

Лекция 12а

Generic data types (generics) (parametric polymorphism)

- 12.1 Въведение**
- 12.2 Необходимост от параметри за тип**
- 12.3 Реализация и компилация**
- 12.4 Параметризирани по тип класове и интерфейси**
- 12.5 Забранени операции при параметризиране по тип**
- 12.6 Параметризиране с повече от една горна граница**
- 12.7 Типове данни с подразбиращи се (raw) параметри за тип.**
- 12.8 Инвариантност, ковариантност и контравариантност.**
- 12.9 Параметър ? за означаване на неизвестен тип**
- 12.10 Долна граница за параметър**
- 12.11 Принципът Get- Put**

Задачи



Основни теми

- Създаване на параметри за тип на методи, за извършване на идентични действия с аргументи от различен тип.
- Създаване на параметри за тип на клас.
- Презареждане на параметризирани по тип методи с други параметризирани или не-параметризирани методи.
- Използване на шаблони, когато не е нужна точна информация за аргумент на метод в тялото на метода.
- Връзката между използване на параметризираните типове и наследствеността
- Принципът GET (Consumer) -PUT (Producer) -> super- extends
- Инвариантност, ковариантност и контравариантност

12a.1 Въведение

Пораждащи прототипове (*generics*)

- Въведение от J2SE 5.0
- Позволява да се пише единствен метод, който, примерно, да сортира, както цели числа, така и всеки друг тип данни за които е дефиниран способ за наредбата им- *параметризиран шаблонна метод*
- Позволява да се пише единствен клас `Stack`, който, примерно, да позволява съставяне на списък, както от цели числа, така и числа с плаваща запетая, низове и данни от всеки друг тип- *параметризиран шаблон на клас*

12a.2 Необходимост от параметри за тип

Презареждане на методи (*overloading*)

- Използват се за дефиниране на аналогични операции върху различни типове от данни

Примери:

- *конструктори на класове*
- *Set методи*
- Версии на метода `printArray`
 - Извежда на печат масив от `Integer`
 - Извежда на печат масив от `Double`
 - Извежда на печат масив от `Character`

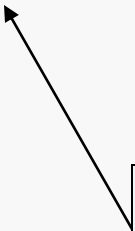
Забележка: Само референтни типове данни могат да се използват с параметризирани по тип методи и класове

```
1 // Fig. 12a.1: OverloadedMethods.java
2 // Using overloaded methods to print array of different types.
3
4 public class OverloadedMethods
5 {
6     // method printArray to print Integer array
7     public static void printArray( Integer[] inputArray )
8     {
9         // display array elements
10        for ( Integer element : inputArray )
11            System.out.printf( "%s ", element );
12
13        System.out.println();
14    } // end method printArray
15
16    // method printArray to print Double array
17    public static void printArray( Double[] inputArray )
18    {
19        // display array elements
20        for ( Double element : inputArray )
21            System.out.printf( "%s ", element );
22
23        System.out.println();
24    } // end method printArray
25
```

Метод `printArray` има за аргумент масив от `Integer` обекти

Метод `printArray` има за аргумент масив от `Double` обекти

```
26 // method printArray to print Character array
27 public static void printArray( Character[] inputArray )
28 {
29     // display array elements
30     for ( Character element : inputArray )
31         System.out.printf( "%s ", element );
32
33     System.out.println();
34 } // end method printArray
35
36 public static void main( String args[] )
37 {
38     // create arrays of Integer, Double and Character
39     Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
40     Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
41     Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
42
```



Метод printArray има за аргумент масив от Character обекти

```
43 System.out.println( "Array integerArray contains:" );
44 printArray( integerArray ); // pass an Integer array
45 System.out.println( "\nArray doubleArray contains:" );
46 printArray( doubleArray ); // pass a Double array
47 System.out.println( "\nArray characterArray contains:" );
48 printArray( characterArray ); // pass a Character array
49 } // end main
50 } // end class OverloadedMethods
```

```
Array integerArray contains:
1 2 3 4 5 6
```

```
Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
```

```
Array characterArray contains:
H E L L O
```

При компиляция, по типа на аргумента (т.е., **Integer[]**), се определя кой метод **printArray** да бъде извикан- този с единствен аргумент от тип **Integer[]** (редове 7-14)

При компиляция, по типа на аргумента (т.е., **Double[]**), се определя кой метод **printArray** да бъде извикан- този с единствен аргумент от тип **Double[]** (редове 17-24)

При компиляция, по типа на аргумента (т.е., **Character[]**), се определя кой метод **printArray** да бъде извикан- този с единствен аргумент от тип **Character[]** (редове 27-34)

12a.2 Необходимост от параметри за тип

Общото в тези `printArray` методи

- Типът на масива се появява на две места
 - В заглавието на метода
 - Във `for` командата

Обединяваме всички такива `printArray` метода в **един** *параметризиран по тип метод* като

- Заменяме името на типа на данните в заглавието на метода и навсякъде където типът на тези данни се използва в метода с име `E` на параметър за тип
- Дефинираме един единствен `printArray` метод
 - Позволява да се извежда текстовото описание на масив от данни от произволен **референтен** тип

```
1 public static void printArray( E[] inputArray )
2 {
3     // display array elements
4     for ( E element : inputArray )
5         System.out.printf( "%s ", element );
6
7     System.out.println();
8 } // end method printArray
```

Заменяме типа на данните с
едно единствено име за тип **E**

Заменяме типа на данните с
едно единствено име за тип **E**

%s позволява да се изведе
текстовото описание на обекти
от произволен тип **E**, който
реализира метода
toString()

Fig. 12a.2 | Методът `printArray` в който всички действителни типове данни са обозначени с параметър за тип **E**.

12a.3 Реализация и компиляция

В случаите, аналогични на Fig. 12a.1, когато презаредените методи са идентични за множество от типове данни е за предпочитане да се използва параметризиран по тип шаблон на метода:

- Извикването на методите са идентични**
- Връщаните данни са идентични**

12a.3 Реализация и компилация

Реализираме Fig. 12a.1, като параметризиран шаблон на метод.

Декларация на параметризиран по тип метод:

- ✓ Декларацията параметър за тип е необходима, за да се различи параметър за тип от идентификатор на референтен тип
- **Секция за деклариране на параметрите за типове:**
 - Ограничена с **ъглови скоби**(< и >)
 - Разположена е **преди описанието за типа на връщаните данни**
 - Съдържа един или повече параметри за описание на тип
 - Наричат се още **формални параметри**

12a.3 Реализация и компилация

Параметър за тип

- Нарича се **параметър за тип**, приема стойност **референтен тип**
- **Идентификатор** задаващ име на **параметър за тип**
- Използва се за **деклариране** на *тип за връщани данни*, *типове на аргументи* на метод и *типове на локални данни*
- Означава **място за вмъкване на истинските типове данни** при изпълнението на *параметризиран по тип* метод
 - Истинските типове данни- типове на реалните аргументи
- Може **да се декларират само веднъж**, но могат да се използват многократно в тялото на метода

```
public static < E > void printTwoArrays(  
                                E[] array1, E[] array2 )
```

12a.3 Реализация и компилация

Конвенции за имена на параметри за тип:

E **Element** (*елементи на масиви, списъци и множества*)

K **Key** (*ключ за сортиране, търсене*)

N **Number** (*цифров тип*)

T **Type** (*общ тип данни*)

V **Value** (*тип данна означаваща стойност*)

S, **U**, **V**,.. и пр. за означаване *2-ри*, *3-ти*, *4-ти*.. и пр. параметър
за тип

Обичайна грешка при програмиране

Пропускането да се постави секцията за декларация с описание на параметрите за типа на метода води до грешка при компилация.

```
1 // Fig. 12a.3: GenericMethodTest.java
2 // Using generic methods to print array of different types.
3
4 public class GenericMethodTest
5 {
6     // generic method printArray
7     public static < E > void printArray( E[] inputArray )
8     {
9         // display array elements
10        for ( E element : inputArray )
11            System.out.printf( "%s ", element );
12
13        System.out.println();
14    } // end method printArray
15
16    public static void main( String args[] )
17    {
18        // create arrays of Integer, Double and Character
19        Integer[] intArray = { 1, 2, 3, 4, 5 };
20        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
21        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };
22    }
```

Използваме секция за деклариране на параметър за тип в параметризирания по тип метод `printArray`

Секцията за деклариране на параметри за тип се огражда с ъглови скоби (< и >)

Използваме параметъра за тип при деклариране на типа на локална данна в `printArray`


```
23 System.out.println( "Array integerArray contains:" );
24 printArray( integerArray ); // pass an Integer array
25 System.out.println( "\nArray doubleArray contains:" );
26 printArray( doubleArray ); // pass a Double array
27 System.out.println( "\nArray characterArray contains:" );
28 printArray( characterArray ); // pass a Character array
29 } // end main
30 } // end class GenericMethodTest
```

Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contains:
H E L L O

Изпълнение на *параметризирани по тип* метод printArray с Integer масив

Изпълнение на *параметризирани по тип* метод printArray с Double масив

Изпълнение на *параметризирани по тип* метод printArray с Character масив

Правила за добро програмиране 12а.1

Препоръчва се параметрите за тип да се обозначават с отделни главни букви. Обикновено, параметър **за тип на елемент от масив** или множество се означава с **E** по аналогия с “**E**lement.”

Обичайна грешка при програмиране

Грешка при компилация възниква, ако извикваната декларация за метод не може да се съпостави на дефиниран метод (*обикновен* или *параметризиран по тип*).

Обичайна грешка при програмиране

Компилаторът извежда грешка, ако не може да открие единствена дефиниция за метод, съвпадаща точно с извикването му, а открива *два или повече параметризирани по тип* методи, които удовлетворяват извикването на метода.

12a.3 Реализация и компиляция

Параметризираните методи могат да се презареждат (overload)

- От друг *параметризиран по тип метод*
 - Със същото име , други аргументи на метода (**друг брой и поредност на тип аргументите**)
- От *не-пораждащи* методи
 - Със **същото име** и **същият брой** аргументи

12a.3 Реализация и компилация

Когато компилаторът открие извикване на метод

- **Търси метод с най- близко съвпадение на “*подписа*” на метода (*име и списък от аргументи*)**
 - **Първо се търси за точно съвпадение по име и списък от аргументи**
 - **Ако не се открие точно съвпадение, се търсе неточно, но приложимо съвпадение с дефиниран метод**

12a.3 Реализация и компиляция

Пример 1:

Пораждащият метод `printArray` от Fig. 12a.3 може да се **презареди с друг параметризиран по тип метод `printArray`**, който има **допълнителни аргументи `lowSubscript` и `highSubscript`**, задаващи подмножество от елементи за извеждане на печат

12a.3 Реализация и компиляция

Пример 2:

Пораждащият метод `printArray` от Fig. 12a.3 може да се **презареди с друг параметризиран по тип метод** `printArray` от **Fig. 12a.3** с версия специфична за `String` обекти, при което тези обекти се извеждат на печат в колонки

12a.3 Реализация и компилация

Транслиране на пораждащи методи при компилация

- Компиляторът **изтрива** се секцията за деклариране на параметрите за тип
 - **Замества** параметрите за тип с реалните типове данни при създаването на обектите. Този тип се определя от ограниченията, наложени на областта от стойности, които може да приема параметъра за тип
 - **Параметрите за тип не съществуват по време на изпълнението(runtime)**
 - Подразбиращият се тип за **заместване** , когато не са наложени ограничения на параметъра за тип, е **Object**
- **Разлики:** Този подход е различен от аналогични техники като **базовите схеми (template) в C++**, при които се генерира отделно копие на сорс кода и то се компилира за всеки тип, предаден на аргумент на метод , чиито тип е зададен параметър за тип.

```
1 public static void printArray( Object[] inputArray )
2 {
3     // display array elements
4     for ( Object element : inputArray )
5         System.out.printf( "%s ", element );
6
7     System.out.println();
8 } // end method printArray
```

Изтрива се секцията на параметрите и се замества с типа на реалния обект **Object**

Заместваме типа на параметрите с реалния обект **Object**

Предимствата на параметризираните методи са най-големи, когато се изисква връщане на данни- горният вариант на метод **printArray** върши същата работа като неговата пораждаща версия

Fig. 12a.4 | Параметризираният по тип метод **printArray** след като изтриването е изпълнено от компилатора.

12a.4 Параметризирани по тип класове и интерфейси

Параметризирани по тип класове и интерфейси

- Използва сбита и проста схема от означения за указване на реалните типове
- По време на **компиляция**, Java
 - Гарантира **сигурност при използване на типовете**-проверява за съвместимост с декларираните параметри за тип
 - Използва **техниката на изтриване**, позволяващо на потребителският код да изпълни пораждащия метод

12a.4 Параметризирани по тип класове и интерфейси

Параметрите за тип са налични единствено по време на компилиране. Щом Компиляторът потвърди, че параметризирания тип удовлетворява изискванията за сигурност на кода, параметрите за тип се изтриват в параметризирания тип и се заместват с тип в съответствие с наложените ограничения на параметрите за тип. На пример, нека по време на компилиране да имаме (a). По време на изпълнение, сорс кодът се променя в (b).

<pre>ArrayList<String> list = new ArrayList<>(); list.add("Sofia"); String state = list.get(0);</pre>	<pre>ArrayList list = new ArrayList(); list.add("Sofia"); String state = (String) (list.get(0));</pre>
---	--

a) Преди компилиране

b) По време на изпълнение



12a.4 Параметризирани по тип класове и интерфейси

При компилиране на параметризирани по тип методи, класове и интерфейси, когато няма наложени ограничения върху параметъра за тип, параметърът за тип се замества с **Object**. Например, видът на един параметризиран по тип метод по време на компилация и по време на изпълнение е показан съответно на (a) и (b).

```
public static <E> void print(E[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

a) Преди компилиране

```
public static void print(Object[] list) {  
    for (int i = 0; i < list.length; i++)  
        System.out.print(list[i] + " ");  
    System.out.println();  
}
```

b) По време на изпълнение

12a.4 Параметризирани по тип класове и интерфейси

Пример- традиционен клас

- Създаваме обикновен `class Box`,
 - работи с обекти от произволен клас
 - Има два метода-
 - `add()` – служи за добавяне на елементи в `Box`
 - `get()` – служи за извличане на елементи от `Box`

class Box

```
public class Box {  
  
    private Object object;  
  
    public void add(Object object) {  
        this.object = object;  
    }  
  
    public Object get() {  
        return object;  
    }  
}
```

Може да се добавят и
извличат произволни
обекти- методите реферират
Object

```
public class BoxDemo1 {  
  
    public static void main(String[] args) {  
  
        // ONLY place Integer objects into this box!  
        Box integerBox = new Box();  
  
        integerBox.add(new Integer(10));  
        Integer someInteger = (Integer)integerBox.get();  
        System.out.println(someInteger);  
    }  
}
```

Ако трябва да се работи само
с Integer обекти, **няма как да
са ограничи или проверява**
по време на компилация-
може **само да се препоръча** в
документацията
съпътстваща class Box

Ако потребителят не се е
съобразил с документацията,
ще възникне грешка при
преобразуване до Integer

class Box

```
public class BoxDemo2 {  
  
    public static void main(String[] args) {  
  
        // ONLY place Integer objects into this box!  
        Box integerBox = new Box();  
  
        // Imagine this is one part of a large application  
        // modified by one programmer.  
        integerBox.add("10"); // note how the type is now String  
  
        // ... and this is another, perhaps written  
        // by a different programmer  
        Integer someInteger = (Integer)integerBox.get();  
        System.out.println(someInteger);  
    }  
}
```

Програмист, който не е чел
документацията, въвежда
String вместо Integer

... води до грешка
ClassCastException при
преобразуване до Integer

12a.4 Параметризирани по тип класове и интерфейси

Параметризирани класове

- Концепцията за структура от данни като `ArrayList` например, може да се разбере **независимо от данните които съхранява.**
- Параметризираните по тип класове позволяват **да се разглеждат структурите от данни независимо от типа на данните**, които се структурират с тях
- Дава възможност за **многократно използване** на програмен код
- Наричат се също **параметризирани типове** (*взимат един или повече параметри*)

Пример: `ArrayList < Double >`

(“ ArrayList om Double,” “ ArrayList om Integer,” ...,“ ArrayList om Employee,”)

class Box с параметър за тип

```
/**
 * Generic version of the Box class.
 */
public class Box<T> {

    private T t; // T stands for "Type"

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

Параметър
за тип на
данните на
class
Box

Секция за дефиниране на
параметри за тип

12a.4 Параметризирани по тип класове и интерфейси

За рефериране на пораждащия `class Box` в потребителски програми заместваме параметъра `T` с конкретен съществуващ клас, например `Integer`:

```
Box<Integer>    integerBox;
```

Това декларира, че `integerBox` ще е референция към "Box от `Integer`" и тази декларация така и се чете.

12a.4 Параметризирани по тип класове и интерфейси

За дефиниране на `integerBox` използваме:

```
var integerBox = new Box<Integer>();
```

или декларация и дефиниция заедно

```
Box<Integer> integerBox =  
    new Box<Integer>();
```

```
Box<Integer> integerBox =  
    new Box<>();
```

Важно:

```
var integerBox = new Box <>();
```

```
// типът на integerBox е Box<Object>
```

class Box с параметър за тип

```
public class BoxDemo3 {  
  
    public static void main(String[] args) {  
        Box<Integer> integerBox = new Box<Integer>();  
        integerBox.add(new Integer(10));  
        Integer someInteger = integerBox.get(); // no cast!  
        System.out.println(someInteger);  
    }  
}
```

```
BoxDemo3.java:5: add(java.lang.Integer)  
                  in Box<java.lang.Integer>  
cannot be applied to (java.lang.String)  
    integerBox.add("10");  
                  ^  
1 error
```

Няма нужда от явно
преобразуване на типа-
параметърът за тип
гарантира, че методът
get() връща Integer

Компиляторът дава **синтактична
грешка** при опит за добавяне на
погрешен тип данни (String
например) към Box

Software Engineering факти

Параметризираните прототипове за методи и класове са измежду едни от най- мощните средства за програмиране в Java.

12a.4 Параметризирани по тип класове и интерфейси

Пример

interface Comparable < T >

- Декларира метод
int compareTo(T object)
- В частност **class Integer** имплементира интерфейса **Comparable<Integer>** позволява сравнения от вида **integer1.compareTo(integer2)**
 - Връща 0 ,ако двата обекта са равни
 - Връща -1 , ако **integer1** е по- малък от **integer2**
 - Връща 1 ,ако **integer1** е по- голям от **integer2**
- Позволява сравнение на два обекта от един клас, чиито тип е **зададен с параметър за тип**

12a.4 Параметризирани по тип класове и интерфейси

interface с пораждащ тип

- Декларира метод, позволяващ сравнение на обекти от един и същ тип
- Позволява да се напише **една единствена дефиниция за интерфейс** за описване на множество от свързани логически типове

Пример

interface Comparable< T >

- **package java.lang**
- Декларира метод за сравнение на два обекта от един и същ клас
- Всички класове “*пакетиращи*” примитивни данни реализират този интерфейс

12a.4 Параметризирани по тип класове и интерфейси

Пример на Fig. 12a.5

- параметризиран по тип метод за определяне на най-големия от три аргумента на метод
- Използва параметър за тип на връщаните данни за определяне на типа на връщаните данни и типа на аргументите
- При сравнението на референтни данни **не може да се използва** аритметично сравнение '>', '<', '==', '<=', '>='

```
public static < T > T maximum( T x, T y, T z ) // does not compile!!!
{ // T is “erased” at compile time and replaced by Object at runtime!
    T max = x; // assume x is initially the largest
    if ( y > max )
        max = y; // y is the largest so far
    if ( z > max )
        max = z; // z is the largest
    return max; // returns the largest object
} // end method maximum
```

```

1 // Fig. 12a.5: MaximumTest.java
2 // Generic method maximum returns the largest of three objects.
3

```

```

4 public class MaximumTest
5 {
6     // determines the largest of three Comparable objects
7     public static < T extends Comparable< T > > T maximum( T x, T y, T z )
8     {
9         T max = x; // assume x is initially the largest
10
11         if ( y.compareTo( max ) > 0 )
12             max = y; // y is the largest so far
13
14         if ( z.compareTo( max ) > 0 )
15             max = z; // z is the largest
16
17         return max; // returns the largest object
18     } // end method maximum
19

```

Присвоява X на локалната данна max

Параметър за тип определя
типа на връщаните данни от
метод maximum

Секцията на параметрите за тип определя
за **ВЪЗМОЖНИ ТИПОВЕ** с този метод само
тези, които са **производни на интерфейс
Comparable**

Извиква метод compareTo на
интерфейс Comparable за
сравнение на y и max

Извиква метод compareTo на
интерфейс Comparable за
сравнение на z и max

```
20 public static void main( String args[] )
21 {
22     System.out.printf( "Maximum of %d, %d and %d is %d\n\n", 3, 4, 5,
23         maximum( 3, 4, 5 ) );
24     System.out.printf( "Maximum of %.1f, %.1f and %.1f is %.1f\n\n"
25         6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );
26     System.out.printf( "Maximum of %s, %s and %s is %s\n", "pear",
27         "apple", "orange", maximum( "pear", "apple", "orange" ) );
28 } // end main
29 } // end class MaximumTest
```

Извиква `maximum` с три цели числа

Извиква `maximum` с три числа в плаваща запетая

Извиква `maximum` с три текстови низа

```
Maximum of 3, 4 and 5 is 5
Maximum of 6.6, 8.8 and 7.7 is 8.8
Maximum of pear, apple and orange is pear
```

12a.4 Параметризирани по тип класове и интерфейси

Горна граница за параметър за тип

- По подразбиране **горната граница** е `Object`
- При ограничение **extends** **горната граница** на Fig. 12a.5 е интерфейс `Comparable`
- За **дефиниране на** **горна граница** използваме **extends**
T extends < T >
- При компилация на параметризиран по тип метод в байткод
 - Параметърът за тип се **замества с горната му граница**
 - **Вмъква се явно преобразуване на типа** на местата, където методът се извиква (т.е. *ред 23 от Fig. 12a.5 се предхожда от явно преобразуване към `Integer` от вида*
`(Integer) maximum(3, 4, 5)`

```
1 public static Comparable maximum(Comparable x, Comparable y, Comparable z)
2 {
3     Comparable max = x; // assume x is initially the largest
4
5     if ( y.compareTo( max ) > 0 )
6         max = y; // y is the largest so far
7
8     if ( z.compareTo( max ) > 0 )
9         max = z; // z is the largest
10
11     return max; // returns the largest object
12 } // end method maximum
```

Видът на
`maximum()`
след компилиране

Изтриването при трансляция от
компилятора замества параметъра
за тип `T` с горната граница
`Comparable`

Изтриването при трансляция от
компилятора замества параметъра
за тип `T` с горната граница
`Comparable`

12a.5 Забранени операции при параметризиране по тип

Следните операции са забранени при параметри за тип:

```
public class MyClass<E> {  
    public static void myMethod(Object item) {  
        if (item instanceof E) { // Compiler error  
            ...  
        }  
        E item2 = new E(); // Compiler error  
        E[] iArray = new E[10]; // Compiler error  
        // Unchecked cast warning  
        // Recompile with -Xlint  
        E obj = (E) new Object();  
    }  
}
```

12a.5 Забранени операции при параметризиране по тип

Неправилно създаване на параметризиран по тип масив

```
ArrayList<Integer> [] intStack =  
    new ArrayList<Integer>[10];
```

Заобиколен начин за създаване на масив с параметър за тип на елементите му е следния:

```
E[] elements = (E[])new Object[capacity];  
// causes an unchecked compile warning  
// This type of compile warning  
// is a limitation of Java generics  
// and is unavoidable
```

12a.5 Забранени операции при параметризиране по тип

Най- чисто е да се създаде масив по обичайния начин

```
ArrayList[] list = new ArrayList[2];
```

Тогава може да пишем

```
list[0] = new ArrayList<Integer>();  
// но... хвърля предупреждение  
list[0].add(2);  
// като може  
//и да се добави въпреки ArrayList<Integer>  
list[0].add("abc");  
System.out.println(list[0]);  
// Извежда: [2, abc]  
// може също  
list[1] = new ArrayList<String>();  
// но тогава list не подлежи на сортиране !!!
```



Защо не може да се създаде масив с параметър за тип

Не е позволено да се създават масиви с параметри за тип на елементите понеже масивът съдържа конкретна информация за типа на елементите си по време на изпълнение. Тази информация се използва по време на изпълнение като се хвърля изключение **ArrayStoreException** в случай, че типът на елементите на масива не съвпада с типа, използван за деклариране на масива. Понеже информацията за стойността на параметъра за тип се изтрива по време на изпълнение, проверката за тип няма как да се извърши. По тази причина, ако беше позволено да се създават масиви с параметър за тип на елементите, поради изтриването на конкретната стойност на параметъра за тип, бихме пропуснали да получим **ArrayStoreException** дори и когато типът на елементите на масива няма наследствена връзка с типа на присвояваната им стойност.

Така например, не бихме различили **ArrayList<Integer>[]**, **ArrayList[]** и **ArrayList<Double>[]**.

Да разгледаме примера на следващия слайд:

Защо не може да се създаде масив с параметър за тип

```
// compile error in the following command
ArrayList<Integer>[] intList = new ArrayList<Integer>[5];
// assume it is OK to create arrays with generic params
// every element of intList is ArrayList of Object
Object[] objArray = intList; // OK to execute
// every element of objArray is ArrayList of Object
// create ArrayList of Double
ArrayList<Double> doubleList = new ArrayList<Double>();
doubleList.add(Double.valueOf(1.23));
//attempt to assign
// an ArrayList of Double-s to ArrayList of Integer-s
objArray[0] = doubleList; // this should fail
// but it would pass because at runtime intList and
// doubleList both are just ArrayList of Objects
```

Обичайна грешка при програмиране

Нека

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

```
ArrayList<Integer> list2 = new ArrayList<>();
```

Въпреки, че **ArrayList<String>** и **ArrayList<Integer>** са два различни типа по време на компилацията, единствено един

клас **ArrayList** се зарежда от JVM по време на изпълнението. Както **list1**, така и **list2** са инстанции на **ArrayList**, затова следните команди са коректни:

```
System.out.println(list1 instanceof ArrayList);
```

```
System.out.println(list2 instanceof ArrayList);
```

Същевременно, командата **list1 instanceof ArrayList<String>** води до грешка при компилация. Понеже

ArrayList<String> се съществува като отделен клас за JVM, то използването му по време на изпълнение няма смисъл.

Обичайна грешка при програмиране

Нека

```
ArrayList<String> list = new ArrayList<>();
```

Проверката за съдържание реферирано с **list**

```
list instanceof ArrayList<String>
```

е синтактична грешка.

Обичайна грешка при програмиране

Не се допуска рефериране на статична данна или статичен метод като се използва **нестатичен параметър** за тип на параметризиран клас. **Разрешено е** да се използва единствено името на класа

```
int m = Counted.MAX;           // ok
int k = Counted <Long>.MAX;    // error
int n = Counted <?>.MAX;      // error

public class Test<E> {
    public static void m(E o1) { // Illegal
    }
    public static <Y> void set(Y t) { } //Legal
    public static E o1; // Illegal

    static {
        E o2; // Illegal
    }
}
```

12a.5a Преодоляване на забранени операции при параметризиране

Ако наистина има нужда да се ползва съдържанието на параметъра за тип по време на изпълнение, то това съдържание трябва да се предаде явно на съответния метод.

Съществуват 3 техники за предаване на съдържанието на параметъра за тип по време на изпълнение:

- Чрез **предаване на object** от типа на параметъра
- Чрез **предаване на масив** с елементи от типа на параметъра
- Чрез **предаване на Class обект**, представящ типа на параметъра

12a.5a Преодоляване на забранени операции при параметризиране

```
public static <T> void someMethod( T dummy) {  
    Class<T> type = dummy.getClass();  
    //... use type reflectively ...  
}  
  
public static <T> void someMethod( T[] dummy) {  
    //... use type reflectively ...  
    Class<T> type = dummy.getClass().getComponentType();  
}  
  
public static <T> void someMethod( Class<T> type) {  
    //... use type reflectively ...  
    //... (T) type.newInstance() ...  
    //... (T[]) Array.newInstance(type, SIZE) ...  
    //... type.isInstance(ref) ...  
    //... type.cast(tmp) ...  
}
```

12a.5a Преодоляване на забранени операции при параметризиране

Следните **способи са допустими**, но хвърлят предупреждение

```
ArrayList<Integer>[] intList =new ArrayList[15];
```

или

```
ArrayList<Integer> tempList =  
                                new ArrayList<Integer>();
```

```
ArrayList<Integer> [] intList =  
    (ArrayList<Integer> [])
```

```
        Array.newInstance(tempStack .getClass(), 10);
```

при последния способ е нужен

```
import java.lang.reflect.Array;
```

Тези **способи се използват**, ако е нужно **сортиране на масива, защото гарантират еднакъв тип за елементите му**. За да се освободите от **ненужни предупреждения** вмъкнете преди началото на метода

```
@SuppressWarnings( "unchecked" )
```



12a.5a Преодоляване на забранени операции при параметризиране

Опцията `-Xlint:unchecked` при компилиране

- Компиляторът в даден случай не може да гарантира 100% сигурност за спазване на типа
- Понеже **произволен обект може да се запише в масив от тип `Object`**, а компилаторът проверява за спазване на типа на масива (*в общия случай различен от `Object`*), указан с параметрите за тип , то се извежда **предупреждение за вероятна грешка** и изисква да се използва опцията

```
javac -Xlint:unchecked GenericClass.java
```

```
/**  
 * This version introduces a generic method.  
 */
```

```
public class Box<T> {
```

```
    private T t;
```

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

```
    public T get() {  
        return t;  
    }
```

```
    public <U> void inspect(U u) {  
        System.out.println("T: " + t.getClass().getName());  
        System.out.println("U: " + u.getClass().getName());  
    }
```

```
    public static void main(String[] args) {  
        Box<Integer> integerBox = new Box<Integer>();  
        integerBox.add(new Integer(10));  
        integerBox.inspect("some text");  
    }  
}
```

Пораждащ клас с параметър за тип T

параметризиран по тип метод с
параметър за тип U

Резултатът от изпълнението на програмата е:
T: java.lang.Integer
U: java.lang.String

```
public class TestGenerics {  
    public static void main(String[] args) {  
        A<String> a = new A<>("A");  
        a.method(2);  
    }  
}
```

```
class A<U>{
```

```
    U e;
```

```
    A(U u) {
```

```
        e = u;
```

```
    }
```

```
    <U> void method(U x) {
```

```
        System.out.println( x.getClass().getName() );
```

```
        System.out.println( e.getClass().getName() );
```

```
    }
```

```
}
```

параметризиран клас с параметър за тип
U

параметризиран метод с параметър за
тип U

Резултатът от изпълнението на програмата е

T: java.lang.Integer

U: java.lang.String

12a.6 Параметризиране с повече от една горна граница

За дефиниране на **повече от един интерфейс или клас** имплементирани като **горна граница** използваме символа **&** както в следния пример:

```
<U extends Number & MyInterface1 & MyInterface2 >
```

```
/**
 * This version introduces a bounded type parameter.
 */
public class Box<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

    public <U extends Number&Serializable> void inspect(U u) {
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text"); // error: this is
                                         // still String!
    }
}
```

Горна граница, позволяваща да се имплементира повече от един интерфейс

12a.7 Типове данни с подразбиращи се (raw) параметри за тип

Необработен тип – *дефиниция*

- Позволява да се създаде пораждащ клас без задаване на реална стойност за параметър за тип

т.е. може да пишем

```
ArrayList objectList = new ArrayList( );
```

- **objectList** се разглежда като *необработен тип данна*
- създава `ArrayList`, които може да *съхранява обекти от произволен тип* (клас)
- Позволява обратна *съвместимост с по-стари версии на езика*, които не използват параметризирани по тип класове

12a.7 Типове данни с подразбиращи се (raw) параметри за тип

Необработен тип - *приложение*

- На променлива от необработен тип `ArrayList` може да се присвои `ArrayList` обект, дефиниран с конкретна стойност за параметъра на типа

```
ArrayList rawTypeList2 = new ArrayList<Double>();
```

- На `ArrayList` променлива с конкретна стойност на параметъра за тип може да се присвои обект от необработен тип `ArrayList`

```
ArrayList< Integer > integerList = new ArrayList();
```

- Разрешено присвояване, но **не се третира като “сигурно”**
- `ArrayList` от **необработен тип** може да съхранява и данни **различни** от `Integer`.
- Изисква опция `-Xlint:unchecked` при компилиране

12a.7a Примери

В **JDK 7** се използва оператора `<>`

```
ArrayList< Box > boxList = new ArrayList <>();
```

което е еквивалентно на

```
ArrayList< Box > boxList = new ArrayList< Box >();
```

т.е компилатора се “*досеца*”, че `ArrayList<>()` е

`ArrayList< Box >` , а не **суровия** тип `ArrayList`.

12a.7a Примери

Силно средство на езика

- Илюстрираме с нова структура данни `ArrayList`
- `java.util.ArrayList`
- Динамично променяне на дължината
- Позволява съхраняване на различен тип данни, дефиниран с параметър за тип
- Директен достъп до данните, аналогично на масив

`ArrayList`

- `set()` и `get()` методи за достъп да данните
- Методи за разширяване- `add()` и `remove()`
- Метод `toString()`

```

1 // Fig. 12b.12: RawTypeTest.java
2 // Raw type test program.
3 import java.util.ArrayList;
4 public class RawTypeTest
5 {
6     private Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
7     private Integer[] integerElements =
8         { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
9
10    // method to test stacks with raw types
11    public void testLists()
12    {
13        // ArrayList of raw types assigned to stack of raw types variable
14        ArrayList rawTypeList1 = new ArrayList ( );
15
16        // ArrayList< Double > assigned to ArrayList of raw types variable
17        ArrayList rawTypeList2 = new ArrayList< Double >( );
18
19        // Stack of raw types assigned to Stack< Integer > variable
20        ArrayList< Integer > integerList = new ArrayList();
21
22        testAdd( "rawTypeList1", rawTypeList1, doubleElements );
23        testDel( "rawTypeList1", rawTypeList1 );
24        testAdd ( "rawTypeList2", rawTypeList2, doubleElements );
25        testDel ( "rawTypeList2", rawTypeList2 );
26        testAdd ( "integerList", integerList, integerElements );
27        testDel ( "integerList", integerList );
28    } // end method testLists
29

```

Създава обект
ArrayList от
необработен тип

Присвоява ArrayList<
Double > на променливата
rawTypeList2

Присвоява ArrayList от
необработен тип на
ArrayList< Integer >
променлива.

Присвояванията тук са разрешени, но дават несигурен код- ArrayList от необработен тип може да съхранява произволни обекти различни от Integer

```
30 // generic method adds elements to an ArrayList
31 public < T > void testAdd( String name, ArrayList< T > list,
32     T[] elements )
33 {
34     // Add elements to an ArrayList
35
36
37     System.out.printf( "\nAdding elements to %s\n", name );
38
39     // add elements to an ArrayList
40     for ( T element : elements )
41     {
42         System.out.printf( "%s ", element );
43         list.add( element ); // add elements
44     } // end for
45 } // end method testAdd
46
47
48
49
50
51
52
```

```
53 // generic method testDel removes data from ArrayList
54 public < T > void testDel( String name, ArrayList< T > list )
55 {
56     // remove elements from list
57
58
59     System.out.printf( "\nRemoving elements from %s\n", name );
60     T popValue; // store element removed from list
61
62     // remove elements from List
63     while ( list.size() > 0 )
64     {
65         popValue = list.remove(list.size()- 1); // remove from list
66         System.out.printf( "%s ", popValue );
67     } // end while
68 } // end method testDel
69
70
71
72
73
74
75
76 public static void main( String args[] )
77 {
78     RawTypeTest application = new RawTypeTest();
79     application.testLists();
80 } // end main
81 } // end class RawTypeTest
```

12a.7a Примери

Необходимост от използване на шаблони

– Пример

- реализация на параметризиран по тип метод **sum**
- Сумира **числови** стойности от дадено множество, представено като **ArrayList**
- **Обектите от числов тип** са производни на **class Number** (базов клас за **Integer** и **Double**)
- Примитивните числови данни ще се **“пакетират”** (**autoboxing**) до **class Number**
- Използва параметър за тип **ArrayList< Number >** за дефиниране на типа на аргумента на метод **sum**
- Използва метод **doubleValue** от **class Number** за преобразуване надолу на обект **Number** до истинската ѝ примитивна стойност **double**

```
1 // Fig. 12b.14: TotalNumbers.java
2 // Summing the elements of an ArrayList.
3 import java.util.ArrayList;
4
5 public class TotalNumbers
6 {
7     public static void main( String args[] )
8     {
9         // create, initialize and output ArrayList of Numbers containing
10        // both Integers and Doubles, then display total of the elements
11        Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
12        ArrayList< Number > numberList = new ArrayList< Number >();
13
14        for ( Number element : numbers )
15            numberList.add( element ); // place each number in numberList
16
17        System.out.printf( "numberList contains: %s\n", numberList );
18        System.out.printf( "Total of the elements in numberList: %.1f\n",
19            sum( numberList ) );
20    } // end main
21
```

1. Декларира и инициализира масив **numbers** от тип **Number**

Стойностите **1, 2** се пакетират като **Integer**
Стойностите **2.4, 4.1** се пакетират като **Double**

2. Декларира и инициализира **numberList**, съхранява **Number** обекти

3. Добавя (метод **add()**) към **ArrayList numberList** елементите на масива **numbers**

4. Изпълнява метода **sum** за пресмятане на елементите от **numberList**

Метод *sum* без използване на шаблон

```
22 // calculate total of ArrayList elements
23 public static double sum( ArrayList< Number > list )
24 {
25     double total = 0; // initialize total
26
27     // calculate sum
28     for ( Number element : list )
29         total += element.doubleValue();
30
31     return total;
32 } // end method sum
33 } // end class TotalNumbers
```

Метод `sum` има аргумент `ArrayList`, съхраняващ `Number` обекти

numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5

Използва метода `doubleValue` на class `Number` за извличане на присвоената примитивна стойност на `Number` обекта като я преобразува до `double`

12a.8 Инвариантност, ковариантност и контравариантност

Инвариантност, ковариантност и контравариантност

Термини, които определят поведението на производни типове данни при преобразованието на типове.

Нека A и B са референтни типове, f е тип на трансформация, а \leq е релацията между типовете (т.е. $A \leq B$ означава, че A е производен тип на B)

Тогава

- **Трансформацията f е ковариантна, ако от $A \leq B$ следва $f(A) \leq f(B)$**
- **Трансформацията f е контравариантна, ако от $A \leq B$ следва $f(B) \leq f(A)$**
- **Трансформацията f е инвариантна, ако нито едно от горните твърдения не е изпълнено**

12a.8 Инвариантност, ковариантност и контравариантност

Примери:

1. Масивите са ковариантни

Ако $A \leq B$ следва $A[] \leq B[]$

т.е. понеже $\text{Integer} \leq \text{Number}$ следва $\text{Integer}[] \leq \text{Number}[]$

(но да си спомним, масивите не е разрешено да се параметризират по тип)

2. Данните, параметризирани по тип са инвариантни

От $\text{Integer} \leq \text{Number}$

не следва,

ниТО $\text{ArrayList} < \text{Integer} > \leq \text{ArrayList} < \text{Number} >$

ниТО $\text{ArrayList} < \text{Number} > \leq \text{ArrayList} < \text{Integer} >$

Защо `List<Number>` не е базов тип за `ArrayList<Integer>`

Параметризирането по тип не поддържа създаване на йерархия от **производни типове**, понеже това би създадо проблеми при постигане на гаранции за неизменност на типа на вече декларирана променлива. По тази причина `ArrayList<BaseType>` не може да се разглежда като базов тип на `ArrayList<SubType>`, където `BaseType` е базов тип за `SubType`.

Пример

```
ArrayList<Long> listLong = new ArrayList<Long>();  
listLong.add(Long.valueOf(100));  
ArrayList<Number> listNumbers = listLong; // compiler error  
listNumbers.add(Double.valueOf(0.50));
```

Извод: Ако параметризирането по тип поддържаше създаване на производни типове, то лесно бихме добавили `Double` към списък от `Long`. Като **следствие** това би довело до `ClassCastException` **по време на изпълнение при обхождане** на списъка от `Long`

12a.8 Инвариантност, ковариантност и контравариантност

Наследственост при използване на пораждане

- Параметризиран клас може да е произведен на не- параметризиран клас, **но има изключения**
т.е., **class Object** е базов клас за всеки параметризиран референтен тип
- Параметризиран клас може да е произведен на друг параметризиран клас
т.е., **class Stack<E>** е произведен на **class Vector<E>**
- Не- параметризиран клас може да е произведен на параметризиран клас
т.е., **Properties** е произведен на **class Hashtable**
- параметризиран по тип метод може в произведен клас може да предефинира метод в базов клас, ако двата метода имат същия *“подпис”*

Обичайна грешка при програмиране

Параметризиран по тип клас не може да наследи **java.lang.Throwable**, понеже следната декларация е забранена:

```
public class MyException<T> extends Exception  
{  
  
}
```

Обичайна грешка при програмиране

Причината е, че бихме имали **catch** от следния вид за **MyException<T>** as follows:

```
try {  
    ...  
}  
  
catch (MyException<T> ex) {  
    ...  
}
```

Тогава JVM трябва да прихване изключение от **try** блока и да го сравни с това описано в **catch** блока. Това, обаче, е невъзможно, понеже по време на изпълнението липсва информация за стойността на типа **T**.

12a.8 Инвариантност, ковариантност и контравариантност

Пример:

Може да се присвои `Integer` на `Object`, понеже `Object` е базов клас за `Integer` :

```
Object someObject = new Object();  
Integer someInteger = new Integer(10);  
someObject = someInteger; // ОК
```

Това в ООП технологията се нарича "**is a**" релация

12a.8a Пример

Вариант, при който метод

```
public static double sum( ArrayList< Number > list )
```

се реализира с шаблон за параметъра на типа на аргумента, който да позволява сумиране на

```
ArrayList< Integer >, ArrayList< Double>
```

- **Number** е базов клас за **Integer**

- **ArrayList< Number >**

не е базов клас на

```
ArrayList< Integer >
```

Ограничение:

Не позволява да подадем аргумент от тип

```
ArrayList< Integer > на метод sum
```

12a.9 Параметър ? за означаване на неизвестен тип

Използваме шаблони за създаване на по-гъвкава схема за реализация на метода `sum`

- `ArrayList< ? extends Number >`
- Шаблонът `?` означава “**неизвестен тип**”
- `extends` ограничава неизвестният тип да е произведен на `Number` или самия клас `Number`
- **Не можем да използваме шаблон** за параметър на тип в тялото на метода

Метод *sum* с използване на шаблон

```
1 // Fig. 12b.15: wildcardTest.java
2 // wildcard test program.
3 import java.util.ArrayList;
4
5 public class wildcardTest
6 {
7     public static void main( String args[] )
8     {
9         // create, initialize and output ArrayList of Integers, then
10        // display total of the elements
11        Integer[] integers = { 1, 2, 3, 4, 5 };
12        ArrayList< Integer > integerList = new ArrayList< Integer >();
13
14        // insert elements in integerList
15        for ( Integer element : integers )
16            integerList.add( element );
17
18        System.out.printf( "integerList contains: %s\n", integerList );
19        System.out.printf( "Total of the elements in integerList: %.0f\n\n",
20            sum( integerList ) );
21
22        // create, initialize and output ArrayList of Doubles, then
23        // display total of the elements
24        Double[] doubles = { 1.1, 3.3, 5.5 };
25        ArrayList< Double > doubleList = new ArrayList< Double >();
26
27        // insert elements in doubleList
28        for ( Double element : doubles )
29            doubleList.add( element );
30
```

Декларираме и инициализираме
ArrayList integerList с
Integer обекти

Извикваме sum за
пресмятане на сума от цели
числа представени с
integerList

Декларираме и инициализираме
ArrayList integerList с
Double обекти Double

Метод *sum* с използване на шаблон

```
31 System.out.printf( "doubleList contains: %s\n", doubleList );
32 System.out.printf( "Total of the elements in doubleList: %.1f\n\n",
33     sum( doubleList ) );
34
35 // create, initialize and output ArrayList of Numbers containing
36 // both Integers and Doubles, then display total of the elements
37 Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
38 ArrayList< Number > numberList = new ArrayList< Number >();
39
40 // insert elements in numberList
41 for ( Number element : numbers )
42     numberList.add( element );
43
44 System.out.printf( "numberList contains: %s\n", numberList );
45 System.out.printf( "Total of the elements in numberList: %.1f\n",
46     sum( numberList ) );
47 } // end main
48
49 // calculate total of stack elements
50 public static double sum( ArrayList< ? extends Number > list )
51 {
52     double total = 0; // initialize total
53 }
```

Извиква `sum` за сумиране на `doubleList`

Декларира и създава `ArrayList integerList` с обекти от клас `Number`

Изпълнява метод `sum` за пресмятане на сумата на елементите в `numberList`

Типът на `ArrayList` в метод `sum` не е непосредствено зададен, знае се само горната граница за този тип, че е `Number`

```
54 // calculate sum
55 for ( Number element : list )
56     total += element.doubleValue();
57
58     return total;
59 } // end method sum
60 } // end class WildcardTest
```

```
integerList contains: [1, 2, 3, 4, 5]
Total of the elements in integerList: 15
```

```
doubleList contains: [1.1, 3.3, 5.5]
Total of the elements in doubleList: 9.9
```

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

В тялото на метода използваме горната
граница **Number** за типа, а не шаблона

Обичайна грешка при програмиране

Използването на **шаблон в секцията да деклариране на параметри за тип** или използването на **шаблон** вместо явно деклариране на тип на променлива **в тялото на метод** е **синтактична грешка**.

12a.9 Параметър ? за означаване на неизвестен тип

Пример:

Понеже `Integer` е производен на `Number`, то следното е също правилно:

```
public void someMethod(Number n) {  
    // method body omitted  
}
```

```
someMethod(new Integer(10)); // OK
```

```
someMethod(new Double(10.1)); // OK
```

12a.9 Параметър ? за означаване на неизвестен тип

Пример:

Аналогични конструкции са възможни с параметризирани по тип класове.

Например, при `Number` за стойност на параметър за тип , може да напишем следното :

```
Box<Number> box = new Box<Number>();
```

```
box.add(new Integer(10)); // OK
```

```
box.add(new Double(10.1)); // OK
```

понеже `Integer` и `Double` са **производни** на `Number`

12a.9 Параметър ? за означаване на неизвестен тип

Пример:

Нека сега разгледаме метода:

```
public void boxTest(Box<Number> n) {  
    // method body omitted  
}
```

Какъв е допустимият тип за аргумент на този метод?

При разглеждане на подписа на метода, разбираме това е **Box<Number>**.

Но дали това позволява да приеме за аргумент също **Box<Integer>** или **Box<Double>**

Отговорът е “НЕ” (invariance). понеже **Box<Integer>** и **Box<Double>** не са производни на **Box<Number>**

12a.9 Параметър ? за означаване на неизвестен тип

Пример:

Box<Integer> и Box<Double> обаче **са производни** на
Box<? **extends Number**>
където **Number** е *горна граница* за параметъра за тип на Box

Тогава

```
public void boxTest(Box<? extends Number > n) {  
    // method body omitted  
}
```

позволява аргументи от тип Box<Integer>и Box<Double>

12a.10 Долна граница за параметър

Пример:

Възможно е **да се зададе долна граница**, чрез използване на ключовата дума **super** вместо **extends**.

Параметър за тип зададен като

`<? super TwoDShape >`

се чете като *“неизвестен тип, който е базов за TwoDShape, или самият клас TwoDShape.”*

За **допълнителна информация** относно *generics* (параметризирани типове с методи и класове) четете на

<http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>

12a.10 Долна граница за параметър

```

1  public class GenericStack<E> {                                generic type E declared
2      private java.util.ArrayList<E> list = new java.util.ArrayList<>();    generic array list
3
4      public int getSize() {                                     getSize
5          return list.size();
6      }
7
8      public E peek() {                                         peek
9          return list.get(getSize() - 1);
10     }
11
12     public void push(E o) {                                     push
13         list.add(o);
14     }
15
16     public E pop() {                                           pop
17         E o = list.get(getSize() - 1);
18         list.remove(getSize() - 1);
19         return o;
20     }
21
22     public boolean isEmpty() {                                   isEmpty
23         return list.isEmpty();
24     }
25
26     @Override
27     public String toString() {
28         return "stack: " + list.toString();
29     }
30 }

```

12a.10 Долна граница за параметър

If `<? super T>` is replaced by `<T>`, a compile error will occur on `add(stack1, stack2)` in line 8, because `stack1`'s type is `GenericStack<String>` and `stack2`'s type is `GenericStack<Object>`. `<? super T>` represents type `T` or a **supertype** of `T`. `Object` is a **supertype** of `String`.

```

1  public class SuperWildCardDemo {
2      public static void main(String[] args) {
3          GenericStack<String> stack1 = new GenericStack<>();
4          GenericStack<Object> stack2 = new GenericStack<>();
5          stack2.push("Java");
6          stack2.push(2);
7          stack1.push("Sun");
8          add(stack1, stack2);
9          AnyWildCardDemo.print(stack2);
10     }
11
12     public static <T> void add(GenericStack<T> stack1,
13         GenericStack<? super T> stack2) {
14         while (!stack1.isEmpty())
15             stack2.push(stack1.pop());
16     }
17 }
```

12a.11 Принципът Get- Put

Принцип:

Използвайте **extends**, когато прочитате (при **get** операция) стойност от структура данни (Producer)

Използвайте **super**, когато присвоявате (при **put** операция) стойност от структура данни (Consumer)

Не се използва **extends** и **super** едновременно прочитане и присвояване на стойност

Пример:

```
public static <T> void copy(Stack<? super T> dest, // put  
                           Stack<? extends T> src) // get
```

Този метод **прочита** стойностите на параметъра **src** и затова се използва **extends**. Същевременно резултатът от изпълнението му се **присвоява** (записва) в параметъра **dst** и затова се използва **super**.

12a.11 Принципът Get- Put

Пример:

```
public static double sum(ArrayList<? extends Number> nums)
{
    double s = 0.0;
    for (Number num : nums) s += num.doubleValue();
    return s;
}
```

```
ArrayList<Integer> ints = new ArrayList<>();
sum(ints) == 6.0; // legal GET
```

```
ArrayList<Double> doubles = new ArrayList<>();
sum(doubles) == 5.92; // legal GET
```

```
ArrayList<Number> nums = new ArrayList<>();
sum(nums) == 8.92; // legal GET
```

12a.11 Принципът Get- Put

Пример:

```
public static void count(ArrayList<? super Integer> ints, int  
n) {  
    for (int i = 0; i < n; i++) ints.add(i);  
}
```

Винаги, когато се използва метода **add()**, то стойности се присвояват (записват) в структурата данни и трябва да се ползва **super**

```
ArrayList<Integer> ints = new ArrayList<>();  
count(ints, 6); ints.add(6); // legal PUT  
ArrayList<Number> doubles = new ArrayList<>();  
count(doubles, 6); doubles.add(6.5) // legal PUT  
ArrayList<Object> objs= new ArrayList<>();  
count(objs, 6); objs.add("six"); // legal PUT
```

12a.11 Принципът Get- Put

Пример:

```
public static double sumCount(ArrayList<Number> nums,  
                                int n) {  
    count(nums, n);  
    return sum(nums);  
}
```

Тук не може да се ползват **extends** и **super** защото се извършва едновременно прочитане и присвояване на стойност

12a.11 Принципът Get- Put

Важно Правило:

Не е разрешено да се добавят елементи към колекция(списък) декларирана с шаблона ? **extends**

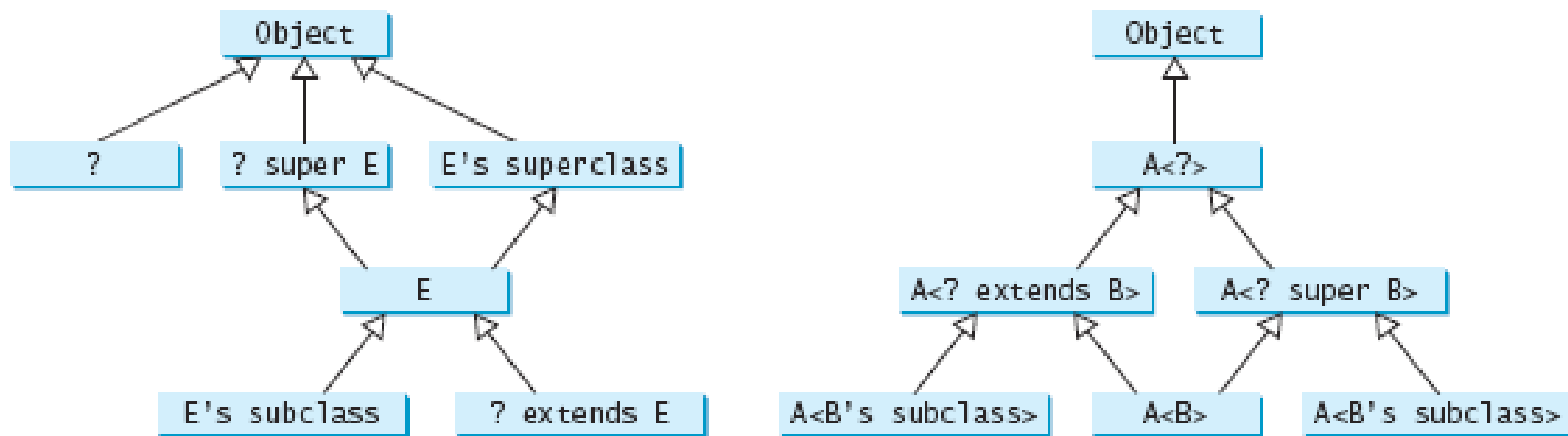
(в този случай е разрешено само четене на елементите, GET)

Например,

```
ArrayList<? extends Number> numbers = new ArrayList<Integer>();  
numbers.add(123); // COMPILE ERROR  
// but this is allowed  
numbers.add(null); // OK
```


12a.11 Принципът Get- Put

Обобщение:



Тук **A** и **B** са класове или интерфейси , а **E** е параметър за тип

Задачи

Задача 1.

Нека са дадени следните класове:

```
public class AnimalHouse<E> {  
    private E animal;  
    public void setAnimal(E x) {  
        animal = x;  
    }  
    public E getAnimal() {  
        return animal;  
    }  
}  
  
public class Animal{  
}  
public class Cat extends Animal {  
}  
  
public class Dog extends Animal {  
}
```

Задачи

За всяка от следните команди :

```
AnimalHouse<Animal> house = new AnimalHouse<Cat>();
```

```
AnimalHouse<Dog> house = new AnimalHouse<Animal>();
```

```
AnimalHouse<?> house = new AnimalHouse<Cat>(); house.setAnimal(new Cat());
```

```
AnimalHouse house = new AnimalHouse(); house.setAnimal(new Dog());
```

Определете дали

- a) Има синтактична грешка,
- b) Компилира се с предупреждение,
- c) Води до грешка апо време на изпълнение
- d) Нито едно от горните (компилира се и се изпълнява без грешка)

Задача 2.

Напишете клас, който служи за библиотека на три средства за информация: книги, видео и вестници. Библиотеката трябва да може да добавя и извлича информация

Напишете версия на класа, който използва пораждане и друга версия без пораждане.

Упътване: Използвайте `ArrayList` за реализиране на библиотеката по принципа на композицията и потребителски дефинирани интерфейси и наследственост за реализиране на методи за добавяне и извличане на информация от библиотеката аналогично на дефиницията на `ListComposition`, даден на лекции

Задачи

Задача 3.

Презаредете пораждащия метод `printArray` от Fig. 12a.3 така че да използва **два допълнителни аргумента**, `lowSubscript` и `highSubscript`. При извикването на този метод да се **изведе на стандартен изход само тези елементи, намиращи се между посочените индекси** с аргументите `lowSubscript` и `highSubscript`.

Проверете `lowSubscript` и `highSubscript`- ако са извън допустимите стойности за индекси на масива, или ако `highSubscript` по- малък или равен на `lowSubscript`, презареденият метод `printArray` да хвърля `InvalidSubscriptException`; в противен случай, `printArray` връща бройт на отпечатаните елементи от масива. Променете също методът `main` за тестване на метод `printArray` с масиви `integerArray`, `doubleArray` и `characterArray`. Тествайте всички възможни ситуации при извикване на `printArray`

Задачи

Задача 4.

Презаредете метода `printArray` от Fig. 12a.3 с версия на не – параметризиран по тип метод , която позволява да се изпълни при задаване на масив от текстови низове и в този случай да отпечата елементите на този масив подредени в колони както е показано по- долу:

<code>Array</code>	<code>stringArray</code>	съдържа:	
<code>one</code>	<code>two</code>	<code>three</code>	<code>four</code>
<code>five</code>	<code>six</code>	<code>seven</code>	<code>eight</code>

Задачи

Задача 5.

Напишете версия на параметризиран по тип метод `isEqualTo()` която сравнява двата аргумента с метода `equals()` и връща **true** ако са равни и **false** в противен случай. Използвайте този параметризиран по тип метод в програма, която извиква `isEqualTo` с множество от библиотечно дефинирани класове като `Object` или `Integer`. Какво се получава при изпълнението на тази програма?

Задача 6.

Напишете пораждащ `class Pair`, който има две данни, чиито тип се определя от два параметъра за тип `F` и `S`. Напишете `get` и `set` методи за тези данни на класа .

Упътване:

Заглавието на класа ще бъде

```
public class Pair< F, S >
```