается на список строк. Если допустить вызов метода add() посредством ссылочной переменной List<?>, то, возможно, будет предпринята попытка вставить произвольный тип элемента в список:

```
void f(List<?> list) {
  list.add(new Date());
  // error: must not invoke
  //methods that take arguments
  // of the "?" type
}
```

Метод, объявленный как f(List<?> list) принимает любой экземпляр List. Компилятор никаким образом не может определить, на какой именно экземпляр List ссылается параметр типа List<?>. Следовательно, компилятор не должен допускать вызов метода add(), поскольку тем самым была бы разрешена вставка “чужеродных” элементов в список.

Общее правило состоит в том, что посредством ссылочной переменной неограниченного шаблонного типа нельзя вызвать какой-либо метод объекта, на который ссылаются, если ему передается аргумент неизвестного типа (обозначенного в шаблоне знаком “?”). Единственным исключением из этого правила является использование null в качестве аргумента, поскольку null не имеет типа. Следовательно, допустим такой вызов:

```
void f(List<?> list) {
   list.add(null);  // fine
}
```

Напротив, методы, *возвращающие* объект неизвестного типа, *можно* вызывать посредством ссылочной переменной неограниченного шаблонного типа:

```
void f(List<?> list) {
  Object ret = list.get(0);
  // fine
}
```

Такой вызов метода get() допустим, потому что получение элементов неизвестного типа не создает никаких проблем; не имеет значения, элементы какого типа сохраняются в списке, т.к. элементы совместимы по присваиванию с Object.

Аналогичные правила применимы и к другим шаблонным типам. Посредством ссылочной переменной шаблонного типа "extends" нельзя вызвать такой метод объекта, на который ссылаются и которому передается аргумент неизвестного типа (обозначенного в шаблоне как "extends") кроме null. Возможен вызов методов, возвращающих объект неизвестного типа. Ниже приведен пример, иллюстрирующий ограничения:

```
void f(List<? extends Shape>
  list) {
  list.add(new Triangle());
  // error: must not invoke
  // methods that take arguments
  // of the "? extends Shape"
  // type
}
```

1.3 «Умный» шаблон

Итак, компилятор не располагает информацией о том, обращается ли ссылочная переменная типа List<? extends Shape> к списку треугольников, как предполагается в вызове метода add(). Ссылочная переменная типа List<? extends Shape> точно так же может ссылаться и

на список окружностей или прямоугольников. В этом случае включение треугольника в список нежелательно.

Напротив, извлечение элементов из списка безопасно, поскольку все элементы принадлежат Shapes:

```
void f(List<? extends Shape>
  list) {
  Shape ret = list.get(0);
    // fine
}
```

Заметим, что для шаблонного типа "extends", в отличие от неограниченного шаблонного типа, есть больше информации относительно типа возвращаемого объекта, т.к. в данном случае определенно известно, что возвращаемый объект по крайней мере типа Shape, а не просто типа Object.

Для шаблонов "super" используются другие правила. Здесь все наоборот. Вызовы методов, возвращающих объект неизвестного типа, ограничены и допустимы только при отсутствии каких-либо допущений относительно возвращаемого типа (возвращаемый объект имеет тип Object). Вызовы методов, которым передаются аргументы неизвестного типа, допустимы всегда. Пример вызова:

```
void f(List<? super Triangle>
  list) {
  Triangle ret = list.get(0);
    // error: must not invoke
    // methods that return the
    // "? super Triangle" type
  Object   obj = list.get(0);
    // fine
}
```

Компилятор не в состоянии определить, действительно ли ссылка типа List<? super Triangle> ссылается на список треугольников, как предполагается в первом вызове метода get(). Ссылочная переменная может ссылаться на список Shapes и может подразумеваться смесь различных типов фигур. Таким образом, возвращаемое методом get() может быть или окружностью или прямоугольником, но не обязательно треугольником. Обращение к get() допустимо, только если рассматривать возвращаемое значение как Object, не строя предположений относительно его типа.

Добавление элементов в список допустимо, но компилятору должна быть доступна информация о том, что элементы находятся в пределах установленных границ:

```
void f(List<? super Triangle>
  list) {
  list.add(new Triangle());
    // fine

  Shape shape = new Triangle();
  list.add(shape);  // error:
    // argument must be of type
    // Triangle
}
```

Можно добавить треугольник в список, но только передавая его методу add() посредством ссылки шаблона нижней границы, а именно типа Triangle в нашем примере. Если применить ссылочную переменную супертипа от Triangle к методу add()—например, ссылку типа Shape—то компилятор не сможет узнать, относится ли ссылка Shape к Triangle или же к Circle или другому подтипу Shape. По этой причине вызов add() был отвергнут, поскольку аргумент не принадлежит к типу, заданному как нижняя граница шаблона.

Теперь более детально рассмотрим виды шаблона и различные ситуации, при которых оптимально использовать тот или иной вид шаблона.

Глава 2. Расширение границ диапазона аргументов

Рассмотрим три вида шаблонов J2SE 1.5 в качестве аргументов параметризованных типов, а также исследуем ограничения по их применению. Типичный пример применения шаблонов был представлен в предыдущей главе, где мы обсуждали метод drawAllShapes(). Если мы хотим заставить метод drawAllShapes() работать не только с коллекциями фигур, но и с коллекциями подтипов Shape, то можем достичь желаемого результата, объявляя аргумент как Collection<? extends Shape>.

```
public void drawAllShapes(
  Collection<? extends Shape>
  c) {
  for (Shape s : c) {
    s.draw();
  }
}

Collection<Triangle> c =
  new LinkedList<Triangle>();
... fill the collection with
  triangles ...
drawAllShapes(c);
```

В сущности, шаблон здесь применен для “ослабления” требований, накладываемых на аргументы метода, и для расширения границ диапазона типов аргумента.

Программисты, знакомые с параметризованными методами, могут заметить, что такого же эффекта можно достичь и без шаблонов: объявляя drawAllShapes() как параметризованный метод, мы тем самым разрешили бы использование коллекций подтипов Shape в качестве аргументов. Параметризация представлялась бы таким образом:

```
public <T extends Shape> void
  drawAllShapes(
  Collection<T> c) {
  for (Shape s : c) {
    s.draw();
  }
}
```

Параметризованному методу drawAllShapes() передаются коллекции, тип элементов которых является подтипом Shape, то же происходит и в другой, непараметризованной версии на основе шаблона. Различие состоит в том, что в параметризованной версии не ограничен доступ к аргументу метода, в то время как реализация с шаблоном не позволяет вызывать методы, принимающие аргументы неизвестного типа. В данном случае это несущественно, поскольку вызывается единственный метод draw (), который не принимает никаких аргументов. В другом контексте параметризованный метод является более гибким, чем непараметризованный с шаблонным аргументом. В общем случае шаблоны с верхней границей могут быть замещены параметризацией с параметром ограниченного типа.

2.1 Использование неограниченных шаблонов

Неограниченные шаблоны используются, когда тип параметра типа не имеет значения, поскольку или объект, на который указывает ссылка, не используется, или ни один метод этого типа не вызывается. Примером использования неограниченного шаблона может служить метод, выводящий все элементы коллекции:

```
void printAll(Collection<?> c) {
  for (Object o : c)
  System.out.println(
    o.toString());
}
```

Все, что здесь требуется от элементов коллекции – они должны иметь метод toString(). Поскольку метод toString() определен в классе Object, это требование несущественно.

Собственно, ничего и не требуется знать о типе содержащихся в коллекции объектов. Следовательно, такие объявления как Collection<?> вполне допустимы.

Отметим разницу между использованием Collection<?> и Collection<Object>: метод, объявленный как void printAll(Collection<Object> c), не принял бы аргумент LinkedList<String>, тогда как для метода, объявленного как void printAll(Collection<?> c), это приемлемо. Оба выражения ассоциируются с понятием "коллекция, которая содержит все". Различие в том, что Collection<Object> является конкретным типом, тогда как Collection<?> типом не является, а только обозначает представителей семейства типов – экземпляров Collection. Следовательно, параметр метода, объявленный как Collection<Object>, должен быть именно этого типа или его подтипа. Параметр метода, объявленный как Collection<?>, может быть любого типа, являющегося экземпляром Collection, или подтипом любого из таких типов.

Тип Collection<?> отличается также от “простого” типа Collection. Вызов методов, приводящий к появлению сообщений об ошибках, если он осуществляется посредством ссылки на тип Collection<?>, привел бы к "unchecked warning" в случае вызова посредством ссылки на тип Collection. Последний является конкретным типом, от которого могут быть созданы объекты и массивы, в то время как Collection<?> не должно появляться в выражениях new.

Неограниченные шаблоны используются также для частичных реализаций. Такие шаблоны являются выражением того, что никакие ограничения не накладываются на тип. Рассмотрим такой пример: предположим, что есть параметризованный класс Pair с двумя полями неизвестных и разных типов:

```
class Pair<S,T> {
  private S first;
  private T second;
  ...
  <A extends S> void setFirst(
    Pair<A,?> p) {
    first = p.first;
  }
  ...
}
```

В параметризованном классе Pair<S,T> есть два поля типа S и T соответственно. Объявление параметра метода setFirst() как Pair<A,?> означает, что для метода не имеет значения второе поле пары. Тип Pair<A,?> обозначает частичную реализацию параметризованного типа Pair.

2.2 Использование шаблонов с нижней и верхней границей.

Такие шаблоны используются с той же целью, что и шаблоны с верхней границей; они применяются как типы методов аргументов, “смягчая” требования, накладываемые на тип аргумента. Рассмотрим в качестве примера иерархию классов с разными типами значений:

```
class Person {
  private String firstName;
  private String lastName;
  private Date   dateOfBirth;
  ...
}
class Employee extends Person {
  private Person Supervisor;
  ...
}
```

Если классы такой иерархии предназначены для реализации параметризованного интерфейса, такого как Comparable, то проблема состоит в том, что: супер- и подклассы не могут реализовать разные варианты одного и того же параметризованного интерфейса; хотя это семантически и оправданно. Возможно, мы хотели бы сделать (но это не разрешено) следующее:

```
class Person implements
  Comparable<Person> {
  private String firstName;
  private String lastName;
```

```
  private Date   dateOfBirth;
  ...
  public int compareTo(
    Person other) {
    ...
  }
}
class Employee extends Person
  implements
  Comparable<Employee> {
    // error: already implements
    // Comparable
  private Person Supervisor;
  ...
  public int compareTo(
    Employee other) {
    ...
  }
}
```

Компилятор отвергнет объявление подкласса, поскольку возникли бы различные реализации одного и того же интерфейса, а именно Comparable<Person> и Comparable<Employee>, что недопустимо. Ограничения происходят оттого, что параметризованные типы и методы транслируются посредством удаления типов. В процессе удаления типов две реализации параметризованного интерфейса отображаются в простой тип, при этом стирается разница между интерфейсами, реализуемыми супер- и подклассами.

В результате, суперкласс определяет, какая реализация параметризованного интерфейса используется для всех его подклассов. Это, с другой стороны, делает невозможной передачу подкласса определенным методам, если методы не используют шаблоны с нижними границами. Ниже приводится пример такого метода, но сначала нужно установить иерархию наших классов так, чтобы все классы в иерархии реализовывали Comparable<Person>:

```
class Person implements
  Comparable<Person> {
  private String firstName;
  private String lastName;
  private Date   dateOfBirth;
  ...
  public int compareTo(
    Person other) {
    ...
  }
}
class Employee extends Person {
  private Person Supervisor;
  ...
  public int compareTo(
    Person other) {
    ...
  }
}
```

Рассмотрим метод, который запрашивает объекты Comparable:

```
<T extends Comparable<T>> void
  someMethod(T arg1, T arg2) {
  ... arg1.compareTo(arg2) ...
}
```

Этот метод может быть вызван со ссылочными переменными типа Person в качестве аргументов, но аргумент типа Employee не будет принят, поскольку Employee сопоставим с Person, но не с Employee. Иначе говоря, Employee не оказался бы в пределах границ. В качестве иллюстрации приведен фрагмент кода:

```
Person p1 = new Employee(
  "Peter", "Miller",
```

```
  new Date(4,3,69));
Person p2 = new Employee(
  "Jim", "Anderson",
  new Date(10,8,84));

someMethod(p1, p2);  // fine

Employee e1 = new Employee(
  "Peter", "Miller",
  new Date(4,3,69));
Employee e2 = new Employee(
  "Jim", "Anderson",
  new Date(10,8,84));

someMethod(e1, e2);  // error:
  // arguments are not within
  // bounds
```

Как видим, существенным является то, что методы, принимающие параметры посредством параметризованного типа интерфейса, объявляются так, что на входе - шаблонные экземпляры параметризованного интерфейса с верхними границами, вместо конкретных экземпляров. В противном случае, применимость метода была бы существенно— и безосновательно— ограничена.

Попробуем ослабить сигнатуру метода. Если мы объявляем метод, используя шаблонную реализацию с верхними границами, вместо конкретной реализации, метод принимает в качестве аргументов объекты супер- и подкласса, то есть объекты Person *и* Employee допустимы:

```
<T extends
  Comparable<? super T>> void
  someMethod(T arg1, T arg2) {
  ... arg1.compareTo(arg2) ...
}
```

Этот пример прекрасно иллюстрирует необходимость в шаблонах с нижними границами. Они помогают решить задачу, которая иначе оказалась бы неразрешимой. Классы в иерархии классов не могут реализовывать различные инициализации одного и того же параметризованного интерфейса; они должны реализовывать один и тот же параметризованный интерфейс, а именно тот, который решено реализовывать для самого верхнего суперкласса. Методы, принимающие аргументы, которые должны осуществлять конкретную реализацию параметризованного интерфейсного типа, могут принимать только объекты суперкласса. С использованием шаблонных реализаций с нижними границами эти методы гораздо полезнее.

Заметим также, что шаблоны с нижней границей, в отличие от шаблонов с верхней границей, нельзя заменить параметризацией самого метода, поскольку переменные типа могут иметь только верхние границы, но не нижние. Для шаблонов существуют оба варианта.

Хотя шаблоны с верхними границами и решают проблему, описанную выше, все же остаются случаи, когда даже шаблоны не помогают. Если в интерфейсе объявляется метод, который возвращает значение типа шаблона вместо того, чтобы брать его в качестве аргумента, то тогда мы будем ограничены правилом, согласно которому методы, возвращающие значения неизвестного типа, не могут быть вызваны посредством ссылки на шаблон с нижней границей. В следующем примере используется параметризованный интерфейс Copyable, который имеет метод copy(), возвращающий значение, а также иерархию реализующих интерфейс классов:

```
interface Copyable<T> {
  T copy();
}
class Person implements
  Copyable<Person> {
  ...
  public Person copy() { ... }
}
class Employee extends Person {
...
  public Employee copy() { ... }
```

```
}
```

Заметим, что мы получили преимущество от использования ковариантных возвращаемых значений. Это еще одно новое языковое свойство Java 1.5. Метод copy() подкласса имеет возвращаемый тип, отличающийся от возвращаемого типа метода copy() суперкласса. Разница состоит в том, что метод подкласса возвращает подтип, тогда как метод суперкласса возвращает супертип, что допустимо в Java 1.5. Несмотря на различие возвращаемых типов метод copy() подкласса переписывает метод copy() суперкласса.

Как было показано ранее, метод, подобный нижеприведенному, сомнителен:

```
<T extends Copyable<T>> T
  produceCopy(T arg) {
  return arg.copy();
}
```

Этот метод не принимает Employees в качестве аргументов:

```
Person p1 = new Employee(
  "Peter", "Miller",
  new Date(4,3,69));

Person p2 = produceCopy(p1);
  // fine

Employee e1 = new Employee(
  "Peter", "Miller",
  new Date(4,3,69));

Employee e2 = produceCopy(e1);
  // error:  arguments are not
  // within bounds
```

Использование шаблона с нижней границей теоретически допускало бы Employees в качестве аргументов, но не позволяло бы вызывать метод copy() из метода produceCopy():

```
<T extends Copyable<? super T>>
  T produceCopy(T arg) {
  return arg.copy();  // error:
  // type "? super T"
  // incompatible with type T
}
```

Возвращаемое из метода copy() значение было бы типа "? super T", т.е. оно могло бы быть любым супертипом T. Произвольные супертипы от T не обязательно совместимы по присваиванию с объявленным возвращаемым типом T, и по этой причине “недовольство” компилятора вполне обосновано.

3. Заключение

Итак, подведем итоги, шаблоны являются частью расширения языка, известного как Java Generics, и дополняют языковые свойства Java 1.5. Шаблоны могут использоваться как аргументы параметризованного типа и представлять членов семейства типов. Существуют три различных вида шаблонов: неограниченные, с нижней границей и с верхней границей.

Неограниченные шаблоны используются, когда тип параметра типа не имеет значения, поскольку или объект, на который указывает ссылка, не используется, или ни один метод этого типа не вызывается и для частичных реализаций. Шаблоны с нижней и с верхней границей в основном применяются как типы методов аргументов, “смягчая” требования, накладываемые на тип аргумента.

Шаблонные экземпляры параметризованных типов обычно используются в качестве параметров и возвращаемых типов в объявлениях методов и допускают более широкий диапазон аргументов, чем это было бы возможно без их использования.

Статью подготовили Рувинская Виктория и Беркович Леонид