

Лекция 8.

Generics

Проблема

- Допустим, мы хотим написать класс для списка на массиве
- И хотим чтобы наш класс был безопасен с точки зрения типов – например, что если у нас список строк, то вставлять в него можно только строки
- С нашими текущими знаниями, нам пришлось бы писать отдельный класс для каждого варианта такого списка – отдельный класс для списка строк, отдельный класс для списка целых чисел и т.д.
- При этом код везде был бы тем же самым
- Очевидно, что так быть не должно

Пример дублирования

- ```
public class StringList {
 private String[] items;
 private int size;

 public boolean add(String item) {
 // реализация
 }
}
```
- ```
public class IntegerList {  
    private Integer[] items;  
    private int size;  
  
    public boolean add(Integer item) {  
        // реализация  
    }  
}
```

Код отличается только
типом элементов в
массиве и некоторых
методах

Решение проблемы

- Было бы хорошо сделать что-то вроде **шаблона класса**, где вместо конкретного типа указан **тип-параметр** (назовем этот параметр **T**)
- И просто можно было бы подставлять нужный тип вместо **T**, и получать новый конкретный класс
- ```
public class List<T> {
 private T[] items;
 private int size;

 public boolean add(T item) {
 // реализация
 }
}
```

# Решение проблемы

- ```
public class List<T> {  
    private T[] items;  
    private int size;  
  
    public boolean add(T item) {  
        // реализация  
    }  
}
```
- В нашем коде создаем уже конкретные реализации списка:
- ```
List<String> stringsList = new List<String>(); // список строк
stringsList.add("text"); // OK
stringsList.add(234); // ошибка компиляции – неверный тип
```

```
List<Integer> intList = new List<Integer>(); // список чисел
```

# Шаблоны и generics

- Такая фича есть в C++, Java, C# и некоторых других языках
- В C++ эта фича называется **шаблоны (templates)**
- В Java и C# эта фича называется **generics**
- Но синтаксис, реализация и возможности этих конструкций в этих языках разные
- \* В примерах на следующих слайдах даже для других языков используется синтаксис Java, чтобы легче была понятна суть

# Шаблоны в C++

- В C++ эта логика реализуется компилятором – компилятор берет ваш шаблон `List<T>` и находит все конкретные места применения этого шаблона (например, `List<String>`)
- Для каждого места использования компилятор генерирует отдельный класс, где вместо типов-параметров подставляются конкретные типы

# Generics в C#

- В C# эта логика реализуется компилятором и средой исполнения
- Во время исполнения если вам первый раз понадобился конкретный **generic** `List<String>`, то среда исполнения «на лету» создает класс-реализацию, где вместо **типа-параметра** `T` будет `String`
- Во всех последующих ситуациях класс `List<String>` уже сгенерирован, и он может просто использоваться



# Generics в Java

- В Java **generic**'и, можно сказать, искусственные – это просто проверки компилятором
- Никакие классы не генерируются
- Просто компилятор проверяет, что вы передаете правильные типы, а в самом деле используется тип **Object**
- Подробнее сейчас рассмотрим

# Generics методы

- **Шаблоны** и **generic**'и могут применяться не только к типам данных, но и к методам
- В C++ и C# для каждого конкретного варианта логика такая же, как для классов
- А в Java также только добавляются проверки компилятора

# Преимущества generic'ов

1. Позволяют избегать дублирования кода и писать обобщенные алгоритмы, которые работают для разных типов
  - Вы пишете **generic** код всего 1 раз, а потом в разных местах используете этот **generic** с разными конкретными типами
2. Предоставляют безопасность типов по сравнению с **raw типами** (т.е. не **generic** типами). Особенно это важно для коллекций

# Где применяются?

- Чаще всего **generics** встречаются именно в коллекциях и других контейнерах
- Также, начиная с Java 1.8, появились **лямбда-функции**, в которых **generics** тоже очень активно используются
- При написании собственных обобщенных алгоритмов и классов

**Generics**

# Generics

- **Generics** позволяют параметризовывать классы и интерфейсы типами данных
- Например:
  - `List<Integer>` - список целых чисел
  - `List<String>` - список строк
- Ссылка на tutorial:  
<http://docs.oracle.com/javase/tutorial/java/generics/index.html>

# Коллекции до Generics

- До generic'ов коллекции в Java были **нетипизированными (untyped / raw)**, т.е. хранили в себе ссылки на **Object**

В коллекцию можно  
вставлять объекты  
любых типов

- Пример:  
`List list = new ArrayList();`  
`list.add(Integer.valueOf(1));`  
`list.add("abc");`

При вытаскивании из  
коллекции, необходимо  
приводить типы

`Integer x = (Integer)list.get(0);`

`// потенциальная ошибка времени исполнения!`

# Недостатки нетипизир. коллекций

1. Нужно выполнять приведение типа каждый раз при вытаскивании элемента из коллекции

```
Integer x = (Integer)list.get(3);
```

2. Отсутствие помощи компилятора при поиске ошибок

```
Integer x = (Integer)list.get(3);
```

```
// исключение – элемент с индексом 3 оказался
// не Integer, а строкой, которую случайно туда
// положили
```



# Коллекции до Generics

- Generics избавили Java от этих недостатков
- `List<Integer> list = new ArrayList<>();`  
`list.add(Integer.valueOf(1));`  
`list.add("abc");` // ошибка компиляции – нельзя!  
  
`Integer x = list.get(0);` // не нужно приведение типа
- Код заметно упростился и стал более безопасным

# Как реализованы Generics?

- В самом деле в Java **generics** существуют только на уровне исходного кода и компилятора
- При компиляции кода в байт-код, информация о **Generic**'ах стирается, а код работает с нетипизированными коллекциями **Object**'ов

# Как реализованы Generics?

- `List<Integer> list = new ArrayList<>();`  
`list.add(Integer.valueOf(1));`  
`Integer x = list.get(0);`
- При компиляции превращается в:  
`List list = new ArrayList();`  
`list.add(Integer.valueOf(1));`  
`Integer x = (Integer)list.get(0);`

# Почему так?

- Это сделано в целях совместимости с предыдущими версиями Java
- За счет такого решения разработчикам Java не пришлось менять байт-код

# Что из этого следует?

- Например, можно присвоить `List<Integer>` переменной типа `List`, и положить в этот список объект, не являющийся `Integer`
- ```
List<Integer> list = new ArrayList<>();  
List untypedList = list;  
untypedList.add("123");  
Integer x = list.get(0);
```

 - // никакой ошибки
 - // ошибка во время
 - // исполнения!
 - // String не приводится к
 - // Integer
- Делать так строго не рекомендуется!

Что из этого следует?

- **Generics** не поддерживают примитивные типы
- Потому что они не являются объектами и не наследуются от **Object**

Как объявить свой generic тип?

- ```
public class Optional<T> {
 private T value;

 public Optional() {
 }

 public Optional(T value) {
 this.value = value;
 }

 public T get() {
 if (value == null) {
 throw new NullPointerException("Value is empty");
 }
 return value;
 }
}
```

Можно использовать  
любую заглавную букву,  
обычно T

Теперь эту букву можно  
использовать в качестве имени  
типа для объявления полей,  
параметров и возвращаемых  
значений функций

# Как объявить свой generic тип?

- ```
public class Pair<T, V> {  
    private T first;  
    private V second;  
  
    // код ...  
}
```

Типов-параметров
может быть несколько,
они указываются через
запятую

- ```
public class Complex<T> {
 private T[] array;
 private List<T> list;
}
```

На основе generic типа  
можно объявлять  
массивы и другие  
generic типы



# Name convention для имен типов

- T – тип
- S, U, V и т.д. – второй, третий и т.д. типы
- Особые случаи:
  - E – элемент коллекции
  - N - число
  - K – ключ (в ассоциативном массиве)
  - V – значение (в ассоциативном массиве)
  - R – тип результата функции

# Generic интерфейсы

- Интерфейсы также могут быть **generic**:
- ```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}
```

Generic интерфейсы

- Можно реализовывать **generic** интерфейсы и наследоваться от **generic** классов:
- ```
public class OrderedPair<K, V> implements Pair<K, V> {
 private K key;
 private V value;


 public OrderedPair(K key, V value) {
 this.key = key;
 this.value = value;
 }

 public K getKey() { return key; }
 public V getValue() { return value; }
}
```

# Generic методы

- Кроме классов и интерфейсов, могут быть **generic** методы
- ```
public class Util {  
    // Статический generic метод  
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {  
        return p1.getKey().equals(p2.getKey()) &&  
            p1.getValue().equals(p2.getValue());  
    }  
}
```

Типы-параметры нужно
указать перед
возвращаемым типом



Сам класс не является
generic

- Вызов метода:

```
Pair<Integer, String> pair1 = new Pair<>(3, "abc");  
Pair<Integer, String> pair2 = new Pair<>(5, "sdfg");  
boolean result = Util.compare(pair1, pair2); // false
```

Невозможность создавать объекты

- Несмотря на то, что можно объявлять переменные и поля **generic** типов, создавать объекты типа-параметра **generic**'а нельзя
- ```
public class MyClass<T> {
 private T t1; // ОК
 private T t2 = new T(); // Ошибка компиляции
}
```
- Так нельзя, потому что по факту **T** заменится на **Object**, а это не то, чего хочется в этом коде

**Ограничения  
на generics**

# Ограничения на generics

- По умолчанию в качестве типа параметра можно использовать любой класс или интерфейс
- Но можно ограничивать эти классы при помощи слова `extends`

# Ограничения на generics

- Например, мы хотим, чтобы наш класс пара принимал только числа в качестве типа-параметра

- ```
public class NumberBox<T extends Number> {  
    private T value;
```

```
    public int getIntValue() {  
        return value.intValue();  
    }  
}
```

Класс Number – это
родительский класс для
числовых оберток
Его наследники – Integer,
Double, Float и т.д.

Можем использовать
методы
ограничивающего класса
Number

Ограничения на generics

- ```
public class NumberBox<T extends Number> {
 private T value;

 public int getIntValue() {
 return value.intValue();
 }
}
```
- Можем передавать в качестве параметра сам ограничивающий тип, либо его наследников
- ```
NumberBox<Integer> b1 = new NumberBox<>(); // OK  
NumberBox<Number> b2 = new NumberBox<>(); // OK  
NumberBox<String> b3 = new NumberBox<>();  
// ошибка компиляции – String не наследник Number
```

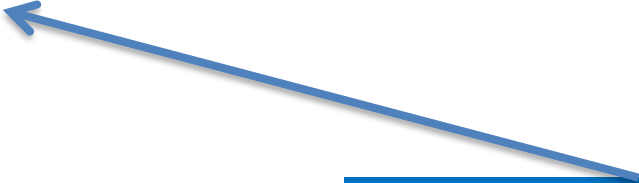
Ограничения на generics

- Аналогично для методов

- ```
public class Util {
 public static <T extends Comparable<T>> int getCountGreater(T[] t, T x) {
 int result = 0;

 for (T element: t) {
 if (element.compareTo(x) > 0) {
 ++result;
 }
 }

 return result;
 }
}
```



Пользуемся методом  
compareTo интерфейса  
Comparable<T>

- Этот код универсален для всех типов, который реализуют интерфейс Comparable<T>

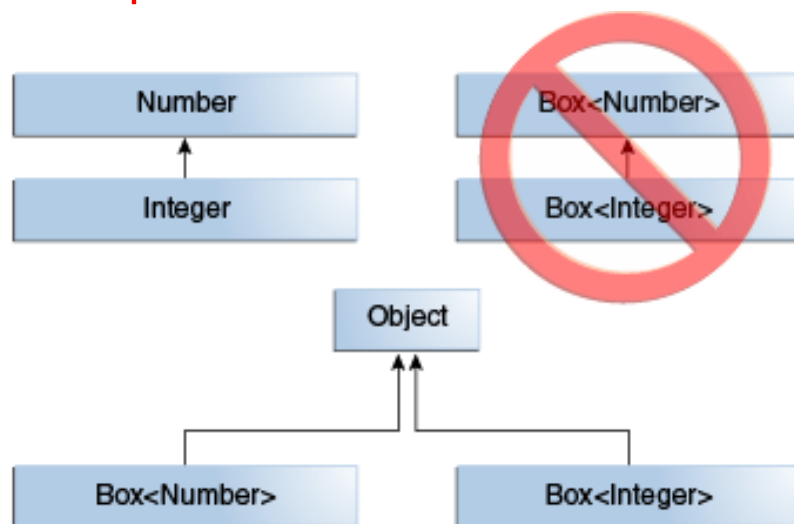
# Несколько ограничений

- Можно указывать несколько ограничений через &
- В качестве ограничения может быть класс и сколько угодно интерфейсов
- Если есть ограничение на класс, то оно должно идти первым
- ```
class A { /* ... */ }  
interface B { /* ... */ }  
interface C { /* ... */ }
```
- ```
class D <T extends A & B & C> { /* ... */ } // ОК
```
- ```
class D <T extends B & A & C> { /* ... */ } // ошибка комп-ии
```

Наследование и Generics

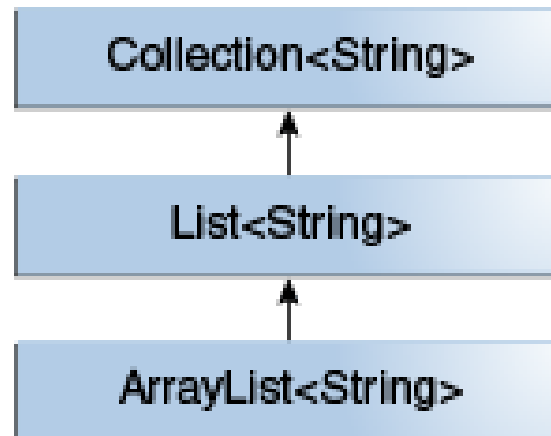
- Помним, что ссылке на родительский класс можно присвоить ссылку на объект производного класса
- `Integer someInteger = 10;`
`Number someNumber = someInteger; // OK`
- Однако
`Box<Integer> boxInteger = new Box<>();`
`Box<Number> boxNumber = boxInteger;`
// ошибка компиляции

Это ограничение
можно обойти,
рассмотрим
позже



Наследование и Generics

- **Generic** классы могут наследоваться друг от друга и реализовывать **generic** интерфейсы



Wildcards

Виды wildcards

- Для generic'ов в Java есть особый синтаксис, который называется **wildcards**
- **?**, называемый **wildcard**, означает неизвестный тип
- Бывают следующие виды **wildcards**:
 - **Upper bounded** (**extends**) – с ограничением сверху
 - **Lower bounded** (**super**) – с ограничением снизу
 - **Unbounded** – без ограничения

Зачем нужны wildcards?

- **Wildcards** позволяют ослабить ограничения на generic типы
- С этим методом есть проблема:
- ```
public void processList(List<Number> list) {
 // ...
}
```
- В данный метод можно передать только `List<Number>`, но нельзя использовать `List<Double>` и `List<Integer>`, хотя `Integer` и `Double` наследуются от `Number`



# Wildcards с ограничением сверху

- Проблему можно решить при помощи **upper bounded wildcard**:
- ```
public void processList(List<? extends Number> list) {  
    for (Number e : list) {  
        // с элементами можно работать как с  
        // ограничивающим типом  
    }  
}
```
- Теперь в метод можно передавать `List<Number>` и списки объектов производных классов, например, `List<Integer>`, `List<Double>`

Wildcards с ограничением сверху

- При использовании **upper bounded wildcard** вы не можете добавлять элементы в список:
- `public void processList(List<? extends Number> list) {
 list.add(3.2); // ошибка компиляции
}`
- Ошибка правильная, т.к. сюда могут передать, например, список `Integer`'ов, а он не должен хранить `Double`'ы
- Т.е., **upper bounded wildcard** должны использоваться только для чтения

Wildcards без ограничений

- Можно указывать просто знак ?, если мы не хотим ограничивать тип
- Это полезно, когда нам совсем не важен тип-параметр
- Пример:

```
public static void printList(List<?> list) {  
    for (Object elem: list) {  
        System.out.print(elem + " ");  
    }  
    System.out.println();  
}
```

Пользуемся
элементами списка как
Object'ами

По смыслу <?> - это
<? extends Object>

Wildcards с ограничением снизу

- В качестве типа-параметра можно использовать класс и его родителей вплоть до `Object`
- Пример:

```
public static void addNumbers(List<? super Integer> list) {  
    for (int i = 1; i <= 10; i++) {  
        list.add(i);  
    }  
}
```
- Сюда можно передать списки `List<Integer>`, `List<Number>`, `List<Object>`
- Все они могут принимать объекты `Integer`

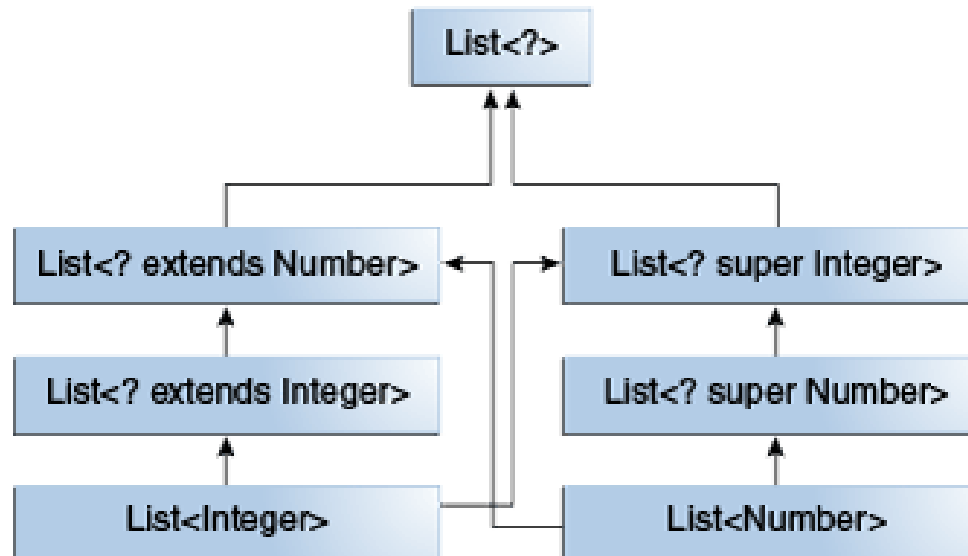
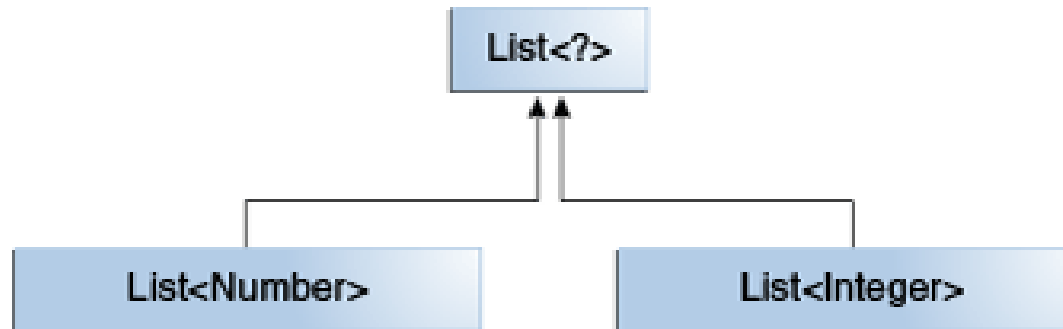
Wildcards с ограничением снизу

- При использовании **wildcards** с ограничением снизу вы можете только добавлять элементы в список
- Вы можете получать элементы только как **Object**

Наследование между generic типами

- Если использовать **wildcards**, то тогда между типами возникает связь наследования:
- ```
class Number { /* ... */ }
class Integer extends Number { /* ... */ }
```
- ```
Integer b = new Integer();  
Number a = b;
```
- ```
List<Integer> lb = new ArrayList<>();
List<Number> la = lb; // ошибка компиляции
```
- ```
List<? extends Integer> intList = new ArrayList<>();  
List<? extends Number> numList = intList;  
// все хорошо
```

Наследование между generic типами



Принцип PECS

- Чтобы понимать когда использовать **wildcards** с **extends**, а когда с **super**, существует принцип **PECS**
- **PECS (Producer-Extends Consumer-Super):**
 - Если generic тип по смыслу является **производителем (producer)** объектов типа **T**, то нужно использовать **wildcard** с **extends**
 - Если generic тип по смыслу является **потребителем (consumer)** объектов типа **T**, то нужно использовать **wildcard** с **super**
- Полезная ссылка:
- <https://habr.com/ru/company/sberbank/blog/416413/>

Принцип PECS

- **PECS (Producer-Extends Consumer-Super)**
- Есть еще другое название этого принципа - **Get and Put Principle**

Принцип PECS

- Классический пример – метод копирования элементов из одного списка в другой:
- ```
public static <T> void copy(List<? extends T> source,
 List<? super T> destination) {
 for (T e : source) {
 destination.add(e);
 }
}
```
- Здесь исходный список является производителем (он выдает объекты **T**), а результирующий – потребителем (он принимает объекты **T**)
- Этот вариант с **wildcard** очень гибкий, т.к. учитывает коллекции наследников и родителей

# Wildcards capture

- Допустим, мы хотим написать метод обмена 2 элементов любого списка:
- ```
public static void swap(List<?> list, int i, int j) {  
    list.set(i, list.set(j, list.get(i))); // ошибка компиляции  
}
```
- Ошибка возникает, потому что мы пытаемся поменять список, хотя у нас есть ограничение `extends Object`
- Но эту проблему можно обойти, если сделать вспомогательный метод

Wildcards capture

- ```
public static void swap(List<?> list, int i, int j) {
 swapHelper(list, i, j);
}
// вспомогательный метод
private static <E> void swapHelper(List<E> list, int i, int j) {
 list.set(i, list.set(j, list.get(i)));
}
```
- Смысл в том, что при помощи вспомогательного метода мы зафиксировали тип элемента коллекции
- И теперь в методе обмена не используется wildcard, и этот код нормально компилируется

# Ковариантность и др.

- Существуют следующие термины:
  - **Инвариантность** –  $T$
  - **Ковариантность** –  $? \text{ extends } T$
  - **Контравариантность** –  $? \text{ super } T$
- Кстати, массивы ссылочных типов ковариантны, т.е. можно сделать так:
  - `Integer[] numbers = { 1, 2, 3 };`  
`Object[] array = numbers;`
- Но класть в **array** можно только `Integer`

# **Затирание ТИПОВ**

# Затирание типов

```
public class Node<T> {
 private T data;
 private Node<T> next;

 public Node(T d, Node<T> next) {
 this.data = d;
 this.next = next;
 }

 public T getData() {
 return data;
 }
}
```

```
public class Node {
 private Object data;
 private Node next;

 public Node(Object d, Node next) {
 this.data = d;
 this.next = next;
 }

 public Object getData() {
 return data;
 }
}
```

# Затирание типов при extends

```
public class Node<T extends Number> {
 private T data;
 private Node<T> next;

 public Node(T d, Node<T> next) {
 this.data = d;
 this.next = next;
 }

 public T getData() {
 return data;
 }
}
```

```
public class Node {
 private Number data;
 private Node next;

 public Node(Number d, Node next) {
 this.data = d;
 this.next = next;
 }

 public Number getData() {
 return data;
 }
}
```



# Затирание в методах

- В методах затирание происходит аналогично
- Если не было ограничения на тип, то он становится `Object`
- Если было ограничение `extends SomeType`, то используется ограничивающий тип
- `public static <T extends Shape> void draw(T shape) {}`
- `public static void draw(Shape shape) {}`

# Bridge методы

- В некоторых случаях, затирание не ограничивается только подстановкой типов, а могут создаваться так называемые **bridge методы**
- Они могут создаваться, если мы хотим переопределить виртуальную функцию в дочернем классе по отношению к generic'у как в случае, который мы рассмотрим

# Bridge методы

- ```
public class Node<T> {  
    private T data;  
    public Node(T data) { this.data = data; }  
    public void setData(T data) { this.data = data; }  
}
```
- ```
public class MyNode extends Node<Integer> {
 public MyNode(Integer data) { super(data); }
 public void setData(Integer data) {
 System.out.println("MyNode");
 super.setData(data);
 }
}
```

# Bridge методы

- ```
public class Node {  
    private Object data;  
    public Node(Object data) { this.data = data; }  
    public void setData(Object data) { this.data = data; }  
}
```
- ```
public class MyNode extends Node {
 public MyNode(Integer data) { super(data); }
 public void setData(Integer data) {
 System.out.println("MyNode");
 super.setData(data);
 }
}
```

После затирания типов

Проблема – метод setData в MyNode теперь не переопределяет метод из Node, т.к. разные сигнатуры


# Bridge методы

- ```
public class MyNode extends Node {  
    public MyNode(Integer data) { super(data); }  

```

```
public void setData(Object data) {  
    setData((Integer)data);  
}
```

Созданный
компилятором
Bridge метод



```
public void setData(Integer data) {  
    System.out.println("MyNode");  
    super.setData(data);  
}
```

Он имеет
сигнатуру как в
базовом классе,
и лишь вызывает
нужный метод

```
}
```

По задаче ArrayList

Проблема

- Пытаемся реализовать `ArrayList`, делаем поля и конструктор
- ```
public class MyArrayList<E> implements List<E> {
 private E[] items;
 private int size;

 public MyArrayList() {
 items = new E[10]; // ошибка компиляции
 }

 /* реализации методов из List<E> */
}
```
- Generic'и в Java не позволяют так писать, но решение есть

# Решение

- ```
public class MyArrayList<E> implements List<E> {  
    private E[] items;  
    private int size;  
  
    public MyArrayList() {  
        items = (E[]) new Object[10]; // warning, но компил-ся  
    }  
  
    /* реализации методов из List<E> */  
}
```
- Так сделать можно – создаем массив `Object`'ов и приводим тип к `E[]`

Избавляемся от warning'a

- В данном случае warning нужно заглушить, т.к. код по смыслу верный
- Чтобы заглушить warning надо нажать на код с warning'ом, нажать **Alt+Enter**, и выбрать этот вариант:

```
public class MyArrayList<E> implements List<E> {  
    private E[] items;  
    private int count;
```

```
    public MyArrayList() {  
        items = (E[])new Object[10];  
    }
```

```
    @Override  
    public int size() {  
        return 0;  
    }
```

```
    @Override  
    public boolean isEmpty() {  
        return false;  
    }
```

| Unchecked warning | |
|-------------------|----------------------------------------------|
| 🔧 | Edit inspection profile setting |
| 🛠️ | Fix all 'Unchecked warning' problems in file |
| 🛠️ | Run inspection on ... |
| ✖ | Disable inspection |
| 👤 | Suppress all inspections for class |
| 👤 | Suppress for class |
| 👤 | Suppress for method |
| 👤 | Suppress for statement |

Избавляемся от warning'a

- Глушить warning нужно только в крайних случаях, как этот
- В 99% случаев warning реально говорит о потенциальной ошибке или неоптимальности, и его нужно исправлять