

# Лекция 8а

## Параметризиране по тип на методи и класове (*generics*)

# Основни теми

- Създаване на прототипове на методи, за извършване на идентични действия с аргументи от различен тип.
- Създаване на параметризиран шаблон на `Stack` клас, който може да представя в свързан списък обекти производни на произволен клас или интерфейс.
- Презареждане на Параметризирани методи с други Параметризирани или не-Параметризирани методи.
- Дефиниране на примитивни типове за пораждање и реализиране на обратна съвместимост.
- Използване на шаблони, когато не е нужна точна информация за аргумент на метод в тялото на метода.
- Връзката между използване на Параметризираните прототипове и наследствеността

- 8a.1 Въведение**
- 8a.2 Причини за използване на Параметризирани прототипове**
- 8a.3 Параметризирани методи: Реализация и транслиране по време на компилация**
- 8a.4 Особености при компилация**
- 8a.5 Презареждане на Параметризирани прототипове на методи**
- 8a.6 Параметризирани прототипове на класове**
- 8a.7 Необработен (суров) тип данни**
- 8a.8 Шаблони за параметри на тип като горна и долна граница на параметър**
- 8a.9 Параметризиране и наследственост : Особености**  
**Задачи**

**Литература:**

**Н. М. Deitel, Р. J. Deitel *Java How to Program, 10 Edition, глава 20***

# 8a.1 Introduction

## Параметризирани прототипове (*generics*)

- Въведени 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 прототип на 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 прототип на клас*

# 8a.1 Introduction

## Параметризирани прототипове (*generics*)

- Позволява да се установи по време на компилация неправилно използване на параметризирания прототип- *проверка на типа при компилация* (compile-time type safety)

Например, ако клас Stack е параметризиран по тип клас, използван за съхраняване на цели числа, то при опит да се постави String в този Stack се извежда грешка при компилация.

# 8a.1 Introduction

## Параметризирани прототипове (*обобщение*)

- Параметризираните прототипове на методи и класове позволяват да се опишат с една единствена дефиниция на метод, цяло множество от методи, респективно- с една единствена дефиниция на клас се описва цяло множество от класове. (**generics**)
- Параметризираните прототипове позволяват да се прихващат грешки в използвания прототип по време на компилация (**compile-time type safety**)
- Проекти на Java с параметризирани типове се компилират с опция `-Xlint:unchecked` на компилатора

# Software Engineering факти 8a.1

---

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

# 8a.2 Необходимост от Параметризирани прототипове

## 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 методи и класове

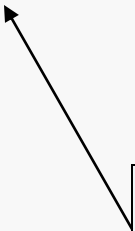


```
1 // Fig. 18.1: OverloadedMethods.java
2 // Using overloaded methods to print array of different types.
3
4 public class OverloadedMethods
5 {
6     // method printArray to print Integer array
7     public static void printArray( Integer[] inputArray )
8     {
9         // display array elements
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)

## 8a.2 Необходимост от Параметризирани прототипове

### 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 ги прототип** като

- Заменяме името на типа на данните в заглавието на метода и навсякъде където типът на тези данни се използва в метода с **параметризирано типа** име `E`
- 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. 8a.2 |** Методът `printArray` в който всички действителни типове данни са заменени с параметризираното име за тип **E**.

# Software Engineering факти 8a.1

---

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

## 8a.3 Параметризирани методи: Реализация и компиляция

**В случаите, аналогични на Fig. 8a.1, когато презаредените методи са идентични за множество от типове данни е за предпочитане да се използва параметризиран прототип на метода**

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

## 8a.3 Параметризирани методи: Реализация и компиляция

Реализираме Fig. 8a.1, като параметризиран прототип на метод

Декларация на параметризиран метод

- **Секция за описание 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**



## 8a.3 Параметризирани методи: Реализация и компилация

### Параметър за тип

- Нарича се **променлива за тип**
- **Идентификатор** задаващ име за параметризиран 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 параметризирания 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 )
```

## Обичайна грешка при програмиране 8a.1

---

**Пропускането да се постави секцията за описание на типа на параметрите на параметризирания метод преди типа на данните, връщани от метода, води до грешка при компилация.**

```
1 // Fig. 18.3: 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 метод `printArray`

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

Използваме параметъра за тип при деклариране 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 параметризирания  
метод `printArray` с `Integer`  
масив

Изпълнение на параметризирания  
метод `printArray` с `Double`  
масив

Изпълнение на параметризирания  
метод `printArray` с `Character`  
масив

## Обичайна грешка при програмиране 8а.2

---

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

## Обичайна грешка при програмиране 8а.3

---

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

## 8a.3 Параметризирани методи: Реализация и компилация

### Транслиране на Параметризирани методи при компилация

- Извършва се изтриване
  - Изтрива се секцията за деклариране на параметрите за тип
  - Замества параметрите за тип с реалните типове данни
  - Подразбиращият се тип за заместване на всички параметри 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.

## Правила за добро програмиране 8a.1

---

**Не използвайте параметър за тип в прихващане на изключение в `catch` условието поради преобразуването на тип при изтриване в процеса на компилация.**



```
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 параметризирана версия

**Fig. 8a.4** | параметризираният метод `printArray` след като изтриването е изпълнено от компилатора.

## 8a.4 Допълнителни особености при транслиране: Методи използващи параметър за тип на връщаните данни

Примерът е даден на Fig. 8a.5

- параметризиран 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 '>', '<', '==', '<=', '>='

## 8a.4 Допълнителни особености при транслиране: Методи използващи параметър за тип на връщаните данни

### **interface** с параметризиран 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

## 8а.4 Допълнителни особености при транслиране: Методи използващи параметър за тип на връщаните данни

### 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**

```

1 // Fig. 18.5: 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

## 8a.4 Допълнителни особености при транслиране: Методи използващи параметър за тип на връщаните данни

### Горна граница за параметъра за тип

- По подразбиране **горната граница** е `Object`
- В дадения пример това е интерфейс `Comparable`
- За **дефиниране на горна граница** използваме **`extends`**

т.е.,

`T extends < T >`

- При транслиране на параметризиран 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. 8a.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`



## 8a.5 Презареждане на параметризиран метод

Параметризираните методи могат да се презареждат (overload)

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

## 8a.5 Презареждане на параметризиран метод

**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**

## 8a.5 Презареждане на параметризиран метод

### Пример 1:

параметризираният метод `printArray` от Fig. 8a.3 може да се **презареде с друг параметризиран метод `printArray`**, който има **допълнителни аргументи `lowSubscript` и `highSubscript`**, задаващи подмножество от елементи за извеждане на печат

## 8a.5 Презареждане на параметризиран метод

### Пример 2:

параметризираният метод `printArray` от Fig. 8a.3 може да се **презарежи с друг параметризиран метод `printArray` от Fig. 8a.3** с версия специфична за `String` обекти, при което тези обекти се извеждат на печат в колонки

## 8а.6 Параметризирани класове

### Параметризирани класове

- Концепцията за структура от данни като Stack например, може да се разбере **независимо от данните които съхранява.**
- Параметризираните 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: `Stack< Double >`

*("Stack om Double," "Stack om Integer," ..., "Stack om Employee," )*

## 8a.6 Параметризирани класове

**Figure 8a.7** представя дефиниция на параметризиран class `Stack`, където свързани списък е реализиран като се използва масив.

- Името на класа се следва от секцията за деклариране на параметрите за тип (ред 4).
- В този пример, параметърът за **тип E** определя **типа на данните в Stack**.
- Секцията на параметрите за тип може да има един или повече параметри за тип, разделени със запетая
- Параметърът за тип E се използва в тази версия на class `Stack` за деклариране на типа на елементите

```

1 // Fig. 18.7: Stack.java
2 // Generic class Stack.
3
4 public class Stack< E >
5 {
6     private final int size; // number of elements in the stack
7     private int top; // location of the top element
8     private E[] elements; // array that stores stack elements
9
10    // no-argument constructor creates a stack of the default size
11    public Stack()
12    {
13        this( 10 ); // default stack size
14    } // end no-argument Stack constructor
15
16    // constructor creates a stack of the specified number of elements
17    public Stack( int s )
18    {
19        size = s > 0 ? s : 10; // set size of Stack
20        top = -1; // Stack initially empty
21
22        elements = ( E[] ) new Object[ size ]; // create array
23    } // end Stack constructor
24

```

Декларация на параметри з атип на параметризирания клас, името на класа се следва от **секция на параметрите за тип**

Декларира `elements` като масив от тип `E`

Създава масив от тип `E`. **Важно:** Схемата за пораждаване не позволява параметри за тип да участват в създаване на масиви, защото съдържанието на параметърът за тип е недостъпно по време на изпълнение(компиляторът го изтрива)

```
25 // push element onto stack; if successful, return true;
26 // otherwise, throw FullStackException
27 public void push( E pushValue )
28 {
29     if ( top == size - 1 ) // if stack is full
30         throw new FullStackException( String.format(
31             "Stack is full, cannot push %s", pushValue ) );
32
33     elements[ ++top ] = pushValue; // place pushValue on stack
34 } // end method push
35
36 // return the top element if not empty; else throw EmptyStackException
37 public E pop()
38 {
39     if ( top == -1 ) // if stack is empty
40         throw new EmptyStackException( "Stack is empty, cannot pop" );
41
42     return elements[ top-- ]; // remove and return top element of stack
43 } // end method pop
44 } // end class Stack< E >
```

Метод **push** поставя елемент от тип **E** върху стека

Метод **pop** връща елемент от върха на стека, който е от тип **E**



```
1 // Fig. 18.8: FullStackException.java
2 // Indicates a stack is full.
3 public class FullStackException extends RuntimeException
4 {
5     // no-argument constructor
6     public FullStackException()
7     {
8         this( "Stack is full" );
9     } // end no-argument FullStackException constructor
10
11     // one-argument constructor
12     public FullStackException( String exception )
13     {
14         super( exception );
15     } // end one-argument FullStackException constructor
16 } // end class FullStackException
```

```
1 // Fig. 18.9: EmptyStackException.java
2 // Indicates a stack is full.
3 public class EmptyStackException extends RuntimeException
4 {
5     // no-argument constructor
6     public EmptyStackException()
7     {
8         this( "Stack is empty" );
9     } // end no-argument EmptyStackException constructor
10
11     // one-argument constructor
12     public EmptyStackException( String exception )
13     {
14         super( exception );
15     } // end one-argument EmptyStackException constructor
16 } // end class EmptyStackException
```

```

1 // Fig. 18.10: StackTest.java
2 // Stack generic class test program.
3
4 public class StackTest
5 {
6     private double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
7     private int[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
8
9     private Stack< Double > doubleStack; // stack stores Double objects
10    private Stack< Integer > integerStack; // stack stores Integer objects
11
12    // test stack objects
13    public void testStacks()
14    {
15        doubleStack = new Stack< Double >( 5 ); // Stack of Doubles
16        integerStack = new Stack< Integer >( 10 ); // Stack of Integers
17
18        testPushDouble(); // push double onto doubleStack
19        testPopDouble(); // pop from doubleStack
20        testPushInteger(); // push int onto intStack
21        testPopInteger(); // pop from intStack
22    } // end method testStacks
23

```

Реалният параметър за тип на клас **Stack** е **Double**

Реалният параметър за тип на клас **Stack** е **Integer**

Създаваме обект **doubleStack** с размерност 5 и **integerStack** с размерност 10

Води до хвърляне на изключение (нарочно, за тестване на ситуацията)-**push** на клас **Stack** се изпълнява с повече елементи (+1), отколкото масива на **Stack** е дефиниран да съдържа

```
24 // test push method with double stack
25 public void testPushDouble()
26 {
27     // push elements onto stack
28     try
29     {
30         System.out.println( "\nPushing elements onto doubleStack" );
31
32         // push elements to stack
33         for ( double element : doubleElements )
34         {
35             System.out.printf( "%.1f ", element );
36             doubleStack.push( element ); // push onto doubleStack
37         } // end for
38     } // end try
39     catch ( FullStackException fullStackException )
40     {
41         System.err.println();
42         fullStackException.printStackTrace();
43     } // end catch FullStackException
44 } // end method testPushDouble
45
```

Извиква метода **push** на клас **Stack**  
за поставяне на **double** стойност  
върху **doubleStack**

```

46 // test pop method with double stack
47 public void testPopDouble()
48 {
49     // pop elements from stack
50     try
51     {
52         System.out.println( "\nPopping elements from doubleStack" );
53         double popValue; // store element removed from stack
54
55         // remove all elements from stack
56         while ( true )
57         { // pop from doubleStack
58             popValue = doubleStack.pop();
59             System.out.printf( "%.1f ", popValue );
60         } // end while
61     } // end try
62     catch( EmptyStackException emptyStackException )
63     {
64         System.err.println();
65         emptyStackException.printStackTrace();
66     } // end catch EmptyStackException
67 } // end method testPopDouble
68

```

Използва се **неявно преобразуване** до примитивен тип `double` (**auto-unboxing**) при връщане на `pop` (`Double`) метода

След изтриване на параметъра за тип от компилатора, методът `pop` на `Stack` връща тип `Object`, който се преобразува надолу до `Double`

Понеже `testPopDouble` в клиентският код очаква да получи `Double` на излизане от `pop`, компилаторът вмъква явно преобразуване до `Double` като

```
popValue = ( Double )
            doubleStack.pop();
```

```
69 // test push method with integer stack
70 public void testPushInteger()
71 {
72     // push elements to stack
73     try
74     {
75         System.out.println( "\nPushing elements onto intStack" );
76
77         // push elements to stack
78         for ( int element : integerElements )
79         {
80             System.out.printf( "%d ", element );
81             integerStack.push( element ); // push onto integerStack
82         } // end for
83     } // end try
84     catch ( FullStackException fullStackException )
85     {
86         System.err.println();
87         fullStackException.printStackTrace();
88     } // end catch FullStackException
89 } // end method testPushInteger
90
```

Извиква метода `push` на клас `Stack`  
за поставяне на `int` стойност върху  
`integerStack`

```

91 // test pop method with integer stack
92 public void testPopInteger()
93 {
94     // pop elements from stack
95     try
96     {
97         System.out.println( "\nPopping elements from intStack" );
98         int popValue; // store element removed from stack
99
100        // remove all elements from stack
101        while ( true )
102        {
103            popValue = integerStack.pop(); // pop from intStack
104            System.out.printf( "%d ", popValue );
105        } // end while
106    } // end try
107    catch( EmptyStackException emptyStackException )
108    {
109        System.err.println();
110        emptyStackException.printStackTrace();
111    } // end catch EmptyStackException
112 } // end method testPopInteger
113
114 public static void main( String args[] )
115 {
116     StackTest application = new StackTest();
117     application.testStacks();
118 } // end main
119 } // end class StackTest

```

Използва се **неявно преобразуване** до примитивен тип **int** (**auto-unboxing**) при връщане на **pop(Integer)** метода. След изтриване на параметъра за тип от компилатора, методът **pop** на **Stack** връща тип **Object**. Понеже **testPopDouble** в клиентският код очаква да получи **Integer** на излизане от **pop**, компилаторът вмъква явно преобразуване до **Integer** като **popValue = ( Integer ) integerStack.pop();**

Pushing elements onto doubleStack

1.1 2.2 3.3 4.4 5.5 6.6

FullStackException: Stack is full, cannot push 6.6

```
at Stack.push(Stack.java:30)
at StackTest.testPushDouble(StackTest.java:36)
at StackTest.testStacks(StackTest.java:18)
at StackTest.main(StackTest.java:117)
```

Popping elements from doubleStack

5.5 4.4 3.3 2.2 1.1

EmptyStackException: Stack is empty, cannot pop

```
at Stack.pop(Stack.java:40)
at StackTest.testPopDouble(StackTest.java:58)
at StackTest.testStacks(StackTest.java:19)
at StackTest.main(StackTest.java:117)
```

Pushing elements onto integerStack

1 2 3 4 5 6 7 8 9 10 11

FullStackException: Stack is full, cannot push 11

```
at Stack.push(Stack.java:30)
at StackTest.testPushInteger(StackTest.java:81)
at StackTest.testStacks(StackTest.java:20)
at StackTest.main(StackTest.java:117)
```

Popping elements from integerStack

10 9 8 7 6 5 4 3 2 1

EmptyStackException: Stack is empty, cannot pop

```
at Stack.pop(Stack.java:40)
at StackTest.testPopInteger(StackTest.java:103)
at StackTest.testStacks(StackTest.java:21)
at StackTest.main(StackTest.java:117)
```



## 8a.6 Параметризирани класове

Следните операции са забранени при параметри за тип:  
(<http://docs.oracle.com/javase/tutorial/java/generics/restrictions.html>)

```
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();  
    }  
}
```

## Обичайна грешка при програмиране 8a.4

```
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  
        E obj = (E)new Object(); //Unchecked cast warning  
    }  
}
```

## 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 необработения тип 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 E o1; // Illegal

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

# 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 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 **List<Integer>[]**, **List[]** и **List<Double>[]**.

Да разгледаме примера на следващия слайд:

# Защо не може да се създаде масив с параметър за тип

// Допускаме, че следната команда е разрешена в Java

```
List<Integer>[] intList = new List<Integer>[5]; // compile error
```

```
List[] objArray = intList; // List<Integer>[] е List[]
```

```
List<Double> doubleList = new ArrayList<Double>(); // позволено
```

```
doubleList.add(Double.valueOf(1.23));
```

```
objArray[0] = doubleList; /* няма грешка, типът на елементите на
```

```
doubleList не може да се различи от типът на елементите на
```

```
intList. Грешка би трябвало да има, понеже всеки елемент на
```

```
objArray е List<Integer>, а doubleList съдържа Double елементи.
```

Но тази грешка няма да се прихване, ако може да се създават

масиви с параметър за тип на елементите, понеже по време на

изпълнение intList и doubleList са просто List \*/

## 8a.6 Параметризирани класове

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

## 8a.6 Параметризирани класове

```
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) ...  
}
```

## 8a.6 Параметризирани класове

// например създаване на инстанция от тип E изисква

```
private static class SomeContainer<E>
{
    E createContents(Class<E> someClass)
    {
        return someClass.newInstance();
    }
}
```



## 8a.6 Параметризирани класове и масиви

You **cannot** create an array using a generic type parameter

```
E[] elements = new E[capacity];
```

```
Stack<Integer> [] stack = new Stack<Integer>[10];
```

You can use the following code to circumvent this restriction:

```
Stack<Integer>[] stack = (Stack<Integer>[])new Stack[10];
```

However, you will still get a compile warning

## 8a.6 Параметризирани класове и масиви

Най-чисто е да се създаде масив по обичайния начин

```
Stack[] stack = new Stack[2];
```

Тогава може да пишем

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

Извежда:

```
[2, abc]
```

## 8a.6 Параметризирани класове и масиви

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

```
Stack<Integer> [] intStack = new Stack[15];
```

*или*

```
Stack<Integer> tempStack = new Stack<Integer>();  
Stack<Integer> [] intStack =  
    (Stack<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")
```

## 8a.6 параметризирани класове

### 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

## 8a.6 параметризирани класове

### Параметризирани класове

- Концепцията за структура от данни като Box например, може да се разбере **независимо от данните които съхранява.**
- параметризираните 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: `Box< Double >`

*(" BOX от Double," " BOX от Integer," ..., " BOX от 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



## 8a.6 параметризирани класове

За рефериране на параметризирания `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.

## 8a.6 параметризирани класове

За дефиниране на `integerBox` използваме:

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

или декларация и дефиниция заедно

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

## 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

## 8a.6 параметризирани класове

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

## 8a.6 параметризирани класове

**Figure 8a.7** представя дефиниция на параметризиран `class Stack`, където свързани списък е реализиран като се използва масив.

- Името на класа се следва *от секцията за деклариране на параметрите* за тип (ред 4).
- В този **пример**, параметърът за **тип E** определя **типа на данните** в `Stack`.
- Секцията на параметрите за тип **може да има един или повече параметри за тип**, разделени със запетая
- Параметърът за тип `E` се използва в тази версия на `class Stack` за деклариране на типа на елементите

## 8a.6 параметризирани класове

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 Stack.java
```

## 8a.6 параметризирани класове

Опцията `-Xlint:unchecked` Предупреждение при компилиране:

```
Stack.java:22: warning: [unchecked] unchecked cast  
found   : java.lang.Object[] required:
```

```
    E[] elements = ( E[] ) new Object[ size ];  
    // create array
```

Компиляторът не може да гарантира, че всеки от елементите на масива може да е от типа `E`, респ. може да се преобразува до този тип

```
Object[] objectArray = elements;  
// elements е масив от Integer
```

```
objectArray[ 0 ] = "hello";
```

```
// грешка, но компилаторът не може да я прихване
```

## 8a.6 параметризирани класове

**параметризиран клас по време на компилация**

- **Компиляторът изтрива параметрите за тип**
- **Компиляторът замества параметрите за тип с техните горни граници**

**параметризиран клас за тестване по време на компилация**

- **Компиляторът извършва проверки за съответствие на типовете**
- **Компиляторът вмъква команди за явно преобразуване на типовете, ако е необходимо**



## 8a.6 параметризирани класове

Дефиниране на параметризирани методи за тестване на **Stack< E >**

– Метод **testPush**

- Извършва същите операции както **testPushDouble** и **testPushInteger**
- Може да се изпълни вместо кой да е от тези методи

– Метод **testPop**

- Извършва същите операции както **testPopDouble** и **testPopInteger**
- Може да се изпълни вместо кой да е от тези методи

```
1 // Fig. 18.11: StackTest2.java
2 // Stack generic class test program.
3
4 public class StackTest2
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    private Stack< Double > doubleStack; // stack stores Double objects
11    private Stack< Integer > integerStack; // stack stores Integer objects
12
13    // test stack objects
14    public void testStacks()
15    {
16        doubleStack = new Stack< Double >( 5 ); // stack of Doubles
17        integerStack = new Stack< Integer >( 10 ); // stack of Integers
18
19        testPush( "doubleStack", doubleStack, doubleElements );
20        testPop( "doubleStack", doubleStack );
21        testPush( "integerStack", integerStack, integerElements );
22        testPop( "integerStack", integerStack );
23    } // end method testStacks
24
```

Извиква параметризираните методи `testPush` и `testPop` да поставят елементи върху стека и да изтрият елементи от стека

```
25 // generic method testPush pushes elements onto a Stack
26 public < T > void testPush( String name, Stack< T > stack,
27     T[] elements )
28 {
29     // push elements onto stack
30     try
31     {
32         System.out.printf( "\nPushing elements onto %s\n", name );
33
34         // push elements onto stack
35         for ( T element : elements )
36         {
37             System.out.printf( "%s ", element );
38             stack.push( element ); // push element onto stack
39         }
40     } // end try
41     catch ( FullStackException fullStackException )
42     {
43         System.out.println();
44         fullStackException.printStackTrace();
45     } // end catch FullStackException
46 } // end method testPush
47
```

параметризиран метод `testPush` заменя `testPushDouble` и `testPushInteger`

Заменя действителния тип елемента `Double/Integer` с параметър за тип `T`

Забележете, че компилаторът **налага съответствие** между типа на `Stack` и елементите, които се поставят върху `Stack` с метода `push()` – това определя предимството на параметризираните методи

```
48 // generic method testPop pops elements from a Stack
49 public < T > void testPop( String name, Stack< T > stack )
50 {
51     // pop elements from stack
52     try
53     {
54         System.out.printf( "\nPopping elements from %s\n", name );
55         T popValue; // store element removed from stack
56
57         // remove elements from stack
58         while ( true )
59         {
60             popValue = stack.pop(); // pop from stack
61             System.out.printf( "%s ", popValue );
62         } // end while
63     } // end try
64     catch( EmptyStackException emptyStackException )
65     {
66         System.out.println();
67         emptyStackException.printStackTrace();
68     } // end catch EmptyStackException
69 } // end method testPop
70
71 public static void main( String args[] )
72 {
73     StackTest2 application = new StackTest2();
74     application.testStacks();
75 } // end main
76 } // end class StackTest2
```

параметризиран метод testPop заменя  
testPopDouble и testPopInteger

Заменя типа на Double/Integer с  
параметър за тип T

Pushing elements onto doubleStack

1.1 2.2 3.3 4.4 5.5 6.6

```
FullStackException: Stack is full, cannot push 6.6
    at Stack.push(Stack.java:30)
    at StackTest2.testPush(StackTest2.java:38)
    at StackTest2.testStacks(StackTest2.java:19)
    at StackTest2.main(StackTest2.java:74)
```

Popping elements from doubleStack

5.5 4.4 3.3 2.2 1.1

```
EmptyStackException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:40)
    at StackTest2.testPop(StackTest2.java:60)
    at StackTest2.testStacks(StackTest2.java:20)
    at StackTest2.main(StackTest2.java:74)
```

Pushing elements onto integerStack

1 2 3 4 5 6 7 8 9 10 11

```
FullStackException: Stack is full, cannot push 11
    at Stack.push(Stack.java:30)
    at StackTest2.testPush(StackTest2.java:38)
    at StackTest2.testStacks(StackTest2.java:21)
    at StackTest2.main(StackTest2.java:74)
```

Popping elements from integerStack

10 9 8 7 6 5 4 3 2 1

```
EmptyStackException: Stack is empty, cannot pop
    at Stack.pop(Stack.java:40)
    at StackTest2.testPop(StackTest2.java:60)
    at StackTest2.testStacks(StackTest2.java:22)
    at StackTest2.main(StackTest2.java:74)
```

## 8а.6 параметризирани класове-особености на конструкторите

Параметрите за тип могат да се декларират също **вътре в метод** или **вътре “подпис”**-ите на **конструктори**, при което се създават параметризирани методи и параметризирани конструктори

Това е **аналогично на създаването на параметризирани класове**, но **обхвата на валидност** на такъв параметър за тип **е ограничен** до метода или конструктора, в който този параметър за тип е деклариран

```

/**
 * 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");
    }
}

```

параметризиран клас с параметър за тип T

параметризиран 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



## 8а.6 параметризирани класове-особености на конструкторите

Един по-реалистичен начин за използване на параметризирани методи с `class Box` е следният пример:

```
public static <U> void fillBoxes(U u, List<Box<U>> boxes) {  
    for (Box<U> box : boxes) {  
        box.add(u);  
    }  
}
```

Тук се дефинира статичен метод за *запълване на елементите в списък* от `Box` с единствен обект от тип, зададен с параметър `U`

## 8a.6 параметризирани класове-особености на конструкторите

За използване на метод `fillBoxes()` са необходими обекти за неговите аргументи, примерно като:

```
Crayon red = ...;
```

```
List<Box<Crayon>> crayonBoxes = ...;
```

**Пълният синтаксис** за извикването на `fillBoxes()` е:

```
Box<Crayon>.fillBoxes(red, crayonBoxes);
```

Тук **явно се изписва конкретния тип** `<Crayon>` за параметъра `U`

## 8a.6 параметризирани класове-особености на конструкторите

Пълният синтаксис изисква явно задаване на типа, присвояван на параметъра за тип.

По- често, това се изпуска и се оставя на компилаторът да се “досети” за необходимият тип :

```
Box.fillBoxes(red, crayonBoxes) ;
```

```
// компилаторът се досеща, че U е клас Crayon
```

Тази особеност се нарича “*откриване на тип*” (“type inference”) и позволява да се извика параметризиран параметризиран метод, както всеки друг обикновен (*не- параметризиран*) метод без да се задава явно ТИПЪТ в ъглови скоби

## 8a.6 параметризирани класове-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 & MyInterface>
```

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

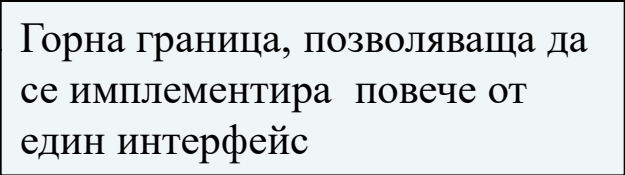
    private T t;

    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

# 8a.7 Необработен (raw) типове данни

## Необработен тип – *дефиниция*

- Позволява да се създаде параметризиран клас без задаване на реална стойност за параметър за тип т.е. може да пишем

```
Stack objectStack = new Stack( 5 );
```

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

# 8a.7 Необработен (raw) типове данни

## Необработен тип - *приложение*

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

```
Stack rawTypeStack2 = new Stack< Double >( 5 );
```

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

```
Stack< Integer > integerStack = new Stack( 10 );
```

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

# 8a.7 Необработен (raw) типове данни

## Необработен тип - *приложение*

```
Stack< Integer > integerStack = new Stack( 10 );
```

не се третира като “*сигурно*”, понеже по същия начин може да се напише и

```
Stack< String> stringStack = new Stack( 10 );
```

```
Stack< Box > boxStack = new Stack( 10 );
```

В **JDK 7** се използва “diamond” оператора <>

```
Stack< Box > boxStack = new Stack<>( 10 );
```

което е еквивалентно на

```
Stack< Box > boxStack = new Stack< Box >( 10 );
```

т.е компилатора се “*досеца*”, че Stack<>( 10 ) е

Stack< Box > , а не **суровия** тип Stack.



# 8a.7 Необработен (raw) типове данни

## Необработен тип - *приложение*

```
Stack< Integer > integerStack = new Stack( 10 );
```

не се третира като “*сигурно*”, понеже по същия начин може да се напише и

```
Stack< String> stringStack = new Stack( 10 );
```

```
Stack< Box > boxStack = new Stack( 10 );
```

В **JDK 7** се използва “diamond” оператора <>

```
Stack< Box > boxStack = new Stack<>( 10 );
```

което е еквивалентно на

```
Stack< Box > boxStack = new Stack< Box >( 10 );
```

т.е компилатора се “*досеца*”, че Stack<>( 10 ) е

Stack< Box > , а не **суровия** тип Stack.

Не се разрешава използването му с анонимни класове

```

1 // Fig. 18.12: RawTypeTest.java
2 // Raw type test program.
3
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 testStacks()
12    {
13        // Stack of raw types assigned to Stack of raw types variable
14        Stack rawTypeStack1 = new Stack( 5 );
15
16        // Stack< Double > assigned to stack of raw types variable
17        Stack rawTypeStack2 = new Stack< Double >( 5 );
18
19        // Stack of raw types assigned to Stack< Integer > variable
20        Stack< Integer > integerStack = new Stack( 10 );
21
22        testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
23        testPop( "rawTypeStack1", rawTypeStack1 );
24        testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
25        testPop( "rawTypeStack2", rawTypeStack2 );
26        testPush( "integerStack", integerStack, integerElements );
27        testPop( "integerStack", integerStack );
28    } // end method testStacks
29

```

Създава обект **Stack**  
от необработен тип

Присвоява **Stack< Double >**  
на променливата  
**rawTypeStack2**

Присвоява **Stack** от  
необработен тип на  
**Stack< Integer >**  
променлива.

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

```
30 // generic method pushes elements onto stack
31 public < T > void testPush( String name, Stack< T > stack,
32     T[] elements )
33 {
34     // push elements onto stack
35     try
36     {
37         System.out.printf( "\nPushing elements onto %s\n", name );
38
39         // push elements onto stack
40         for ( T element : elements )
41         {
42             System.out.printf( "%s ", element );
43             stack.push( element ); // push element onto stack
44         } // end for
45     } // end try
46     catch ( FullStackException fullStackException )
47     {
48         System.out.println();
49         fullStackException.printStackTrace();
50     } // end catch FullStackException
51 } // end method testPush
52
```

```
53 // generic method testPop pops elements from stack
54 public < T > void testPop( String name, Stack< T > stack )
55 {
56     // pop elements from stack
57     try
58     {
59         System.out.printf( "\nPopping elements from %s\n", name );
60         T popValue; // store element removed from stack
61
62         // remove elements from Stack
63         while ( true )
64         {
65             popValue = stack.pop(); // pop from stack
66             System.out.printf( "%s ", popValue );
67         } // end while
68     } // end try
69     catch( EmptyStackException emptyStackException )
70     {
71         System.out.println();
72         emptyStackException.printStackTrace();
73     } // end catch EmptyStackException
74 } // end method testPop
75
76 public static void main( String args[] )
77 {
78     RawTypeTest application = new RawTypeTest();
79     application.testStacks();
80 } // end main
81 } // end class RawTypeTest
```

Pushing elements onto rawTypeStack1

1.1 2.2 3.3 4.4 5.5 6.6

FullStackException: Stack is full, cannot push 6.6

at Stack.push(Stack.java:30)

at RawTypeTest.testPush(RawTypeTest.java:43)

at RawTypeTest.testStacks(RawTypeTest.java:22)

at RawTypeTest.main(RawTypeTest.java:79)

Popping elements from rawTypeStack1

5.5 4.4 3.3 2.2 1.1

EmptyStackException: Stack is empty, cannot pop

at Stack.pop(Stack.java:40)

at RawTypeTest.testPop(RawTypeTest.java:65)

at RawTypeTest.testStacks(RawTypeTest.java:23)

at RawTypeTest.main(RawTypeTest.java:79)

Pushing elements onto rawTypeStack2

1.1 2.2 3.3 4.4 5.5 6.6

FullStackException: Stack is full, cannot push 6.6

at Stack.push(Stack.java:30)

at RawTypeTest.testPush(RawTypeTest.java:43)

at RawTypeTest.testStacks(RawTypeTest.java:24)

at RawTypeTest.main(RawTypeTest.java:79)

Popping elements from rawTypeStack2

5.5 4.4 3.3 2.2 1.1

EmptyStackException: Stack is empty, cannot pop

at Stack.pop(Stack.java:40)

at RawTypeTest.testPop(RawTypeTest.java:65)

at RawTypeTest.testStacks(RawTypeTest.java:25)

at RawTypeTest.main(RawTypeTest.java:79)

Pushing elements onto integerStack

1 2 3 4 5 6 7 8 9 10 11

FullStackException: Stack is full, cannot push 11

at Stack.push(Stack.java:30)

at RawTypeTest.testPush(RawTypeTest.java:43)

at RawTypeTest.testStacks(RawTypeTest.java:26)

at RawTypeTest.main(RawTypeTest.java:79)

Popping elements from integerStack

10 9 8 7 6 5 4 3 2 1

EmptyStackException: Stack is empty, cannot pop

at Stack.pop(Stack.java:40)

at RawTypeTest.testPop(RawTypeTest.java:65)

at RawTypeTest.testStacks(RawTypeTest.java:27)

at RawTypeTest.main(RawTypeTest.java:79)

```

RawTypeTest.java:20: warning: unchecked assignment
found    : Stack
required: Stack<java.lang.Integer>
    Stack< Integer > integerStack = new Stack( 10 );
                                   ^
RawTypeTest.java:22: warning: [unchecked] unchecked method invocation:
<T>testPush(java.lang.String,Stack<T>,T[]) in RawTypeTest is applied to
(java.lang.String,Stack,java.lang.Double[])
    testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
    ^
RawTypeTest.java:23: warning: [unchecked] unchecked method invocation:
<T>testPop(java.lang.String,Stack<T>) in RawTypeTest is applied to
(java.lang.String,Stack)
    testPop( "rawTypeStack1", rawTypeStack1 );
    ^
RawTypeTest.java:24: warning: [unchecked] unchecked method invocation:
<T>testPush(java.lang.String,Stack<T>,T[]) in RawTypeTest is applied to
(java.lang.String,Stack,java.lang.Double[])
    testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
    ^
RawTypeTest.java:25: warning: [unchecked] unchecked method invocation:
<T>testPop(java.lang.String,Stack<T>) in RawTypeTest is applied to
(java.lang.String,Stack)
    testPop( "rawTypeStack2", rawTypeStack2 );
    ^
5 warnings

```

**Fig. 8a.13 | Предупредителни съобщения от компилатора.**

## 8a.8 Шаблони за параметри на тип като горна и долна граница на параметър

### 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()`



## 8a.8 Шаблони за параметри на тип като горна и долна граница на параметър

### Необходимост от използване на шаблони

#### – Пример

- реализация на параметризиран 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. 18.14: 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

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

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

## 8a.8 Шаблиони за параметри на тип като горна и долна граница на параметър

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

- `ArrayList< ? extends Number >`
- Шаблонът `?` означава “**неизвестен тип**”, множеството от всички типове
- `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

## Обичайна грешка при програмиране 8а.4

**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.**

## 8a.8 Шаблони за параметри на тип като горна и долна граница на параметър

Вариант, при който метод `sum` се реализира с шаблон за параметър на типа на аргумента

- `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`

## Метод *sum* с използване на шаблон

```
1 // Fig. 18.15: 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

# 8a.9 Параметризиране и наследственост: Особености

## Наследственост при използване на пораждане

- параметризиран клас може да е произведен на не-параметризиран клас  
т.е., `class Object` е базов клас за всеки параметризиран клас
- параметризиран клас може да е произведен на друг параметризиран клас  
т.е., `class Stack` е произведен на `class Vector`
- Не-параметризиран клас може да е произведен на параметризиран клас  
т.е., `Properties` е произведен на `class Hashtable`
- параметризиран метод може в произведен клас може да предефинира метод в базов клас, ако двата метода имат същия *“подпис”*

## 8a.9 Параметризиране и наследственост : Особености

### 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

## 8a.9 Параметризиране и наследственост : Особенности

### Пример:

Понеже `Integer` е произведен 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
```

## 8a.9 Параметризиране и наследственост : Особенности

### Пример:

Аналогични конструкции са възможни с параметризирани 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`

## 8a.9 Параметризиране и наследственост : Особености

### 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,” понеже **Box<Integer>** и **Box<Double>** are не са производни на **Box<Number>**.

## 8a.9 Параметризиране и наследственост : Особености

### 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>`

## 8а.9 Параметризиране и наследственост : Особености

### 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>



## 8a.9 Параметризиране и наследственост : Особености

`super` не може да се използва за деклариране на параметър на тип

`// hypothetical! currently illegal in Java`

```
<T super Integer> T add(T number) { return number++; }
```

Ако беше допустимо, то този метод щеше да се изпълнява със следните декларации

```
Integer anInteger
```

```
Number aNumber
```

```
Object anObject
```

```
String aString
```

Очакването `<T super Integer>` (ако беше позволено) е да позволи изпълнение `add(anInteger)`, `add(aNumber)`, и `add(anObject)`, но не и `add(aString)`.

Същевременно, `String` е `Object`, така че `add(aString)` би се компилирало, но ще доведе до грешка при изпълнение

## 8a.10 Принципът Get- Put

### Принцип:

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

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

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

### Пример:

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

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**.

## 8a.10 Принципът Get- Put

### Пример:

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

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

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

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

## 8a.10 Принципът Get- Put

Пример:

```
public static void count(Stack<? 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**

```
Stack<Integer> ints = new Stack<>();
```

```
count(ints, 6); ints.add(6); // legal PUT
```

```
Stack<Number> doubles = new Stack<>();
```

```
count(doubles, 6); doubles.add(6.5) // legal PUT
```

```
Stack<Object> objs= new Stack<>();
```

```
count(objs, 6); objs.add("six"); // legal PUT
```

## 8a.10 Принципът Get- Put

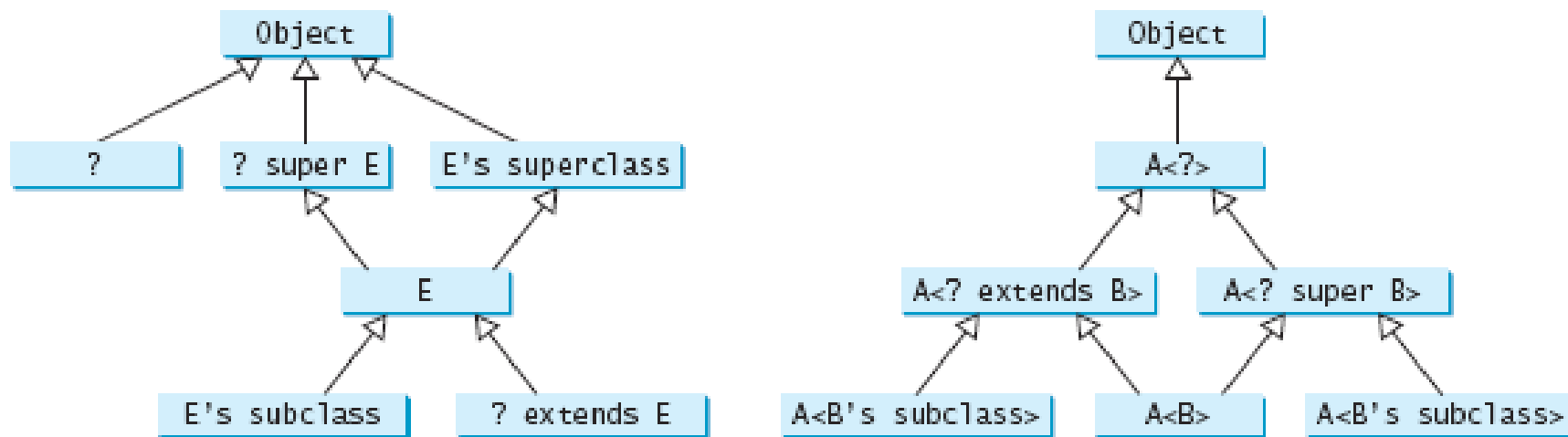
### Пример:

```
public static double sumCount(Stack<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

# 8a.10 Принципът Get- Put

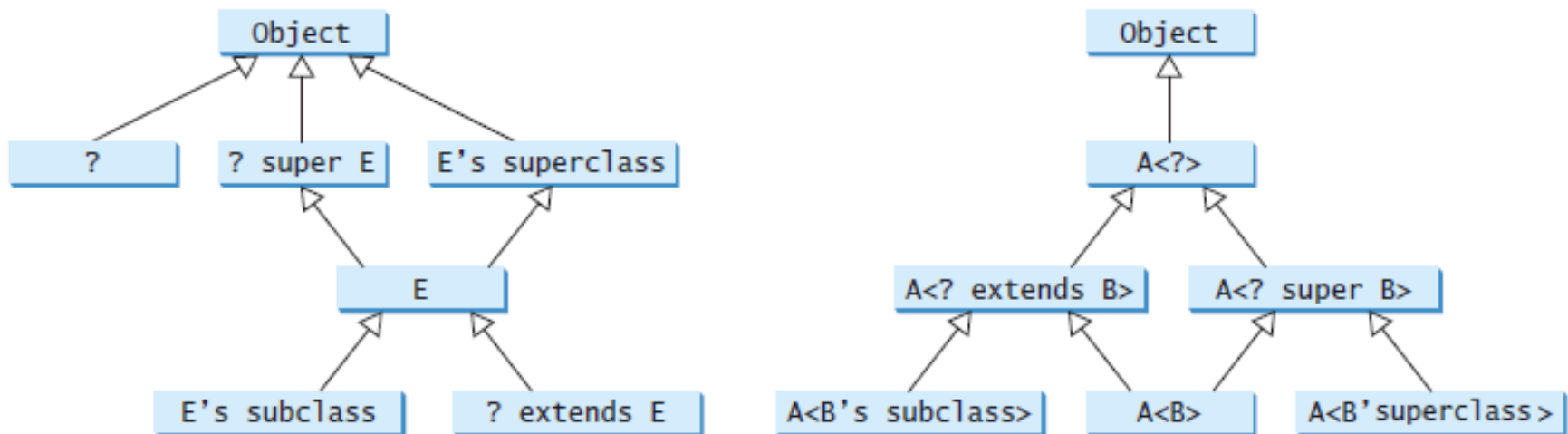
Обобщение:



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

# 8a.11 Параметризиране и наследственост : Особенности

The inheritance relationship involving generic types and wildcard types is summarized in the following figure. Here **A** and **B** represent classes or interfaces, and **E** is a generic type parameter.



## 8a.11 Параметризиране и наследственост : Особенности

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 }

```



## 8a.12 Анонимен клас с <> оператор

В JDK 9 се разширява използването на <> и с анонимни класове.

```
abstract class Handler<T> {  
    public T content;  
  
    public Handler(T content) {  
        this.content = content;  
    }  
    abstract void handle();  
}
```

## 8a.12 Анонимен клас с <> оператор

В JDK 9 се разширява използването на <> и с анонимни класове.

```
abstract class Handler<T> {  
    public T content;  
  
    public Handler(T content) {  
        this.content = content;  
    }  
    abstract void handle();  
}
```

// Before JDK 9

```
public class Tester {  
    public static void main(String[] args) {  
        Handler<Integer> intHandler = new Handler<Integer>(1) {  
            @Override  
            public void handle() {  
                System.out.println(content);  
            }  
        };  
        intHandler.handle(); // outputs 1  
        Handler<? extends Number> intHandler1 = new Handler<Number>(2) {  
            @Override  
            public void handle() {  
                System.out.println(content);  
            }  
        };  
        intHandler1.handle(); // outputs 2  
        Handler<?> handler = new Handler<Object>("test") {  
            @Override  
            public void handle() {  
                System.out.println(content);  
            }  
        };  
        handler.handle(); // outputs test  
    }  
}
```

// After JDK 9

```
public class Tester {  
    public static void main(String[] args) {  
        Handler<Integer> intHandler = new Handler<>(1) {  
            @Override  
            public void handle() {  
                System.out.println(content);  
            }  
        };  
        intHandler.handle(); // outputs 1  
        Handler<? extends Number> intHandler1 = new Handler<>(2) {  
            @Override  
            public void handle() {  
                System.out.println(content);  
            }  
        };  
        intHandler1.handle(); // outputs 2  
        Handler<?> handler = new Handler<>("test") {  
            @Override  
            public void handle() {  
                System.out.println(content);  
            }  
        };  
        handler.handle(); // outputs test  
    }  
}
```

# 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 `ListCompositon`, даден на лекции

# Задачи

## Задача 3.

Презаредете параметризирания метод `printArray` от Fig. 8a.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. 8a.3 с версия на не – параметризиран метод , която позволява да се изпълни при задаване на масив от текстови низове и в този случай да отпечатва елементите на този масив подредени в колони както е показано по-долу:

| <code>Array</code> | <code>stringArray</code> | <code>съдържа:</code> |                    |
|--------------------|--------------------------|-----------------------|--------------------|
| <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 метод `isEqualTo()` която сравнява двата аргумента с метода `equals()` и връща **true** ако са равни и **false** в противен случай. Използвайте този параметризиран 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 параметризиран `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 >
```