

Core Java 6

Generic - пришел, увидел, обобщил

Обобщения или generics (обобщенные типы и методы) позволяют нам уйти от жесткого определения используемых типов.

Сдѣлаем Pair

Параметризация типов

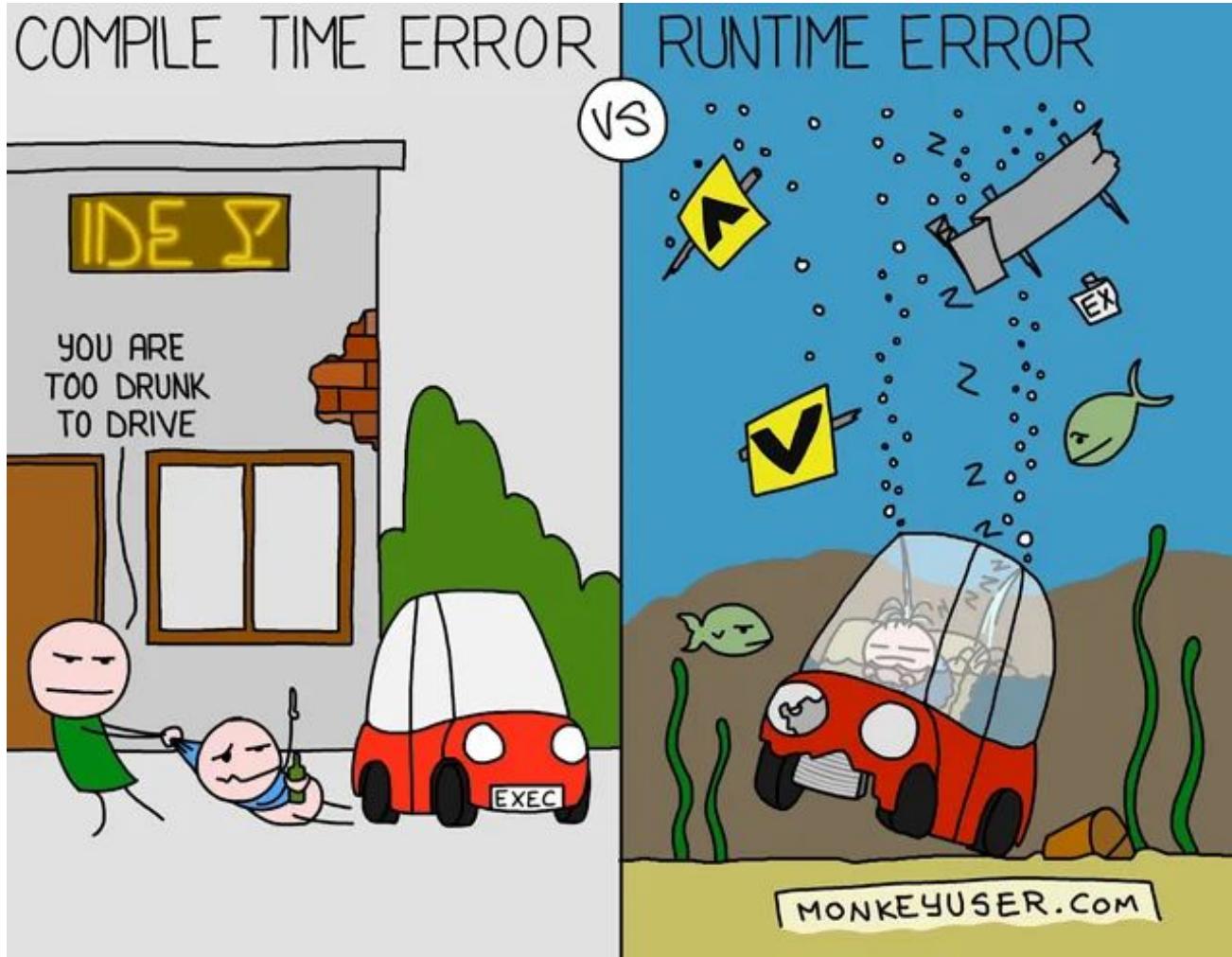
- Класс или интерфейс может быть объявлен обобщенным:
 - class List<T>
 - class Map<K, V>
 - class X<T extends Y & Z>
- Т, K, V - параметры типа
- Параметр типа может использоваться внутри нестатических членов класса:
 - Тип поля, параметра, переменной
 - Возвращаемый тип метода
 - Подставляться параметром другого типа
- Для использования параметризованного типа необходимо подставить все параметры - с помощью явных не примитивных типов, других типов или маски

До появления дженериков

```
Manager ceo = ...;
Manager cto = ...;
Employee cleaner = ...;
List managers = new ArrayList();
managers.add(ceo);
managers.add(cto);

//bug!!
managers.add(cleaner);

//typecast with runtime exception -- too late!
Manager m = (Manager) managers.get(2);
```



Generic методы (простой вариант)

```
public static <T> T getRandom(T... values) {  
    return values[ThreadLocalRandom.current().nextInt(values.length)];  
}
```

```
String randS = getRandom("a", "b", "c");  
Integer randI = getRandom(10, 13, 56);
```

Промежуточные выводы

- Использование параметризованных классов простое
- Методы - еще проще
- Написать свои - тут сложнее

Можно ли проще?

```
public static <T, K, U, M extends Map<K, U>>
Collector<T, ?, M> toMap(Function<? super T, ? extends K> keyMapper,
                         Function<? super T, ? extends U> valueMapper,
                         BinaryOperator<U> mergeFunction,
                         Supplier<M> mapFactory) {
    BiConsumer<M, T> accumulator
        = (map, element) -> map.merge(keyMapper.apply(element),
                                         valueMapper.apply(element), mergeFunction);
    return new CollectorImpl<>(mapFactory, accumulator, mapMerger(mergeFunction), CH_ID);
}
```

ЧТО



Optional + Shmoption

```
/** A box which is either empty or contains a non-null value */
static class Option<T> {
    T value;

    /** Passing null means absent value */
    public Option(T value) { this.value = value; }

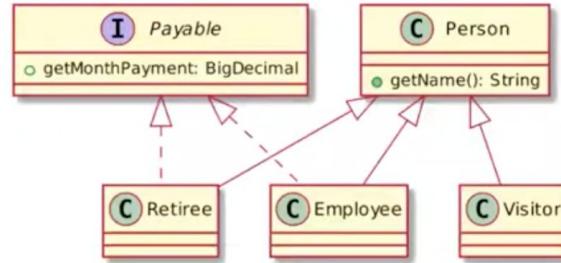
    /** Never returns null */
    public T get() {
        if(value == null) throw new NoSuchElementException();
        return value;
    }

    public T orElse(T other) { return value == null ? other : value; }

    public boolean isPresent() { return value != null; }
}
```

Оператор ромб

Intersection types

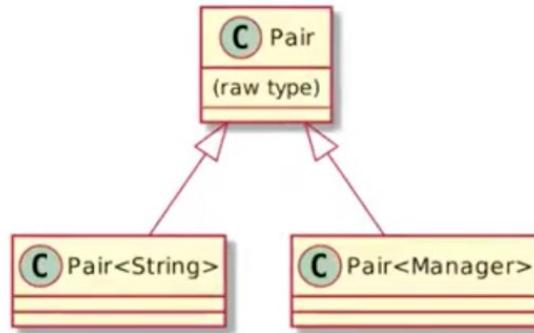


```
//через амперсанд сколько угодно интерфейсов,
//но не более одного класса
public <T extends Person & Payable>
    String getRandomNameAndPayment(List<T> items) {
    T result =
        items.get(
            ThreadLocalRandom.current().nextInt(items.size()));
    return result.getName() //из Person!
        + result.getPayment(); //из Payable!
}
```

Реализация дженериков

- Появилась в Java 5
- Была задача обратной совместимости
- Generics - возможности языка, а не платформы
- Type Erasure

Type Erasure



```
//ошибка компиляции! не знаем мы в рантайме параметр типа!
if (a instanceof Pair<String>) ...
```

```
//вот так -- получится...
if (a instanceof I Pair<?>) ...
```

Type Erasure

```
class Holder<T> {  
    private T value;  
  
    public Holder(T t) {  
        this.value = t;  
    }  
    public T get() {  
        return value;  
    }  
}
```

```
class Holder {  
    private Object value;  
  
    public Holder(Object t) {  
        this.value = t;  
    }  
    public Object get() {  
        return value;  
    }  
}
```

```
Holder<Integer> holder = new Holder<>(10);  
Integer integer = holder.get();
```

```
Holder holder = new Holder(10);  
Integer integer = (Integer) holder.get();
```

Bridge методы

```
public interface Comparable<T> {  
    int compareTo(T t);  
}
```

Bridge методы

```
public interface Comparable {  
    int compareTo(Object o);  
}  
  
public class Person implements Comparable<Person> {  
    @Override  
    public int compareTo(Person o) {  
        return 0;  
    }  
}
```

Bridge методы

```
public interface Comparable {  
    int compareTo(Object o);  
}  
  
public class Person implements Comparable<Person> {  
    @Override  
    public int compareTo(Person o) {  
        return 0;  
    }  
  
    @RealOverride  
    public int compareTo(Object o) {  
        return compareTo((Person)o)  
    }  
}
```

Wildcard

Надтипы и подтипы

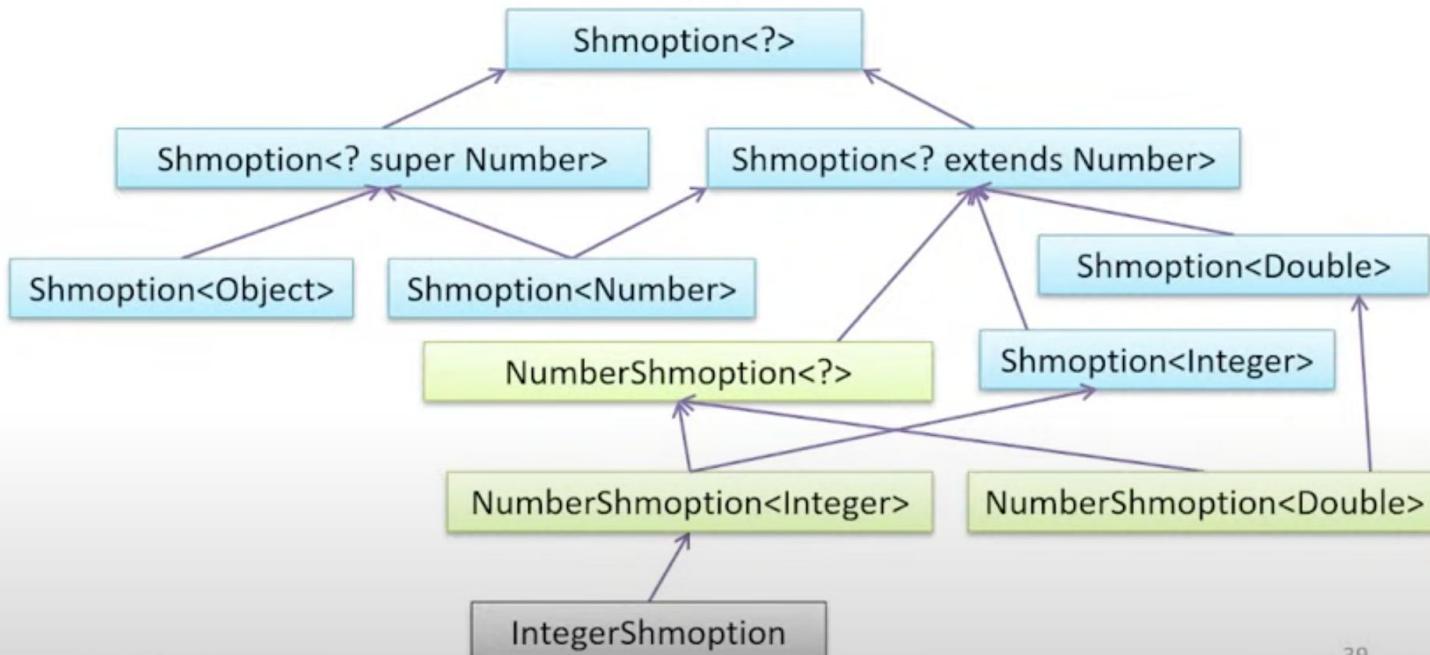
- ✓ У типов в Java определён частичный порядок “ $A > B$ ” A – надтип B или B – подтип A (“ $A :> B$ ”, если допустимо $A = B$)
- ✓ “ $A :> B$ ” – рефлексивно, антисимметрично, транзитивно (рефлексивно-транзитивное замыкание над предикатом “ $A >_1 B$ ” – прямой надтип).
- ✓ “ $A > B$ ” – антирефлексивно, антисимметрично, транзитивно.
- ✓ Надтип не должен предоставлять больше возможностей, чем предоставляет подтип.

Надтипы и подтипы

$S :> T \not\rightarrow X<S> :> X<T>$

Generic наследование

Подтипы

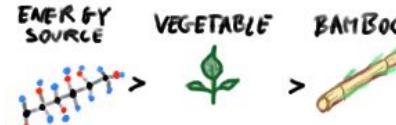


PECS

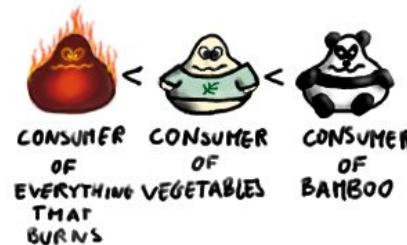
Producer Extends
Consumer Super

CONTRAVARIANCE:

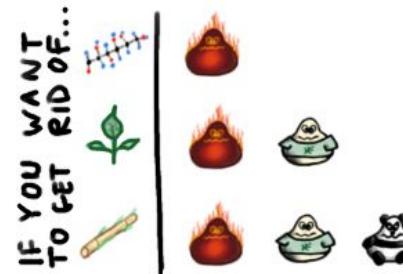
HIERARCHY OF X:



CONSUMERS [-X]:



... YOU CAN GIVE IT TO:

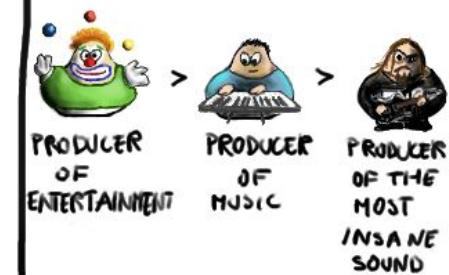


COVARIANCE:

HIERARCHY OF X:



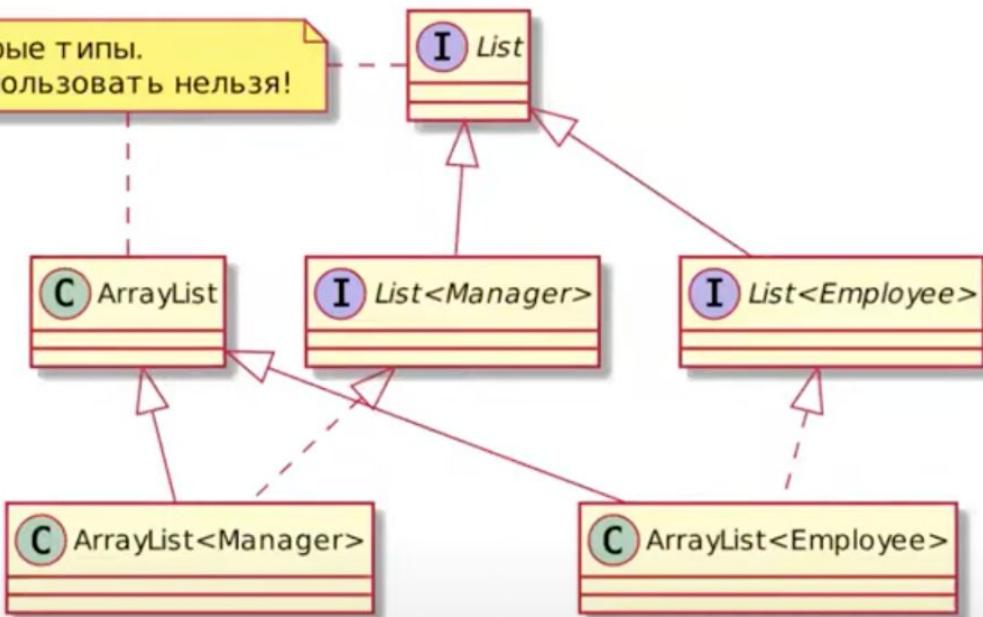
PRODUCERS [+X]:



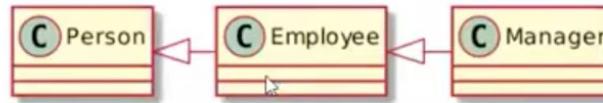
... YOU CAN GET IT FROM:



Сырые типы.
Использовать нельзя!



А если нужно так



```
List<Manager> managers = ...
List<Employee> employees = ...
```

```
//Допустимые варианты, хотим чтоб компилировалось!
employees.addAllFrom(managers);
managers.addAllTo(employees);
```

```
//Недопустимые варианты, не хотим чтоб компилировалось!
managers.addAllFrom(employees);
employees.addAllTo(managers);
```

PECS

If a parameterized type represents a T producer,
use <? extends T>;

if it represents a T consumer, use <? super T>.

Joshua Bloch

```
public static <T> T max(Collection<? extends T> coll,  
                         Comparator<? super T> comp)
```

```
Collections.max(List<Integer>, Comparator<Number>);  
Collections.max(List<String>, Comparator<Object>);
```

Covariant return type

```
interface Supplier {  
    Object get();  
}  
  
interface StringSupplier extends Supplier {  
    @Override  
    String get();  
}
```

Covariant return type

```
Class<Integer> clazz = Integer.class;  
  
Integer integer = 2016;  
Class<Integer> clazz = integer.getClass();
```

Covariant return type

```
Class<Integer> clazz = Integer.class;  
  
Integer integer = 2016;  
Class<Integer> clazz = integer.getClass(); //compile error
```

Covariant return type

```
class Parent {  
    public Number run(String s) {...}  
}
```



```
class Child extends Parent {  
    @Override  
    public Integer run(String s) {...}  
}
```

Covariant return type - если бы возвращался Class<T>

```
class Object {  
    Class<?> getClass();  
}
```



```
class Number {  
    Class<Number> getClass();  
}
```



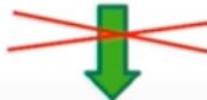
```
class Integer {  
    Class<Integer> getClass();  
}
```

Covariant return type - если бы возвращался Class<T>

```
class Object {  
    Class<?> getClass();  
}
```



```
class Number {  
    Class<Number> getClass();  
}
```



```
Number n = new Integer(1);  
Class<Number> clazz = n.getClass();
```

```
class Integer {  
    Class<Integer> getClass();  
}
```

Covariant return type - если бы возвращался Class<T>

```
class Object {  
    Class<?> getClass();  
}
```



```
class Number {  
    Class<? extends Number> getClass();  
}
```



```
Number n = new Integer(1);  
Class<? extends Number> c = n.getClass()
```

```
class Integer {  
    Class<? extends Integer> getClass();  
}
```

Снова Generic методы и конструкторы

```
static <T> void setNotNull(Shmoption<? super T> shmoption, T value) {  
    if (value == null) throw new IllegalArgumentException();  
    shmoption.set(value);  
}  
  
setNotNull(n, 123);  
ShmoptionUtils.<Number>setNotNull(n, 123);
```

Снова Generic методы и конструкторы - 2 ссылки

```
Object obj;  
  
@SuppressWarnings("unchecked")  
<T> T getT() {  
    return null;  
}  
  
<T> T getT() {  
    return (T) obj;  
}
```

Снова Generic методы и конструкторы - нельзя инстанцировать типы параметры

```
class Pair<T> {  
    T newValue {  
        return new T(); //увы, ошибка компиляции!  
    }  
}
```

Generic и примитивы

- День сегодняшний: нужна производительность? – пишем специальные реализации.
 - В стандартной библиотеке:
 - Stream<Integer> → IntStream
 - Stream<Double> → DoubleStream.
 - В специализированных библиотеках вроде [fastutil](#):
 - ArrayList<Integer> → IntArrayList,
 - HashMap<Integer, V> → Int2ObjectMap<V> (ВНИМАНИЕ: реальная потребность в таких библиотеках возникает редко!!)
- День завтрашний: Project Valhalla, specialized generics. Решит проблему раз навсегда.

Снова Generic методы и конструкторы - нельзя инстанцировать типы параметры

```
class Container<T> {
    //bounded wildcard type, речь впереди
    Class<? extends T> clazz;

    Container(Class<? extends T> clazz) {
        this.clazz = clazz;
    }

    T newInstance() throws ReflectiveOperationException {
        //если нашёлся открытый конструктор без параметров!
        return clazz.newInstance();
    }
}
```

```
Container<Employee> container1 = new Container<>(Employee.class);
```

Снова Generic методы и конструкторы - нельзя инстанцировать массив типа параметры

```
public T[] toArray() {  
    //Не скомпилируется  
    return new T[size];  
}
```

Решается передачей параметра, например, в ArrayList:

```
/* Если массив достаточных размеров -- использую м его,  
если недостаточных -- конструируем новый через рефлексию*/  
public <T> T[] toArray(T[] a)
```

Стирание

- ✓ Не все типы известны во время выполнения
- ✓ В частности, значения параметров типов «стёрты»
- ✓ Преобразование типов не всегда бывает безопасным (checked)
- ✓ Если оно небезопасно, вы получите unchecked cast warning
- ✓ При выполнении это может привести к проблемам (ClassCastException) в дальнейшем
- ✓ Программа считается безопасной с точки зрения системы типов, если типы выражений в рантайме соответствуют типам во время компиляции (ClassCastException происходит только при явном преобразовании типа).
- ✓ Если предупреждений нет, то программа безопасна *

Стирание

```
Shmoption<Integer> integer = new Shmoption<>(10);
Shmoption<String> string = ((Shmoption<String>) integer); // ошибка
Shmoption<?> any = integer;
Shmoption<String> string2 = (Shmoption<String>) any; // предупреждение
String s = string2.get(); // ClassCastException

NumberShmoption<Integer> number =
    (NumberShmoption<Integer>)integer; // normally
```

Доступно в рантайме

```
public class Runtime<T extends Number>
    implements Callable<Double> {
    private final List<Integer> integers = emptyList();

    public List<T> numbers() { return emptyList(); }

    public List<String> strings() { return emptyList(); }

    @Override
    public Double call() { return 0d; }
}
```

Дженерики и массивы

```
Shmoption<?>[] array = new Shmoption<?>[10];   
Shmoption<Integer>[] arrayInt = new Shmoption<Integer>[10]; 
```

Дженерики и массивы

```
Shmoption<Integer>[] arrayInt = new Shmoption[10]; // предупреждение
Object[] obj = arrayInt;
obj[0] = new Shmoption<>("foo");
Shmoption<Integer> shmoption = arrayInt[0];
Integer x = shmoption.get();
System.out.println(x);
```

Дженерики и массивы

```
//Не скомпилируется: Generic Array Creation.  
List<String>[] a = new ArrayList<String>[10];  
//...ведь такой массив не будет иметь  
//полную информацию о своих элементах!
```

Дженерики и массивы

```
List<String>[] a = (List<String>[]) new List<?>[10];
Object[] objArray = a;
                ▾

objArray[0] = (List<String>) Arrays.asList("foo");
//a[1] не пропустит в compile-time
//но objArray[1] пропустит и в compile-time, и в run-time
objArray[1] = (List<Manager>) Arrays.asList(new Manager());

//Runtime error: Manager cannot be cast to String
String s = a[1].get(0);
//...это и называется heap pollution.
```

Дженерики и массивы

//Скомпилируется ?

```
List<Number>[] lists = new ArrayList<Number>[10];  
List<?>[] lists = new ArrayList<?>[10];
```

Дженерики и массивы

//Скомпилируется ?

```
List<Number>[] lists = new ArrayList<Number>[10]; //error  
List<?>[]      lists = new ArrayList<?>[10];      //ok
```

Дженерики и массивы

```
List<Number>[] lists = new ArrayList<Number>[10];  
  
Object[] objects = lists;  
objects[0] = new ArrayList<String>();  
  
lists[0].add(1L); // ☹
```

Varargs

```
static void printAll(Object... objects) {  
    for (Object object : objects) {  
        System.out.println(object);  
    }  
}
```

```
printAll("a", 1, "b", 2.0);
```

- ✓ Последний параметр
- ✓ Похож на массив
- ✓ Можно превратить массив в vararg, не теряя совместимости

Varargs

```
static void printAll(Object... objects) {  
    for (Object object : objects) {  
        System.out.println(object);  
    }  
}  
  
printAll(null, null);  
printAll(null);
```

Varargs

```
static <T> boolean isOneOf(T value, T... options) {  
    for (T option : options) {  
        if (Objects.equals(value, option)) return true;  
    }  
    return false;  
}
```

Varargs

```
@SafeVarargs
static <T> boolean isOneOf(T value, T... options) {
    for (T option : options) {
        if (Objects.equals(value, option)) return true;
    }
    return false;
}
```

Varargs

```
//warning
public void run(List<String>... lists) {
    Object[] objectArray = lists;
    objectArray[0] = Arrays.asList(42);
    String s = lists[0].get(0); // ClassCastException
}
```

Heap Pollution

```
@SafeVarargs
static <T> boolean isOneOf(T value, T... options) {
    for (T option : options) {
        if (Objects.equals(value, option)) return true;
    }
    return false;
}
```

```
isOneOf(shmoption, new Shmoption<>("foo"),
        new Shmoption<>("bar"));
```

Почему второй вариант допустим?

//Скомпилируется ?

```
List<Number>[] lists = new ArrayList<Number>[10]; //error  
List<?>[]      lists = new ArrayList<?>[10];      //ok
```

Про wildcard

- Что можно положить сюда ? .add()

```
List<? extends Number> numbers = new ArrayList<>()
```

- А сюда ?

```
List<? super Number> numbers = new ArrayList<Object>();
```

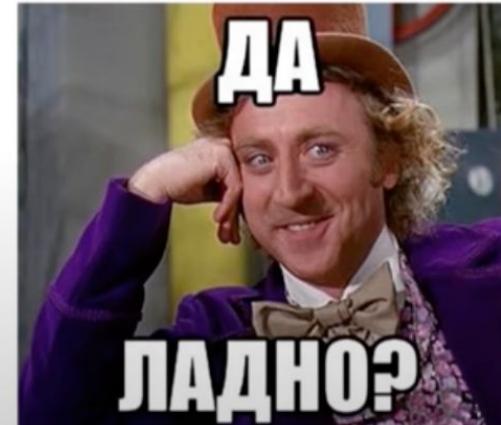
```
List<? extends Number> numbers = new ArrayList<>()
```

казалось бы..

- Number x
- Integer x
- Double x
-

На самом деле

- null



```
// что видит компилятор
List<? extends Number> numbers = ????

// что можно присвоить
numbers = new ArrayList<Number>();
numbers = new ArrayList<Integer>();
numbers = new ArrayList<Long>();
...

public void process(List<? extends Number> numbers) {
    numbers.add(234L);
}
```

```
// что видит компилятор
List<? extends Number> numbers = ????

// что можно присвоить
numbers = new ArrayList<Number>();
numbers = new ArrayList<Integer>();
numbers = new ArrayList<Long>();

...
// Компилятор не знает, чем на самом деле параметризован
List, поэтому безопасно можно добавить только null
public void process(List<? extends Number> numbers) {
    numbers.add(234L); // валидно только для List<Number> и
                      <Long>
}
```

Зачем?

List<? **extends** Number> похож на Number[]
с разрешением только на чтение

```
Number[] numbers = new Integer[10];  
List<? extends Number> numbers = new ArrayList<Integer>();
```

```
List<? super Number> numbers = new ArrayList<>()
```

казалось бы..

- Object x
- Number +

На самом деле

- все что ? extends Number
- Number +
- Integer +
- Double +
- null +

```
// что видит компилятор
List<? super Number> numbers = ????

// что можно присвоить
numbers = new ArrayList<Object>();
numbers = new ArrayList<Number>();

public void process(List<? super Number> numbers) {
    numbers.add(234L);
    numbers.add(100D);
    numbers.add(new Object());
}
```

Что тут не так

```
class GenericException<T> extends Exception {
    private final T details;

    public GenericException(T details) {
        this.details = details;
    }

    public T getDetails() {
        return details;
    }
}
```

Нельзя параметризовать

- Классы, имеющие в предках Throwable
- Анонимные классы
- Enums

```
try {
    run();
} catch (GenericException<String> e) {
    ...
} catch (GenericException<Integer> e) {
    ...
}
```

Нельзя использовать в static контексте

```
public class Container<T> {
    private static T value; //не скомпилируется.
    public static T getValue(); //не скомпилируется
}

//Статический контекст ОДИН НА ВСЕХ
Container<Foo>.getValue();
Container<Bar>.getValue();
```

Нельзя реализовывать разные параметризации одного и того же интерфейса

Source code

```
class Employee implements Comparable<Employee>{
    @Override
    int compareTo(Employee e) {
        ...
    }
}

class Manager
    extends Employee
    implements Comparable<Manager> {
    @Override
    int compareTo(Manager m) {
        ...
    }
}
```

Compiled

```
class Manager
    extends Employee
    implements Comparable{
    //bridge method for Employee
    int compareTo(Object m) {
        return compareTo((Manager)m);
    }

    //bridge method for Manager
    int compareTo(Object e) {
        return compareTo((Employee)e);
    }
}
```

Wildcard capture

```
public static void swap(Pair<?> p) {  
    Object f = p.getFirst();  
    Object s = p.getSecond();  
    //УУУПС!!  
    // (мы знаем, что они правильного типа,  
    // но ничего не можем поделать)  
    p.setFirst(...);  
    p.setSecond(...);  
}
```

метод с type capture

```
public static void swap(Pair<?> p) {  
    return swapHelper(p);  
}  
  
private static <T> void swapHelper(Pair<T> p) {  
    T f = p.getFirst();  
    p.setFirst(p.getSecond());  
    p.setSecond(f);  
}
```

Recursive generics

```
class Holder<E, SELF extends Holder<E, SELF>>{
    E value;
    SELF setValue(E value) {
        this.value = value;
        return (SELF) this;
    }
}

class StringHolder extends Holder<String, StringHolder> {
    void doSmth() {...};
}

StringHolder h = new StringHolder().setValue("aaa").doSmth()
```

Recursive generics

```
BaseStream<T, S extends BaseStream<T, S>> {  
    S sequential();  
    S parallel()  
}
```

```
Stream<T> extends BaseStream<T, Stream<T>>
```

```
stream.filter(Objects::nonNull)  
    .parallel()  
    .map(Integer::parseInt)
```

Recursive generics

```
BaseStream<T> {
    BaseStream<T> sequential();
    BaseStream<T> parallel()
}
```

```
Stream<T> extends BaseStream<T>
```

```
stream.filter(Objects::nonNull)
    .parallel()
    .map(Integer::parseInt)
```

Что тут не так

```
public class Helper<T> {
    public List<Integer> numbers() {
        return Arrays.asList(1, 2);
    }

    public static void main(String[] args) {
        Helper helper = new Helper<>();
        for (Integer number : helper.numbers()) {
            ...
        }
    }
}
```

Raw удаляет всю информацию о дженерики

```
public class Helper<T> {
    public List<Integer> numbers() {
        return Arrays.asList(1, 2);
    }

    public static void main(String[] args) {
        Helper helper = new Helper<>();
        List list = helper.numbers();
        for (Integer number : list) { //error
            ...
        }
    }
}
```

```
public class Helper<T> {
    public List<Integer> numbers() {
        return Arrays.asList(1, 2);
    }

    public static void main(String[] args) {
        Helper<?> helper = new Helper<>();
        for (Integer number : helper.numbers()) {
            ...
        }
    }
}
```

```
public static void main(String[] args) {  
    String s = newList(); // почему компилируется?????????  
}  
  
private static <T extends List<Integer>> T newList() {  
    return (T) new ArrayList<Integer>();  
}
```

