

# Теория дженериков в Java или как на практике ставить скобки

Viacheslav

## Введение

Начиная с JSE 5.0 в арсенал языка Java были добавлены дженерики.



## Что такое дженерики в Java?

**Дженерики** (обобщения) — это особые средства языка Java для реализации обобщённого программирования: особого подхода к описанию данных и алгоритмов, позволяющего работать с различными типами данных без изменения их описания. На сайте Oracle дженерикам посвящён отдельный tutorial: "[Lesson: Generics](#)".

Во-первых, чтобы понять дженерики, нужно разобраться, зачем они вообще нужны и что они дают. В tutorial в разделе "[Why Use Generics?](#)" сказано, что одно из назначений — более сильная проверка типов во время компиляции и устранение необходимости явного приведения.



Приготовим для опытов любимый [tutorialspoint online java compiler](#). Представим себе такой вот код:

```
import java.util.*;
public class HelloWorld{
    public static void main(String []args){
        List list = new ArrayList();
        list.add("Hello");
        String text = list.get(0) + ", world!";
        System.out.print(text);
    }
}
```

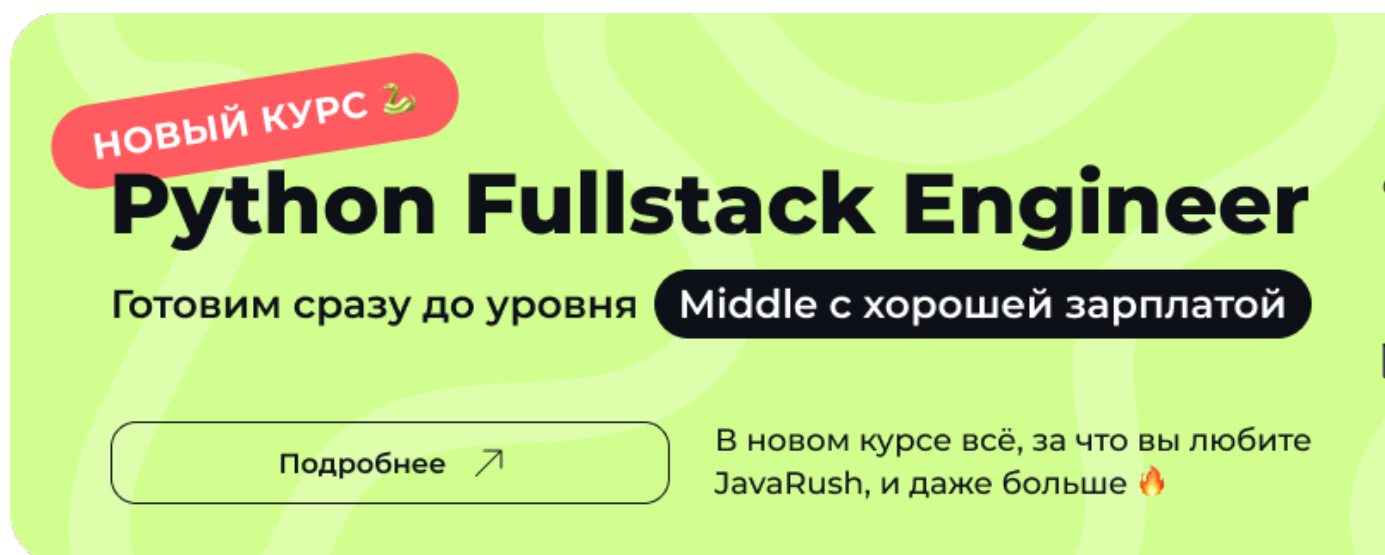
Этот код выполнится хорошо. Но что если к нам пришли и сказали, что фраза "Hello, world!" избита и можно вернуть только Hello? Удалим из кода конкатенацию со строкой ", world!". Казалось бы, что

может быть безобиднее? Но на деле мы получим ошибку **ПРИ КОМПИЛЯЦИИ**: `error: incompatible types: Object cannot be converted to String` Всё дело в том, что в нашем случае `List` хранит список объектов типа `Object`. Так как `String` — наследник для `Object` (ибо все классы неявно наследуются в Java от `Object`), то требует явного приведения, чего мы не сделали. А при конкатенации для объекта будет вызван статический метод `String.valueOf(obj)`, который в итоге вызовет метод `toString` для `Object`. То есть `List` у нас содержит `Object`. Выходит, там где нам нужен конкретный тип, а не `Object`, нам придётся самим делать приведение типов:

```
import java.util.*;
public class HelloWorld{
    public static void main(String []args){
        List list = new ArrayList();
        list.add("Hello!");
        list.add(123);
        for (Object str : list) {
            System.out.println((String)str);
        }
    }
}
```

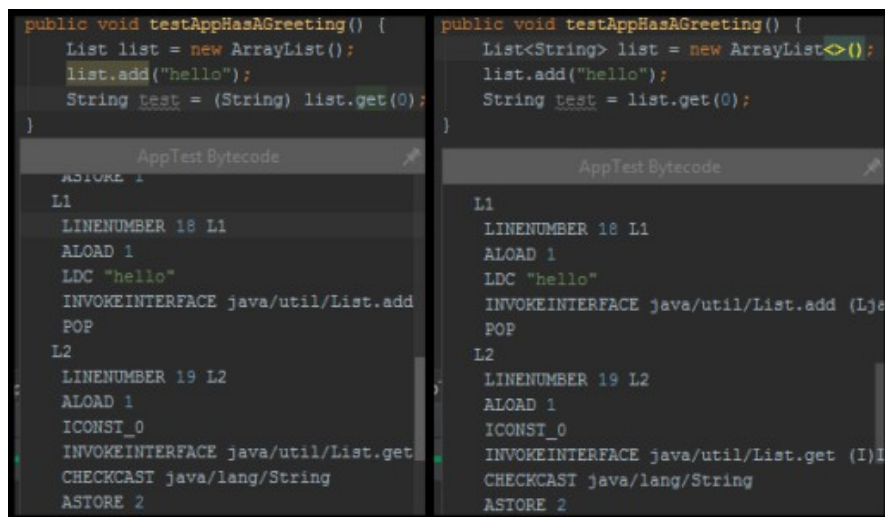
Однако, в данном случае, т.к. `List` принимает список объектов, он хранит не только `String`, но и `Integer`. Но самое плохое, в этом случае компилятор не увидит ничего плохого. И тут мы получим ошибку уже **ВО ВРЕМЯ ВЫПОЛНЕНИЯ** (ещё говорят, что ошибка получена "в Runtime"). Ошибка будет:

`java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String` Согласитесь, не самое приятное. И всё это потому, что компилятор — не искусственный интеллект и он не может угадать всё, что подразумевает программист. Чтобы рассказать компилятору подробнее о своих намерениях, какие типы мы собираемся использовать, в Java SE 5 ввели **дженерики**. Исправим наш вариант, подсказав компилятору, что же мы хотим:

A promotional banner for a course titled "Python Fullstack Engineer". The background is light green with abstract circular patterns. At the top left, a red pill-shaped badge contains the text "НОВЫЙ КУРС" followed by a small green icon of a person. The main title "Python Fullstack Engineer" is in large, bold, black font. Below it, the text "Готовим сразу до уровня" is followed by a dark blue pill-shaped badge containing the text "Middle с хорошей зарплатой". At the bottom left, there is a button with the text "Подробнее" and a right-pointing arrow. At the bottom right, there is a text block: "В новом курсе всё, за что вы любите JavaRush, и даже больше" followed by a small orange flame icon.

```
import java.util.*;
public class HelloWorld {
    public static void main(String []args){
        List<String> list = new ArrayList<>();
        list.add("Hello!");
        list.add(123);
        for (Object str : list) {
            System.out.println(str);
        }
    }
}
```

Как мы видим, нам больше не нужно приведение к `String`. Кроме того, у нас появились угловые скобки (angle brackets), которые обрамляют дженерики. Теперь компилятор не даст скомпилировать класс, пока мы не удалим добавление 123 в список, т.к. это `Integer`. Он нам так и скажет. Многие называют дженерики "синтаксическим сахаром". И они правы, так как дженерики действительно при компиляции станут теми самыми кастами. Посмотрим на байткод скомпилированных классов: с кастом вручную и с использованием дженериков:

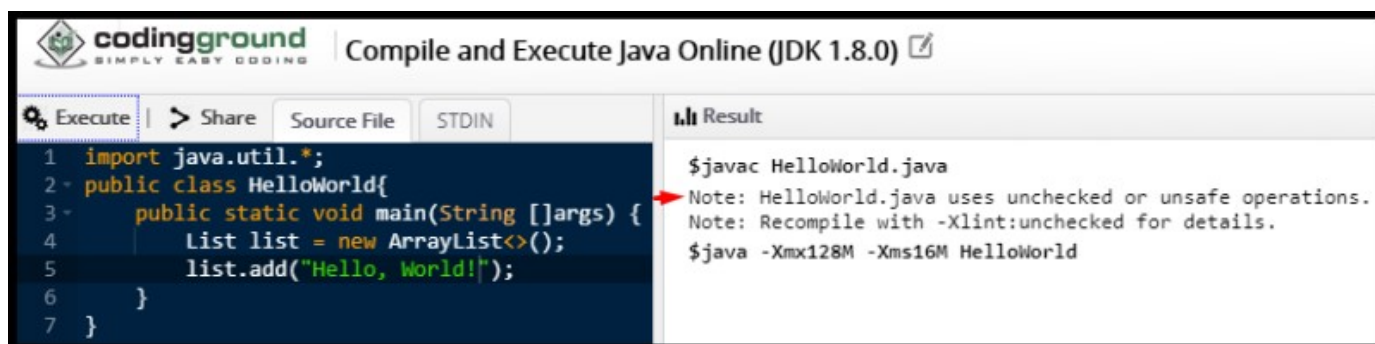


После компиляции какая-либо информация о дженериках стирается. Это называется "Стирание типов" или ["Type Erasure"](#). Стирание типов и дженерики сделаны так, чтобы обеспечить обратную совместимость со старыми версиями JDK, но при этом дать возможность помогать компилятору с определением типа в новых версиях Java.

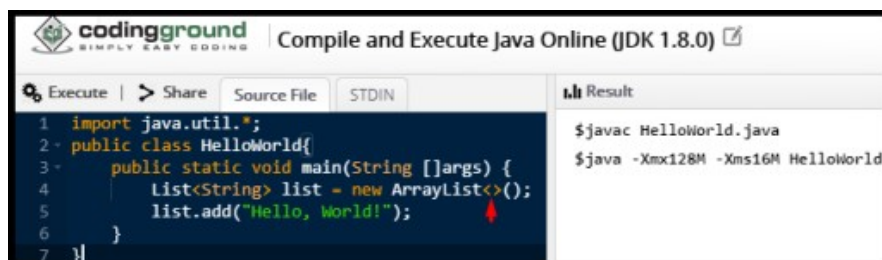


## Raw Types или сырые типы

Говоря о дженериках мы всегда имеем две категории: типизированные типы (Generic Types) и "сырые" типы (Raw Types). **Сырые типы** — это типы без указания "уточнения" в фигурных скобках (angle brackets):



Типизированные типы — наоборот, с указанием "уточнения":



Как мы видим, мы использовали необычную конструкцию, отмеченную стрелкой на скриншоте. Это особый синтаксис, который добавили в Java SE 7, и называется он ["the diamond"](#), что в переводе означает алмаз. Почему? Можно провести аналогию формы алмаза и формы фигурных скобок: <> Также Diamond синтаксис связан с понятием ["Type Inference"](#), или же выводение типов. Ведь компилятор, видя справа <>

смотрит на левую часть, где расположено объявление типа переменной, в которую присваивается значение. И по этой части понимает, каким типом типизируется значение справа. На самом деле, если в левой части указан дженерик, а справа не указан, компилятор сможет вывести тип:

```
import java.util.*;
public class HelloWorld{
    public static void main(String []args) {
        List<String> list = new ArrayList();
        list.add("Hello World");
        String data = list.get(0);
        System.out.println(data);
    }
}
```

Однако это будет смешиванием нового стиля с дженериками и старого стиля без них. И это крайне нежелательно. При компиляции кода выше мы получим сообщение: Note: HelloWorld.java uses unchecked or unsafe operations. На самом деле кажется непонятным, зачем вообще нужен тут diamond добавлять. Но вот пример:

```
import java.util.*;
public class HelloWorld{
    public static void main(String []args) {
        List<String> list = Arrays.asList("Hello", "World");
        List<Integer> data = new ArrayList(list);
        Integer intNumber = data.get(0);
        System.out.println(data);
    }
}
```

Как мы помним, у ArrayList есть и второй конструктор, который принимает на вход коллекцию. И вот тут-то и кроется коварство. Без diamond синтаксиса компилятор не понимает, что его обманывают, а вот с diamond — понимает. Поэтому,

### правило #1

: всегда использовать diamond синтаксис, если мы используем типизированные типы. В противном случае мы рискуем пропустить, где у нас используется raw type. Чтобы избежать предупреждений в логе о том, что "uses unchecked or unsafe operations" можно над используемым методом или классом указать особую аннотацию: @SuppressWarnings("unchecked") Suppress переводится как подавлять, то есть дословно — подавить предупреждения. Но подумайте, почему вы решили её указать? Вспомните о правиле номер один и, возможно, вам нужно добавить типизацию.



## Типизированные методы (Generic Methods)

Дженерики позволяют типизировать методы. Данной возможности в tutorial от Oracle посвящен отдельный раздел: "[Generic Methods](#)". Из данного tutorial важно запомнить про синтаксис:

- включает список типизированных параметров внутри угловых скобок;
- список типизированных параметров идёт до возвращаемого метода.

Посмотрим на пример:

```
import java.util.*;
public class HelloWorld{

    public static class Util {
        public static <T> T getValue(Object obj, Class<T> clazz) {
            return (T) obj;
        }
        public static <T> T getValue(Object obj) {
            return (T) obj;
        }
    }
}
```

```

    }
}

public static void main(String []args) {
    List list = Arrays.asList("Author", "Book");
    for (Object element : list) {
        String data = Util.getValue(element, String.class);
        System.out.println(data);
        System.out.println(Util.<String>getValue(element));
    }
}
}

```

Если посмотреть на класс Util, видим в нём два типизированных метода. Благодаря возможности выведения типов мы можем предоставить определение типа непосредственно компилятору, а можем сами это указать. Оба варианта представлены в примере. Кстати, синтаксис весьма логичен, если подумать. При типизировании метода мы указываем дженерик ДО метода, потому что если мы будем использовать дженерик после метода, Java не сможет понять, какой тип использовать. Поэтому сначала объявляем, что будем использовать дженерик T, а потом уже говорим, что этот дженерик мы собираемся возвращать. Естественно, Util.<Integer>getValue(element, String.class) упадёт с ошибкой incompatible types: Class<String> cannot be converted to Class<Integer>. При использовании типизированных методов стоит всегда помнить про стирание типов. Посмотрим на пример:

```

import java.util.*;
public class HelloWorld {

    public static class Util {
        public static <T> T getValue(Object obj) {
            return (T) obj;
        }
    }

    public static void main(String []args) {
        List list = Arrays.asList(2, 3);
        for (Object element : list) {
            System.out.println(Util.<Integer>getValue(element) + 1);
        }
    }
}

```

Он будет прекрасно работать. Но только до тех пор, пока компилятор будет понимать, что у вызываемого метода тип Integer. Заменяем вывод на консоль на следующую строку:

System.out.println(Util.getValue(element) + 1); И мы получим ошибку: bad operand types for binary operator '+', first type: Object, second type: int То есть произошло стирание типов. Компилятор видит, что тип никто не указал, тип указывается как Object и выполнение кода падает с ошибкой.



## Типизированные классы (Generic Types)

Типизировать можно не только методы, но и сами классы. У Oracle в их гайде этому посвящён раздел "[Generic Types](#)". Рассмотрим пример:

```

public static class SomeType<T> {
    public <E> void test(Collection<E> collection) {
        for (E element : collection) {
            System.out.println(element);
        }
    }
    public void test(List<Integer> collection) {
        for (Integer element : collection) {
            System.out.println(element);
        }
    }
}

```

```
    }  
}
```

Тут всё просто. Если мы используем класс, дженерик указывается после имени класса. Давайте теперь в методе `main` создадим экземпляр этого класса:

```
public static void main(String []args) {  
    SomeType<String> st = new SomeType<>();  
    List<String> list = Arrays.asList("test");  
    st.test(list);  
}
```

Он отработает хорошо. Компилятор видит, что есть `List` из чисел и `Collection` типа `String`. Но что если мы сотрём дженерики и сделаем так:

```
SomeType st = new SomeType();  
List<String> list = Arrays.asList("test");  
st.test(list);
```

Мы получим ошибку: `java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer`. Опять стирание типов. Поскольку у класса больше нет дженерика, компилятор решает: раз мы передали `List`, метод с `List<Integer>` более подходящий. И мы падаем с ошибкой. Поэтому,

**правило #2: Если класс типизирован, всегда указывать тип в дженерике**

.

## Ограничения

К типам, указываемым в дженериках мы можем применить ограничение. Например, мы хотим, чтобы контейнер принимал на вход только `Number`. Данная возможность описана в [Oracle Tutorial](#) в разделе [Bounded Type Parameters](#). Посмотрим на пример:

```
import java.util.*;  
public class HelloWorld{  
  
    public static class NumberContainer<T extends Number> {  
        private T number;  
  
        public NumberContainer(T number) { this.number = number; }  
  
        public void print() {  
            System.out.println(number);  
        }  
    }  
  
    public static void main(String []args) {  
        NumberContainer number1 = new NumberContainer(2L);  
        NumberContainer number2 = new NumberContainer(1);  
        NumberContainer number3 = new NumberContainer("f");  
    }  
}
```



**Получите профессию  
Java-разработчика**

на онлайн-курсе с



Онлайн-лекции  
с опытными  
менторами



Групповое обучение  
и поддержка  
в закрытом чате



10 проектов в вашем  
портфолио и сотни  
часов кодинга

**Скоро стартуют занятия в новой группе – поспешите!**



Как видим, мы ограничили тип дженерика как класс/интерфейс Number и наследники. Интересно, что можно указать не только класс, но и интерфейсы. Например: `public static class NumberContainer<T extends Number & Comparable> {` Ещё у дженериков есть понятие Wildcard <https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html> Они в свою очередь делятся на три типа:

- Upper Bounded Wildcards - `<? extends Number>`
- Unbounded Wildcards - `<?>`
- Lower Bounded Wildcards - `<? super Integer>`

C Wildcard действует так называемый [Get Put principle](#). Можно их выразить в следующем виде:

- Use an `extends` wildcard when you only get values out of a structure
- Use a `super` wildcard when you only put values into a structure.
- And don't use a wildcard when you both want to get and put from/to a

Данный принцип ещё называют принципом PECS (Producer Extends Consumer Super). Подробнее можно прочитать на хабре в статье "[Использование generic wildcards для повышения удобства Java API](#)", а также в отличном обсуждении на stackoverflow: "[Использование wildcard в Generics Java](#)". Вот небольшой пример из исходников Java — метод Collections.copy:

```
* @param <T> the class of the objects in the lists
* @param dest The destination list.
* @param src The source list.
* @throws IndexOutOfBoundsException if the destination list
*         to contain the entire source List.
* @throws UnsupportedOperationException if the destination
*         list-iterator does not support the <tt>set</tt>
*/
public static <T> void copy(List<? super T> dest, List<? ex
```

Ну и небольшой примерчик того, как НЕ будет работать:

```
public static class TestClass {
    public static void print(List<? extends String> list) {
        list.add("Hello World!");
        System.out.println(list.get(0));
    }
}

public static void main(String []args) {
    List<String> list = new ArrayList<>();
    TestClass.print(list);
}
```

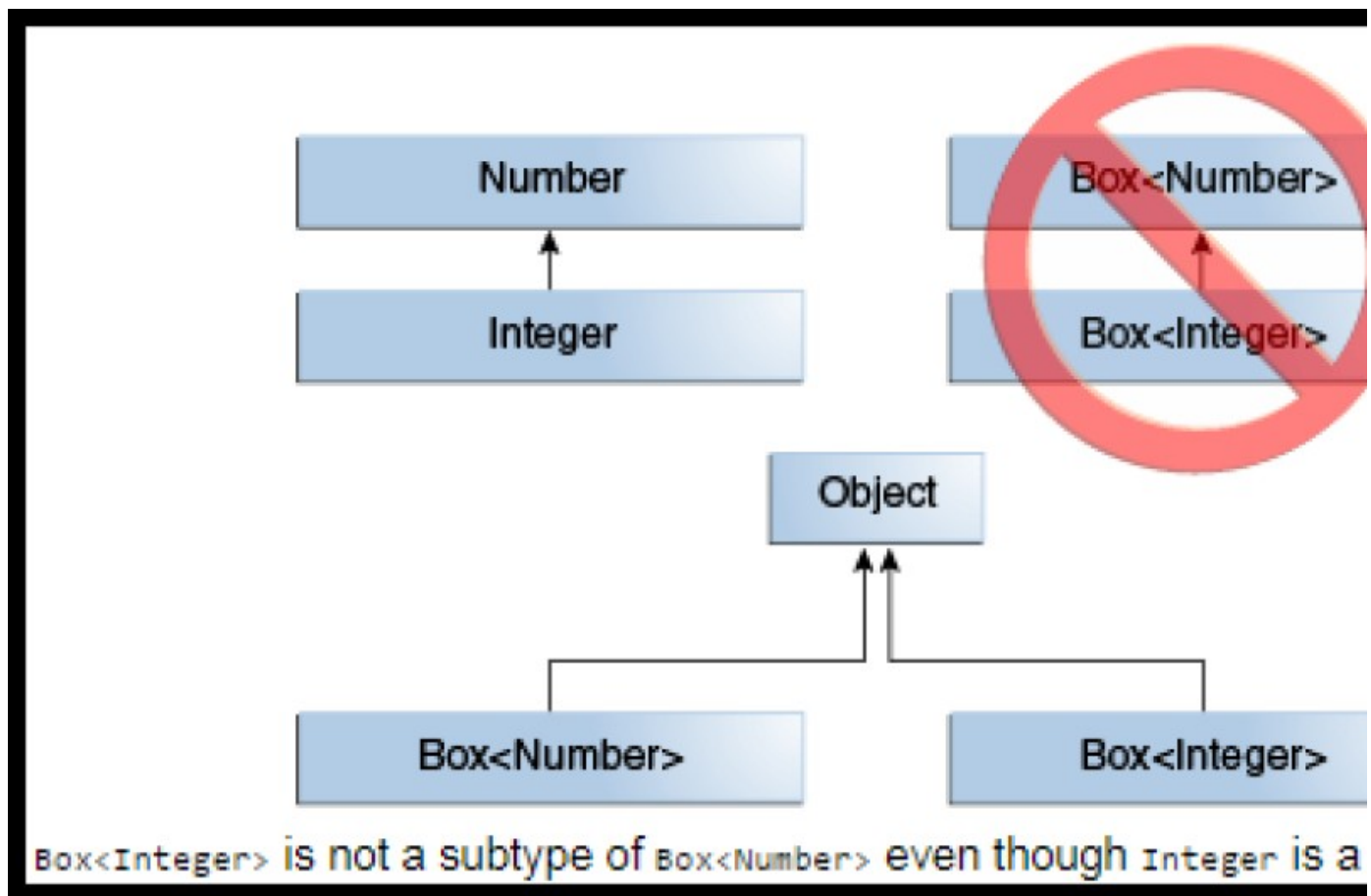
Но если заменить extends на super, всё станет хорошо. Так как мы наполняем список list значением перед выводом, он для нас является потребителем, то есть consumer'ом. Следовательно, используем super.

## Наследование

Есть ещё одна необычная особенность дженериков - это их наследование. Наследование дженериков описано в tutorial от Oracle в разделе "[Generics, Inheritance, and Subtypes](#)". Главное это запомнить и осознать следующее. Мы не можем сделать так:

```
List<CharSequence> list1 = new ArrayList<String>();
```

Потому что наследование работает с дженериками по-другому:



И вот ещё хороший пример, который упадёт с ошибкой:

```
List<String> list1 = new ArrayList<>();
List<Object> list2 = list1;
```

Тут тоже всё просто. `List<String>` не является наследником `List<Object>`, хотя `String` является наследником `Object`.

## Final

Вот мы и освежили в памяти дженерики. Если их редко использовать во всей их мощи, какие-то детали выпадают из памяти. Надеюсь, данный небольшой обзор поможет освежить в памяти. А для большего результата настоятельно рекомендую ознакомиться со следующими материалами:

- Юрий Ткач: [Сырые типы - Generics #1 - Advanced Java](#)
- [Наследование и расширители обобщений - Generics #2 - Advanced Java](#)
- [Рекурсивное расширение типа - Generics #3 - Advanced Java](#)
- [Александр Маторин — Неочевидные Дженерики](#)
- [Введение в Java. Generics. Wildcards | Технострим](#)
- [O'Reilly : Java Generics and Collections](#)

#Viacheslav