Модуль 2. Объектно-ориентированное программирование

Тема* 2.8. Параметризованные типы

Оглавление

Тема 2.8. Параметризованные типы

2.8.1. Тип Generic

Начиная с JDK 1.5, в Java появляются новые возможности для программирования. Одним из таких нововведений являются Generics [1]. Generics являются аналогией с конструкцией "Шаблонов"(template) в C++, но имеет свои нюансы. Generics позволяют абстрагировать множество типов [2-3]. Наиболее распространенными примерами являются Коллекции.

Вот типичное использование такого рода (без Generics):

```
1. List myIntList = new LinkedList();
2. myIntList.add(new Integer(0));
3. Integer x = (Integer) myIntList.iterator().next();
```

Как правило, программист знает, какие данные должны быть в List'е. Тем не менее, стоит обратить особое внимание на Приведение типа (**cast**) в строчке 3. Компилятор может лишь гарантировать, что метод next() вернёт Object, но чтобы обеспечить присвоение переменной типа Integer правильным и безопасным, требуется cast. Cast не только создает беспорядки, но дает возможность появление runtime error из-за невнимательности программиста, который необходимо каким-то образом обрабатывать.

И появляется такой вопрос: "Как с этим бороться? " В частности: "Как же зарезервировать List для определенного типа данных?"

Как раз такую проблему решают Generics.

```
1. List<Integer> myIntList = new LinkedList<Integer>();
2. myIntList.add(new Integer(0));
3. Integer x = myIntList.iterator().next();
```

Обратите внимание на объявления типа для переменной myIntList. Он указывает на то, что это не просто произвольный List, а List<Integer>. Мы говорим, что List является generic-интерфейсом, который принимает параметр типа - в этом случае, Integer. Кроме того, необходимо обратить внимание на то, что теперь cast выполняется в строчке 3 автоматически.

Вместо приведения к Integer в строчке 3, у нас теперь есть Integer в качестве параметра в строчке 1. Здесь существенное отличие. Теперь компилятор может проверить этот тип на корректность во время компиляции.

И когда мы говорим, что myIntList объявлен как List<Integer>, это будет справедливо во всем коде и компилятор это гарантирует.

Эффект от Generics особенно проявляется в крупных проектах: он улучшает читаемость и надежность кода в целом.

Свойства Generics:

- Строгая типизация
- Единая реализация
- Ранне обнаружение ошибок на этапе компиляции

```
public interface List<E> {

       E get(int i);
       set(int i, E e);
       add(E e);
       Iterator<E> iterator();
       …
}
```

Для того чтобы использовать класс как Generics, мы должны прописать после имени класса **<...>**, куда можно подставить любое имя, wildcard и т.д.

После того как было объявлено имя generic-типа его можно использовать как обычный тип внутри метода. И когда в коде будет объявлен, к примеру, List<Integer>, то E станет Integer для переменной list (как показано выше).

2.8.2. Несовместимость generic-типов

Это одна из самых важных вещей, которую вы должны узнать о Generics

Как говориться: "В бочке мёда есть ложка дегтя". Для того чтобы сохранить целостности и независимости друг от друга Коллекции, у Generics существует так называемая "Несовместимость generic-типов" [1-3].

Суть такова:

Пусть у нас есть тип Foo, который является подтипом Bar, и еще G - наследник Коллекции, то G<Foo> **не является наследником** G<Bar> [3].

Пример:

```
      List<Integer> li = new ArrayList<Integer>();
       List<Object> lo = li;
       //Иначе – ошибки
        lo.add(“hello”);
       // ClassCastException: String -> int
       Integer li = lo.get(0);
```

2.8.3. Проблемы реализации Generics

Решение 1 - Wildcard

Пусть мы захотели написать метод, который берет Collection<Object> и выводит на экран. И мы захотели вызвать dump для Integer.

Проблема:

```
      void dump(Collection<Object> c) {
               for (Iterator<Object> i = c.iterator(); i.hasNext(); ) {
                  Object o = i.next();
                  System.out.println(o);
```

```
            }
        }
        List<Object> l; dump(l);
        List<Integer> l; dump(l); // Ошибка
```

В этом примере List<Integer> не может использовать метод dump, так как он не является подтипом List<Object>.

Проблема в том что эта реализация кода не эффективна, так как Collection<Object> не является полностью родительской коллекцией всех остальных коллекции, т.е. Collection<Object> имеет ограничения.

Для решения этой проблемы используется wildcard ("?"). Он не имеет ограничения в использовании (то есть имеет соответствие с любым типом) и в этом его плюсы. И теперь, мы можем назвать это с любым типом коллекции.

Решение:

```
    void dump(Collection<?> c) {
        for (Iterator<?> i = c.iterator(); i.hasNext();) {
            Object o = i.next();
            System.out.println(o);
        }
    }
```

Решение 2 – Bounded Wildcard

Пусть мы захотели написать метод, который рисует List<Shape>. И у Shape есть наследник Circle. И мы хотим вызвать draw для Circle.

Проблема:

```
    void draw(List<Shape> c) {
        for (Iterator<Shape> i = c.iterator(); i.hasNext(); ) {
            Shape s = i.next();
            s.draw();
        }
    }
    List<Shape> l; draw(l);
    List<Circle> l; draw(l); // Ошибка
```

Проблема в том, что у нас не получиться из-за несовместимости типов. Предложенное решение используется, если метод который нужно реализовать использовал бы определенный тип и его подтипов. Так называемое "Ограничение сверху".

Для этого нужно вместо <Shape> прописать <? extends Shape>.

Решение

```
void draw(List<? extends Shape> c) {
        for (Iterator<? extends Shape> i = c.iterator();
                i.hasNext(); ) {
            Shape s = i.next();
            s.draw();
        }
    }
```

Решение 3 – Generic-Метод

Пусть вы захотели сделать метод, который берет массив Object и переносить их в коллекцию.

Проблема:

```
void addAll(Object[] a, Collection<?> c) {
   for (int i = 0; i < a.length; i++) {
                c.add(a[i]);
       }
}
addAll(new String[10], new ArrayList<String>());
addAll(new Object[10], new ArrayList<Object>());
addAll(new Object[10], new ArrayList<String>()); // Ошибка
addAll(new String[10], new ArrayList<Object>()); // Ошибка
```

Напомним, что вы не можете просто поместить Object в коллекции неизвестного типа. Способ решения этой проблемы является использование "Generic-Метод" Для этого перед методом нужно объявить <T> и использовать его.

Решение:

```
<T> void addAll(T[] a, Collection<T> c) {
          for (int i = 0; i < a.length; i++) {
              c.add(a[i]);
          }
}
```

Но все равно после выполнение останется ошибка в третьей строчке :

```
addAll(new Object[10], new ArrayList<String>()); // Ошибка
```

Решение 4 – Bounded type argument

Реализуем метод копирование из одной коллекции в другую

Проблема:

```
<M> void addAll(Collection<M> c, Collection<M> c2) {
      for (Iterator<M> i = c.iterator(); i.hasNext(); ) {
            M o = i.next();
            c2.add(o);
      }
}
addAll(new AL<Integer>(), new AL<Integer>());
addAll(new AL<Integer>(), new AL<Object>()); //Ошибка
```

Проблема в том что две Коллекции могут быть разных типов (несовместимость generic-типов). Для таких случаев было придуман Bounded type argument [3]. Он нужен если метод ,который мы пишем использовал бы определенный тип данных. Для этого нужно ввести <N extends M> (N принимает только значения M). Также можно корректно писать <T extends A & B & C>. (Принимает значения нескольких переменных)

Решение:

```
<M, N extends M> void addAll(Collection<N> c, Collection<M> c2) {
      for (Iterator<N> i = c.iterator(); i.hasNext(); ) {
            N o = i.next();
            c2.add(o);
      }
}
```

Решение 5 – Lower bounded wcard

Реализуем метод нахождение максимума в коллекции.

Проблема:

```
<T extends Comparable<T>>
  T max(Collection<T> c) {
   …
  }
List<Integer> il; Integer I = max(il);
class Test implements Comparable<Object> {…}
List<Test> tl; Test t = max(tl); // Ошибка
```

<T extends Comparable<T>> обозначает что T обязан реализовывать интерфейс Comparable<T>.

Ошибка возникает из за того что Test реализует интерфейс Comparable<Object>. Решение этой проблемы - Lower bounded wcard("Ограничение снизу"). Суть в том что мы будет реализовывать метод не только для T, но и для его Супер-типов(Родительских типов). Например: Если мы напишем

`List<T super Integer> list;`

Мы можем заполнить его List<Integer>, List<Number> или List<Object>.

Решение:

```
<T extends Comparable<? super T>>
T max(Collection<T> c) {
      …
}
```

Решение 6 – Wildcard Capture

Реализуем метод Swap в List<?>

Проблема:

```
void swap(List<?> list, int i, int j) {
      list.set(i, list.get(j)); // Ошибка
}
```

Проблема в том, что метод List.set() не может работать с List<?>, так как ему не известно как он List. Для решение этой проблемы используют "Wildcard Capture" (или "Capture helpers"). Суть

заключается в том, чтобы обмануть компилятор. Напишем еще один метод с параметризованной переменной и будем его использовать внутри нашего метода.

Решение:

```
void swap(List<?> list, int i, int j) {
      swapImpl(list, i, j);
}
<T> void swapImpl(List<T> list, int i, int j) {
      T temp = list.get(i);
      list.set(i, list.get(j));
      list.set(j, temp);
```

2.8.4. Ограничения Generic

Также нужно запомнить простые правила для работы с Generics.

- Невозможно создать массив параметра типа:

```
T[] ta =   new T[10]; // Ошибка !!
```

2.8.5. Преобразование типов

В Generics также можно манипулировать с информацией, хранящийся в переменных.

- Уничтожение информации о типе:

```
List l = new ArrayList<String>();
```

- Добавление информации о типе:

```
List<String> l = (List<String>) new ArrayList();
List<String> l1 = new ArrayList();
```

Примеры кода

Первый пример:

```
List<String> ls;
List<Integer> li;
ls.getClass() == li.getClass() // True
ls instanceof List // True
ls instanceof List<String> // Запрещено
```

Второй пример: нахождение максимума в Коллекции Integer.

- Без Generics:

```
Collection c;
Iterator i = c.iterator();
Integer max = (Integer) i.next();
while(i.hasNext()) {
      Integer next = (Integer) i.next();
```

```
        if (next.compareTo(max) > 0) {
         max = next;
        }
   }
```

- С помощью Generics

```
   Collection<Integer> c;
   Iterator<Integer> i = c.iterator();
   Integer max = i.next();
   while(i.hasNext()) {
        Integer next = i.next();
        if (next.compareTo(max) > 0) {
             max = next;
        }
   }
```

Упражнение 2.8.1

Решения с подробными комментариями можно найти в виде проекта в учебном курсе.

1. Давайте создадим несколько вспомогательных классов и интерфейсов, чтобы лучше понять смысл дженериков и как их правильно использовать. Создадим базовый абстрактный класс Animal, от которого унаследуем несколько не абстрактных животных - Cat, Dog и т.д.. У всех животных есть кличка, поэтому введем поле name в базовом классе.

```
public abstract class Animal {
   private String name;
   public Animal(String name) {
       this.name = name;
   }
   public String getName() {
       return name;
   }
}
```

Многие животные умеют произносить какие-либо звуки, поэтому введем интерефейс Say с единственным методом. Скорей всего все собаки умеют гавкать, в отличие от котов. Поэтому унаследуем интерфейс Barking от Say, который не будет вводить новых функций и нужен лишь для группировки классов.

```
public interface Say {
   String say();
}

public interface Barking extends Say {
}
```

Для целей урока введем еще два класса, которые будут наследниками класса Dog: BigDog и SmallDog

```
public class Cat extends Animal implements Say {
    public Cat(String name) {
        super(name);
    }
    @Override
    public String say() {
        return "mow-wow";
    }
}

public class Dog extends Animal implements Barking {
    public Dog(String name) {
        super(name);
    }
    @Override
    public String say() {
        return "dow-wow";
    }
}

public class SmallDog extends Dog implements Barking {
    public SmallDog(String name) {
        super(name);
    }
    @Override
    public String say() {
        return "gav-gav";
    }
}

public class BigDog extends Dog implements Barking {
    public BigDog(String name) {
        super(name);
    }
    @Override
    public String say() {
        return "GAV-GAV";
    }
}
```

2. Создайте обобщенную коллекцию, которая хранит только собачек. Попробуйте добавить в эту коллекцию котика.

Пояснение. Показываем, что нами созданные классы также можно использовать в дженериках и коллекции также становятся type-safe. Также отмечаем, что в коллекцию можно добавить всех потомков, указанного в скобках <> класса.

Мы хотим создать коллекию, которая будет хранить только собак. Без использования обообщенных типов, нам бы пришлось самостоятельно проверять при добавлении каждого объекта в коллекцию на принадлежность его к классу или, что еще хуже, довериться программисту и поверить, что он не ошибется.

```
List dogs = new ArrayList();
Dog someDog = new Dog("D-O-G");
Dog fluffy = new Dog("Fluffy");
```

```
Cat tomas = new Cat("Tomas");
dogs.add(fluffy);
dogs.add(someDog);
dogs.add(tomas);// компилятор никак не проверяет, что здесь присутсвует логическая ошибка
```

Компилятор никак не проверяет, что здесь пристуствует логическая ошибка и более того, если мы попытаемся получить объект из коллекции и привести его к нужному типу, то мы получим ошибку времени исполнения.

```
Cat tomas = (Cat)dogs.get(2);// никаких ошибок компиляции. Но приложение упадет при выполнении этой строчки
```

Начиная с JDK 1.5 на помощь программисту появляются обобщенные типы или дженерики, которые являются type-safe и позволяют избегать таких ошибок еще на этапе компиляции. Создадим такую коллекцию, которая будет хранить только собак и добавим в нее несколько экземпляров класса Dog:

```
List<Dog> dogs = new ArrayList<>();
Dog someDog = new Dog("D-O-G");
Dog fluffy = new Dog("Fluffy");
Cat tomas = new Cat("Tomas");
dogs.add(fluffy);
dogs.add(someDog);
//dogs.add(tomas); - Ошибка компиляции
```

Поскольку Томас не собака, то и в коллекцию он попасть не может. Но если нам понадобится расширить возможности нашей коллекции, чтобы хранить всех животных, мы можем указать компилятору хранить в ней любых животных (Animal). Здесь работают все правила наследования. Создадим такую коллецию и дополнительно несколько больших и маленьких собак, которые через класс Dog также являются потомками класса Animal. Они имеют полное право находиться в этой коллекции в отличии от непонятного “cat”, который является объектом класса String.

```
List<Animal> animals = new ArrayList<>();
BigDog spike = new BigDog("Spike");
BigDog lessy = new BigDog("Lessy");
SmallDog gimmy = new SmallDog("Gimmy");
animals.add(fluffy);
animals.add(tomas);
animals.add(someDog);
animals.add(lessy);
animals.add(spike);
animals.add(gimmy);
//animals.add("cat");// - Ошибка компиляции
//Поскольку все они животные, то в цикле мы можем спросить их клички
for (Animal animal : animals) {
   System.out.println("My name is: " + animal.getName());
}
```

Результат:

```
My name is: Fluffy
My name is: Tomas
My name is: D-O-G
My name is: Lessy
My name is: Spike
My name is: Gimmy
```

3. Создайте обобщенную коллекцию, которая хранит только тех, кто умеет гавкать (interface Barking). Создайте несколько классов, реализующих этот интерфейс (SmallDog, BigDog implements Barking). Создайте функцию, которая принимает коллекцию, состоящую из гавкающих объектов и в цикле прогавкайте все элементы коллекции.

Пояснение: показать, что в скобках <> может быть не только класс, но и интерфейс, что бывает очень удобно при проектировании какой-то сложной и обобщеной системы. Также показываем, что в коллекцию мы не сможем добавить не "гавкающих" животных

Как было отмечено выше для классов, которые выступают параметрами типов в обобщенных классах, сохраняются все правила наследования (но не для самих обобщенных классов, о чем будет показано ниже), поэтому обобщенным типом может выступать и интерфейс. Это бывает очень удобно если вам нужна от классов только их функциональность и вам все равно как она реализована. Мы помним, что все наши собаки умеют гавкать (Barking). Давайте создадим коллекию, только гавкающих объектов и “прогавкаем” их в цикле

```
List<Barking> barkings = new ArrayList<>();
barkings.add(someDog);
barkings.add(spike);
barkings.add(gimmy);
//barkings.add(tomas); // - Ошибка компиляции, Томас не умеет гавкать
for (Barking barking : barkings) {
   System.out.println("I can say: " + barking.say());
}
```

Результат:

```
I can say: dow-wow
I can say: GAV-GAV
I can say: gav-gav
```

4. Создайте обобщенный класс DogHouse, который хранит в своем домике только определенных животных(например только Dog)

Пояснение: показать создание собственных обобщенных классов. Намекнуть на их ограничения в наследовании.

Использование обобщенных коллекций весьма удобно и что более важно - безопасно. Но мы не ограничены только созданными для нас классами, мы также можем создавать обобщенные классы. Давайте создадим такой класс, который будет уметь хранить внутри себя определенный тип данных и выдавать какую-то информацию о хранимых объектах:

```
public class DogHouse<T> {
```

```
    private List<T> list = new ArrayList<>();

    public List<T> getList() {
        return list;
    }

    public void add(T t) {
        list.add(t);
    }

    public int count() {
        return list.size();
    }

    public T findByIndex(int i) {
        return list.get(i);
    }

}
```

Мы можем использовать его и без дженериков, но рискуем получить все ту же проблему как и со стандартными коллекциями

```
DogHouse badDogHouse = new DogHouse();
badDogHouse.add(someDog);
badDogHouse.add(spike);
badDogHouse.add(tomas);//хмм, это не то что мы хотели, мы бы хотели, чтобы при использовании нашего класса в него помещали только собак, но Томас без проблем залез в этот дом
```

Правильным использованием будет явно указать компилятору, что мы хотим хранить здесь именно собак.

```
DogHouse<Dog> niceDogHouse = new DogHouse<>();
niceDogHouse.add(someDog);
niceDogHouse.add(spike);
niceDogHouse.add(gimmy);
//niceDogHouse.add(tomas);//ага, а вот сюда Томасу дорога закрыта
for (Dog dog : niceDogHouse.getList()) {
    System.out.println("I can say from nice dog house: " + dog.getName());
}
```

5. Создайте обобщенный класс, который бы принимал два параметра класса и выводил бы суммарную информацию о них в обобщенном классе. Например, создайте класс ChildDog<SmallDog, BigDog>

Пояснение: конечно же параметров классов может быть больше чем один

Конечно же мы не ограничены количество классов параметров в обобщенном классе. Давайте создадим класс, который будет принимать два параметра класса. И мы очень не хотим, чтобы одним из родителей оказался кот (или вообще что-либо не понятное)

```
public class ChildDog<T1 extends Dog, T2 extends Dog> extends Animal {

    T1 parent1;
    T2 parent2;

    public ChildDog(T1 parent1, T2 parent2) {
        super(parent1.getName() + "&" + parent2.getName());
        this.parent1 = parent1;
        this.parent2 = parent2;
    }
}

ChildDog<SmallDog, BigDog> metis = new ChildDog<>(gimmy, lessy);
//ChildDog<SmallDog, BigDog> metis_oO = new ChildDog<>(gimmy, tomas);// - Ошибка
компиляции
System.out.println("Hi, I'm metis, my name is: " + metis.getName());
```

Получим результат:

```
Hi, I'm metis, my name is: Gimmy&Lessy
```

Как видим мы создали метиса, имя которого является объединением имен его родителей. Мы на уровне компиляции указали, что первым родителем должен быть SmallDog, а вторым BigDog. При попытке подсунуть одним из родителей Томаса, компилятор разумно запрещает это.

6. Создайте элитный собачий домик, который бы не позволял создать домик для котиков

Пояснение: Вводим понятие ограничение сверху extends

В предыдущем примере мы воспользовались ключевым словом extends, чтобы воспользоваться свойством getName параметров класса. Зачастую необходимо ограничить типы, которые может принимать обобщенный класс. Например, мы можем создать DogHouse с параметром типа Cat

```
DogHouse<Cat> niceCatHouse = new DogHouse<>();
niceCatHouse.add(tomas);
//niceCatHouse.add(someDog); // ошибка компиляции
```

Это совсем не то, что мы хотели при проектировании нашего класса. В такой дом Томас спокойно проходит, в то время как ни одна собака не сможет. Но с точки зрения компилятора, никаких ошибок в коде нет. Здесь и приходит на помощь ключевое слово extends. Давайте спроектируем новый дом, в который не сможет пройти ни один кот.

```
public class EliteDogHouse<T extends Dog> {

    private List<T> list = new ArrayList<>();

    public List<T> getList() {
        return list;
    }
```

```
    public void add(T t) {
        list.add(t);
    }

    public int count() {
        return list.size();
    }

    public T findByIndex(int i) {
        return list.get(i);
    }


EliteDogHouse<Dog> eliteDogHouse = new EliteDogHouse<>();
eliteDogHouse.add(spike);
eliteDogHouse.add(gimmy);
eliteDogHouse.add(someDog);
//eliteDogHouse.add(tomas);// - Ошибка компиляции. Нет, Томас, сюда тебе нельзя
//EliteDogHouse<Cat> eliteCatHouse = new EliteDogHouse<>();// - Ошибка компиляции.
Такой домик даже построить не получится
```

Таким способом мы указываем компилятору и всем пользователям нашего класса, что мы ожидаем в качестве параметров классы типа Dog и его наследников. Построить элитный домик с параметром типа Cat не получится.

7. Создайте коллекцию элитных собачьих домиков и выведите информацию о количестве собак в каждом домике в цикле for

Пояснение: Объясняем проблемы наследования дженериков и вводим понятие wildcard - ?

Давайте теперь построим целую аллею элитных собачьих домиков и узнаем, сколько живет собак в каждом таком доме на аллее. Прежде чем строить аллею, надо создать несколько элитных домиков и заселить их собаками.

```
EliteDogHouse<SmallDog> eliteSmallDogHouse = new EliteDogHouse<>();
eliteSmallDogHouse.add(gimmy);
EliteDogHouse<BigDog> eliteBigDogHouse = new EliteDogHouse<>();
eliteBigDogHouse.add(spike);
eliteBigDogHouse.add(lessy);
EliteDogHouse<Dog> eliteAnyDogHouse = new EliteDogHouse<>();
eliteAnyDogHouse.add(spike);
eliteAnyDogHouse.add(fluffy);
DogHouse<Dog> poorDogHouse = new DogHouse<>();
poorDogHouse.add(someDog);
List<EliteDogHouse<Dog>> eliteDogHouses = new ArrayList<>();
eliteDogHouses.add(eliteAnyDogHouse);//OK
//eliteDogHouses.add(poorDogHouse);//конечно же дешевому собачьему домику не место
среди элитных домов
//eliteDogHouses.add(eliteSmallDogHouse);//fail - как так то? eliteSmallDogHouse
является элитным домиком, но мы не можем его добавить сюда!
```

Мы создали 3 элитных дома для больших, маленьких и обычных собак и заселили их питомцами. Также мы создали обыный дом, чтобы показать, что ему не место среди аллеи элитных домов, что

нам учтиво сообщил компилятор при попытке добавить такой дом в типизированную коллекцию. Но компилятор почему-то запретил нам добавлять и элитный домик для маленьких и больших собак, разрешив только добавить элитный домик для обычных собак. Здесь мы встречаемся с ограничением наследования дженериков. Суть его заключается в том, что из того, что SmallDog является наследником класса Dog, не следует, что обобщенный класс EliteDogHouse<SmallDog> будет наследником класса EliteDogHouse<Dog>. Это два абсолютно разных класса, не имеющих никаких родственных связей. Это справедливо и для параметров класса Object. Несмотря на то, что все классы являются потомками этого класса, в дженериках вся информация о родстве теряется. Для решения данной проблемы был введет специальный символ *wildcard* **?**, который говорит компилятору, что мы хотим иметь дело с любым классом. Его также можно ограничить с помощью ключевого слова *extends*, чтобы ограничить множество возможных используемых классов.

```
List<EliteDogHouse<? extends Dog>> beverlyHillsAlley = new ArrayList<>();
beverlyHillsAlley.add(eliteAnyDogHouse);//OK
beverlyHillsAlley.add(eliteSmallDogHouse);//OK
beverlyHillsAlley.add(eliteBigDogHouse);//OK
for (EliteDogHouse house : beverlyHillsAlley) {
   System.out.println("This house contains: " + house.count() + " dogs");
}
```

Получим:

```
This house contains: 2 dogs
This house contains: 1 dogs
This house contains: 2 dogs
```

Список источников

1. http://docs.oracle.com/javase/tutorial/extra/generics/index.html
2. http://onewebsql.com/blog/generics-extends-super
3. http://neerc.ifmo.ru/wiki/index.php?title=Generics

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Боровому Дмитрию Игоревичу.