ГЕНЕТИЧНО ПРОГРАМИРАНЕ

ЛЕКЦИОНЕН КУРС "ПРОГРАМИРАНЕ НА JAVA"

Distributed @learning center

СТРУКТУРА НА ЛЕКЦИЯТА

- Въведение
- Генетични класове
- Параметри с ограничени типове
- Генетични методи
- Маски
- Примери

УВОД

- От първата версия на Java много нови функции са добавени
  - Всички те са подобрили и разширили обхвата на езика
- Едно от разширенията с особено дълбоко и широкообхватно въздействие са възможностите за генетичното програмиране (generics)
  - Поради ефекти върху целия език
  - Напр., добавя напълно нов елемент на синтаксиса и причини промени в много от класовете и методите в основното API
- Не е преувеличено да се каже, че включването на генетичното програмиране основно преформулира характера на Java

УВОД

- Темата е доста голяма
- Базово разбиране на генетичния подход е необходимо за Java програмистите
- На пръв поглед генетичният синтаксис може да изглежда малко смущаващ
- По принцип generics са лесни за използване

ОСНОВИ

- В основата си терминът означава параметризирани типове
- Параметризираните типове са важни
  - Те позволяват създаване на класове, интерфейси и методи, в които типът данни, с които те работят, е определен като параметър
- Клас, интерфейс или метод, който работи с параметър за тип, се нарича генетичен
  - Генетичен клас
  - Генетичен метод

СЦЕНАРИЙ

- Да разгледаме следния сценарий:
  - Искаме да се разработи контейнер, който ще бъде използван за приемане на обектите в приложения
  - Типът на обектите не винаги ще бъде същият за различните приложения
  - Затова е необходимо да се разработи един контейнер, който има способността да съхранява обекти от различен тип
- За сценария, най-очевидният начин за постигане на целта ще бъде да се разработи контейнер, който има способността да съхраняват и извличат обекти от тип Object, а след това да ги преобразува в различни типове

ПРИМЕР

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
public final class OldStyleList   {
   public static void main(String[] args)   {
      final List personList = new ArrayList();
       personList.add(new Person("Иван", new Date(), "Пловдив"));
       personList.add(new Person("Мария", new Date(), "Пловдив"));
       personList.add(new Dog("Sarah from Plovdiv"));
       for (int i = 0; i < personList.size(); i++)   {
          final Person person = (Person) personList.get(i);
          System.out.println(person.getName() + " от " + person.getCity());
       }
   }
   private OldStyleList()   {   }
}
class Dog   {
   private final String name;
   public Dog(final String name)   {  this.name = name;   }

   @Override
   public String toString()   {  return "Dog [name='" + name + "']";   }
}
```

? Резултат

```
Иван от Пловдив
Мария от Пловдив
Exception in thread "main" java.lang.ClassCastException: Dog cannot be cast to
Person at OldStyleList.main(OldStyleList.java:27)
```

ПРИМЕР

```
public class ObjectContainer {
   private Object obj;

   // @return the obj

   public Object getObj() {
      return obj;
   }

   // @param obj the obj to set

   public void setObj(Object obj) {
      this.obj = obj;
   }

}
```

- Въпреки, че контейнерът ще постигне желания резултат, няма да е най-подходящото решение за нашата цел
- Понеже има потенциала да предизвика изключения надолу по пътя
  - ✓ Не е сигурен към типовете
  - ✓ Изисква използвате явно преобразуване всеки път, когато се търси обекта

ИЗПОЛЗВАНЕ

```
ObjectContainer myObj = new ObjectContainer();

myObj.setObj("Test");
System.out.println("Value of myObj:" + myObj.getObj());

myObj.setObj(3);
System.out.println("Value of myObj:" + myObj.getObj());

List objectList = new ArrayList();
objectList.add(myObj);

String myStr = (String) ((ObjectContainer)objectList.get(0)).getObj();
System.out.println("myStr: " + myStr);
```

Съхраняване на низ

Съхраняване на цяло число

ИЗПОЛЗВАНЕ

? Резултат

```
ObjectContainer myObj = new ObjectContainer();

myObj.setObj("Test");
System.out.println("Value of myObj:" + myObj.getObj());

myObj.setObj(3);
System.out.println("Value of myObj:" + myObj.getObj());

List objectList = new ArrayList();
objectList.add(myObj);

String myStr = (String) ((ObjectContainer)objectList.get(0)).getObj();
System.out.println("myStr: " + myStr);
```

```
Value of myObj:Test
Exception in thread "main" Value of myObj:3
java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
at ObjectContainer.main(ObjectContainer.java:37)
```

ИЗПОЛЗВАНЕ

```
ObjectContainer myObj = new ObjectContainer();

myObj.setObj("Test");
System.out.println("Value of myObj:" + myObj.getObj());

myObj.setObj(3);
System.out.println("Value of myObj:" + myObj.getObj());

List objectList = new ArrayList();
objectList.add(myObj);

String myStr = (String) ((ObjectContainer)objectList.get(0)).getObj();
System.out.println("myStr: " + myStr);
```

Трябва да направим преобразуване към коректния тип за да избегнем ClassCast Exception

ИЗПОЛЗВАНЕ

```
ObjectContainer myObj = new ObjectContainer();

myObj.setObj("Test");
System.out.println("Value of myObj:" + myObj.getObj());

myObj.setObj(3);
System.out.println("Value of myObj:" + myObj.getObj());

List objectList = new ArrayList();
objectList.add(myObj);

Integer myStr = (Integer) ((ObjectContainer)objectList.get(0)).getObj();
System.out.println("myStr: " + myStr);
```

ИЗПОЛЗВАНЕ

? Резултат

```
ObjectContainer myObj = new ObjectContainer();

myObj.setObj("Test");
System.out.println("Value of myObj:" + myObj.getObj());

myObj.setObj(3);
System.out.println("Value of myObj:" + myObj.getObj());

List objectList = new ArrayList();
objectList.add(myObj);

Integer myStr = (Integer) ((ObjectContainer)objectList.get(0)).getObj();
System.out.println("myStr: " + myStr);
```

```
Value of myObj:Test
Value of myObj:3
myStr: 3
```

ПО-ДОБРО РЕШЕНИЕ

- Generics могат да се използват за разработване на по-добро решение
  - Като се използва контейнер, който може да има определен тип (генетичен тип), определен при инициализацията
  - Позволява създаването на инстанции на контейнера, които могат да се използват за съхранение на обекти от определени типове
- Един генетичен тип е клас или интерфейс, който може да се параметризира спрямо типа
  - Което означава, че актуалният тип може да бъде определен чрез заместване на генетичния тип с конкретен тип
- Актуалният тип ще ограничава стойностите, които ще се използват в рамките на контейнера
  - Премахва се изискването за преобразуване
  - Осигурява се по-силна проверка на типовете по време на компилация

ПРИМЕР

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public final class NewStyleList  {
   public static void main(String[] args)   {
      final List<Person> personList = new ArrayList<Person>();
      personList.add(new Person("Иван", new Date(), "Пловдив"));
      personList.add(new Person("Мария", new Date(), "Пловдив"));
      // personList.add(new Dog("Sarah from Plovdiv"));  // Compile-Error

      for (int i = 0; i < personList.size(); i++)   {
         final Person person = personList.get(i);
         System.out.println(person.getName() + " от " + person.getCity());
      }
   }

   private NewStyleList()  {    }
}
```

? Резултат

Иван от Пловдив
Мария от Пловдив

ПРИМЕР

```
public class GenericContainer<T>  {
   private T obj;

   public GenericContainer()  {    }

   public GenericContainer(T t)  {
      obj = t;
   }
   // @return the obj

   public T getObj()  {
      return obj;
   }

   // @param obj the obj to set

   public void setObj(T t)  {
      obj = t;
   }
}
```

- Най-съществените различия са, че дефиницията на класа съдържа <T> и полето obj не е вече от тип Object, а по-скоро от генетичния тип T
- Скобите в дефиницията на класа заграждат секцията на тип-параметрите, които ще бъдат използвани в рамките на класа
- T е параметър, свързан с генетичния общ тип, който е дефиниран в този клас

ИЗПОЛЗВАНЕ

```
public class GenericContainer<T>  {
   private T obj;

   public GenericContainer()  {    }

   public GenericContainer(T t)  {
      obj = t;
   }
   // @return the obj

   public T getObj()  {
      return obj;
   }

   // @param obj the obj to set

   public void setObj(T t)  {
      obj = t;
   }
}
```

- За да използваме генетичния контейнер, трябва да присвоим тип на контейнера чрез спецификация, дадена в <>
- В примера, инициализираме контейнера за цели числа
- При записване на низ компилаторът ще възрази

```
GenericContainer<Integer> myInt =  new
GenericContainer<Integer>();

myInt.setObj(3);  // OK
myInt.setObj("Int"); // няма да се компилира
```

ПОЛЗИ ОТ GENERICS

- Някои от основните ползи от използване на генетични типове
  - По-строги проверки на типовете е един от най-важните, защото това спестява време, като се предотвратява възникването на ClassCastExceptions, които могат да бъдат изхвърлени по време на изпълнение
  - Премахване на преобразуването, което означава използване по-малко код, тъй като компилаторът знае точно какъв тип ще се съхраняват в една колекция
  - Могат да се разработят генерични алгоритми, които могат да бъдат персонализирани за решаване на конкретна задача

ПРИМЕР

```
List myObjList = new ArrayList();

for(int x=0; x <=10; x++) {
    ObjectContainer myObj = new ObjectContainer();
    myObj.setObj("Test" + x);
    myObjList.add(myObj);
}

for(int x=0; x <= myObjList.size()-1; x++) {
   ObjectContainer obj = (ObjectContainer) myObjList.get(x);
   System.out.println("Object Value: " + obj.getObj());
}

List<GenericContainer> genericList = new ArrayList<GenericContainer>();

for(int x=0; x <=10; x++) {
    GenericContainer<String> myGeneric = new GenericContainer<String>();
    myGeneric.setObj(" Generic Test" + x);
    genericList.add(myGeneric);
}

for(GenericContainer<String> obj:genericList) {
    String objectString = obj.getObj();
    System.out.println(objectString);
}
```

Съхраняване инстанции на ObjectContainer

За получаване на обекти е необходимо преобразуване

Съхраняване инстанции на GenericContainer

За получаване на обекти не е необходимо преобразуване

ПРИМЕР

Разлики между съхраняване на обекти в Object контейнер и съхраняване в GenericContainer

```
List myObjList =

for(int x=0; x <=
   ObjectContainer myObj = new ObjectContainer();
   myObj.setObj("Test" + x);
   myObjList.add(myObj);
}

for(int x=0; x <=
  ObjectContaine
  System.out.prin
}

List<GenericCont

for(int x=0; x <=1
   GenericContainer<String> myGeneric = new GenericContainer<String>();
   myGeneric.setObj(" Generic Test" + x);
   genericList.add(myGeneric);
}

for(GenericContainer<String> obj:genericList) {
   String objectString = obj.getObj();
   System.out.println(objectString);
}
```

- При използване на ArrayList, ние сме в състояние да определим вида на колекцията при създаването ѝ, като се използва означението в скобите (<GenericContainer>)
- Колекцията ще бъде в състояние да съхранява само GenericContainer обекти (или на подкласове на GenericContainer), като не е необходимо преобразуване при извличане на обекти от колекцията

ПРИМЕР

? Резултат

```
List myObjList = new ArrayList();

for(int x=0; x <=10; x++) {
    ObjectContainer myObj = new ObjectContainer();
    myObj.setObj("Test" + x);
    myObjList.add(myObj);
}

for(int x=0; x <= myObjList.size()-1; x++) {
   ObjectContainer obj = (ObjectContainer) myObjList.get(x);
   System.out.println("Object Value: " + obj.getObj());
}

List<GenericContainer> genericList = new ArrayList<GenericContainer>();

for(int x=0; x <=10; x++) {
    GenericContainer<String> myGeneric = new GenericContainer<String>();
    myGeneric.setObj(" Generic Test" + x);
    genericList.add(myGeneric);
}

for(GenericContainer<String> obj:genericList) {
    String objectString = obj.getObj();
    System.out.println(objectString);
}
```

```
Object Value: Test0
Object Value: Test1
Object Value: Test2
Object Value: Test3
Object Value: Test4
Object Value: Test5
Object Value: Test6
Object Value: Test7
Object Value: Test8
Object Value: Test9
Object Value: Test10
 Generic Test0
 Generic Test1
 Generic Test2
 Generic Test3
 Generic Test4
 Generic Test5
 Generic Test6
 Generic Test7
 Generic Test8
 Generic Test9
 Generic Test10
```

ИЗПОЛЗВАНЕ НА GENERICS

- Съществуват различни възможности за използване на generics
- В примера беше разгледан случай за генериране на генетични типове
  - Това е една добра отправна точка за изучаване синтаксиса на generics на ниво класове и интерфейси
  - Сигнатурата клас съдържа раздел за тип-параметри, зададен в ъглови скоби (<>) след името на класа
  - Напр., public class GenericContainer<T> { ...

ТИПОВЕ ПРОМЕНЛИВИ

- Тип-параметрите (наричат се също типови променливи) се използват като пазители на места, които индикират, че даден тип ще бъде назначен в runtime
- Може да има един или повече тип-параметри, които могат да бъдат използвани в рамките на един клас при необходимост
- Според използваната конвенция, тип-параметрите се означават с една единствена главна буква, която показва типа на дефинирания параметър
- Стандартни означения:
  - E: елемент
  - K: ключ
  - N: число
  - T: тип
  - V: стойност
  - S, U, V, … : втори, трети, четвърти параметър в един списък

ПОВЕЧЕ ГЕНЕТИЧНИ ТИПОВЕ

- В определени случаи е полезно да има възможност да се използват повече от един генетичен тип в един клас или интерфейс
- Повечето тип-параметрите могат да се използват в клас или интерфейс чрез поставяне разделен със запетаи списък на типове между ъглови скоби

ПРИМЕР

```
public class MultiGenericContainer<T, S> {
    private T firstPosition;
    private S secondPosition;

    public MultiGenericContainer(T firstPosition, S secondPosition){
        this.firstPosition = firstPosition;
        this.secondPosition = secondPosition;
    }
    public T getFirstPosition(){
        return firstPosition;
    }
    public void setFirstPosition(T firstPosition){
        this.firstPosition = firstPosition;
    }
    public S getSecondPosition(){
        return secondPosition;
    }
    public void setSecondPosition(S secondPosition)   {
        this.secondPosition = secondPosition;
    }
}
```

- MultiGenericContainer съхранява два типа обекти, всеки от тях се определя при инициализацията
- Означенията – в съответствие с конвенцията

ИЗПОЛЗВАНЕ

```
public class MultiGenericContainer<T, S> {
    private T firstPosition;
    private S secondPosition;

    public MultiGenericContainer(T first
        this.firstPosition = firstPosition;
        this.secondPosition = secondPositi
    }
    public T getFirstPosition(){
        return firstPosition;
    }
    public void setFirstPosition(T firstPo
        this.firstPosition = firstPosition;
    }
    public S getSecondPosition(){
        return secondPosition;
    }
    public void setSecondPosition(S secondPosition)   {
        this.secondPosition = secondPosition;
    }
}
```

```
MultiGenericContainer<String, String> mondayWeather =
        new MultiGenericContainer<String, String>("Monday", "Sunny");
MultiGenericContainer<Integer, Double> dayOfWeekDegrees =
        new MultiGenericContainer<Integer, Double>(1, 78.0);

String mondayForecast = mondayWeather.getFirstPosition();

double sundayDegrees = dayOfWeekDegrees.getSecondPosition();
```

UNBOXING & AUTOBOXING

- При деклариране на инстанция от генетичен тип аргументът трябва да бъде референтен тип
- Не можем да използваме примитивен тип, като int или char
- Невъзможността да се определи примитивен тип не е сериозно ограничение
  - Можем да използваме обвивките на примитивните типове
- Механизмът unboxing & autoboxing в Java правят използването на обвивките прозрачно

ПРИМЕР

? Коментар

```
MultiGenericContainer<String, String> mondayWeather =
      new MultiGenericContainer<String, String>("Monday", "Sunny");
MultiGenericContainer<Integer, Double> dayOfWeekDegrees =
      new MultiGenericContainer<Integer, Double>(1, 78.0);

String mondayForecast = mondayWeather.getFirstPosition();

double sundayDegrees = dayOfWeekDegrees.getSecondPosition();
```

autoboxing

unboxing

ОПЕРАТОР „ДИАМАНТ“

? Коментар

- Тип-интерфейс: способност на Java компилатора да определи автоматично тип-параметрите, позовавайки се на декларацията и извикването на метода
- Вместо да декларираме типовете повторно можем да използваме оператора “диамант”

```
MultiGenericContainer<String, String> mondayWeather =
      new MultiGenericContainer<>("Monday", "Sunny");
MultiGenericContainer<Integer, Double> dayOfWeekDegrees =
      new MultiGenericContainer<>(1, 78.0);
```

ОГРАНИЧЕНИ ТИПОВЕ

- В предходните примери параметрите на типа могат да бъдат заменени от всеки тип клас
- Това е добре за много цели, но понякога е полезно да се ограничат типовете, които могат да бъде предаван на тип параметър
- Да приемем напр., че искаме да създадем генетичен клас, който съхранява цифрова стойност и е в състояние да изпълнява различни математически функции, като изчисляване на реципрочните или получаване на частно
- Освен това, искаме да използвате класа, за да изчислите тези количества за всеки тип число, включително int, float, double
- По този начин искате да посочим типа от числови стойности генетично, използвайки тип на параметър

ПРИМЕР

```
public class NumericFns<T> {
     T num;

     // Pass the constructor a reference to a numeric object
     NumericFns(T n) {
          num = n;
     }

    // Returns the reciprocal
    double reciprocal() {
        return 1 / num.doubleValue();    // Error
     }

     // Returns the fractional componet
     double fraction() {
          return num.doubleValue() - num.intValue();   // Error
      }
}
```

ПРИМЕР

```
public class NumericFns<T> {
    T num;

    // Pass the constructor a reference to a numeric object
    NumericFns(T n) {
        num = n;
    }

    // Returns the reciprocal
    double reciprocal() {
        return 1 / num.doubleValue();    // Error
    }

    // Returns the fractional componet
    double fraction() {
        return num.doubleValue() - num.intValue();   // Err
    }
}
```

- NumericFns няма да се компилира - двата метода ще генерират грешки при компилиране
- reciprocal() метод връща реципрочна стойност - за това трябва да раздели 1 на стойността на num
- Стойността на num се получава чрез извикване на doubleValue() за получаване на double версията на числения обект, съхраняван в num

ПРИЧИНА ЗА ГРЕШКАТА

- Всички числови класове, като напр., Integer и Double, са подкласове на Number
- Number дефинира метод doubleValue(), който е достъпен за всички класове с цифрови обвивки
- Проблемът е, че компилаторът няма начин да знае, че възнамеряваме да създаваме NumericFns обекти, използващи само цифрови типове

ПРИМЕР

```
public class NumericFns<T> {
    T num;

    // Pass the constructor a reference to a numeric object
    NumericFns(T n) {
        num = n;
    }

    // Returns the reciprocal
    double reciprocal() {
        return 1 / num.doubleValue();    // Error
    }

    // Returns the fractional componet
    double fraction() {
        return num.doubleValue() - num.intValue();   // Error
    }
}
```

Същият тип грешка се появява отново при fractional() – при извикване на doubleValue() и intValue()

ОБОБЩЕНИЕ

- И двете повиквания водят до съобщения за грешка, които сочат, че тези методи са неизвестни
- За да разрешим този проблем е необходим някакъв начин да кажем на компилатора, че възнамеряваме да предавате само цифрови типове на T
- Освен това, има нужда по някакъв начин да се гарантира, че всъщност се предават само цифрови типове

ПРИМЕР

```
public class NumericFns<T extends Number> {
    T num;

    // Pass the constructor a reference to a numeric object
    NumericFns(T n) {
        num = n;
    }

    // Returns the reciprocal
    double reciprocal() {
        return 1 / num.doubleValue();
    }

    // Returns the fractional componet
    double fraction() {
        return num.doubleValue() - num.intValue();    }
}
```

Същият тип грешка се появява отново при fractional() – при извикване на doubleValue() и intValue()

ОГРАНИЧЕНИ ТИПОВЕ

- В случая искаме да контролираме типове, които могат да бъдат специфицирани, вместо достъпът да бъде широко отворен
- Ограничени типове
  - Използват се за ограничаване на границите на генетичния тип
  - Използват се ключовите думи extends и super
  - Напр.:
    - <T extends UpperBoundType>
    - <T super LowerBoundType>

ПРИМЕР

```
public class GenericNumberContainer <T extends Number> {
  private T obj;

  public GenericNumberContainer(){     }

  public GenericNumberContainer(T t)  {
     obj = t;
  }
  //  @return the obj

  public T getObj() {
     return obj;
  }

  // @param obj the obj to set

  public void setObj(T t) {
     obj = t;
  }
}
```

GenericNumberContainer специфицира, че генетичният тип наследява Number

ИЗПОЛЗВАНЕ

? Коментар

```
public class GenericNumberContainer <T extends Number> {
  private T obj;

  public GenericNumberContainer(){
  }

  public GenericNumberContainer(T t){
     obj = t;
  }
  /**
   * @retu
   */
  public T
     return
  }

  /**
   * @param obj the obj to set
   */
  public void setObj(T t) {
     obj = t;
  }
}
```

```
GenericNumberContainer<Integer> gn = new GenericNumberContainer<Integer>();
gn.setObj(3);

GenericNumberContainer<String> gn2 = new GenericNumberContainer<String>();
```

ГЕНЕТИЧНИ МЕТОДИ

- В предходните примери, методите в един генетичен клас могат да използват типизирани параметри
- Следователно автоматично генетични по отношение на тези параметри
- Възможно е също да се декларира генетичен метод, който използва един или повече собствени типизирани параметри
- Освен това е възможно да се създаваме генетични методи в негенетични класове

ГЕНЕТИЧНИ МЕТОДИ

- Възможно е, да не знаем типа на един аргумент, който се предава на метод
- Generics могат да се използват на ниво методи за решаване на такива ситуации
- Методите могат да съдържат:
  - Тип-аргументи
  - Тип-резултати (връщани стойности)

ПРИМЕР

```
public class Calculator {

    public static Integer addInteger(Integer a, Integer b)  {
        return a + b;
    }

    public static Float addFloat(Float a, Float b)  {
        return a + b;
    }

    public static <N extends Number> double add(N a, N b)
{
        double sum = 0;
        sum = a.doubleValue() + b.doubleValue();
        return sum;
    }
  }
```

Не-генетични методи

ПРИМЕР

```
public class Calculator {

    public static Integer addInteger(Integer a, Integer b)  {
        return a + b;
    }

    public static Float addFloat(Float a, Float b)  {
        return a + b;
    }

    public static <N extends Number> double add(N a, N b)  {
        double sum = 0;
        sum = a.doubleValue() + b.doubleValue();
        return sum;
    }
}
```

Генетичен метод

ИЗПОЛЗВАНЕ

```
public class CalculatorExample {
  public static void main(String[] args) {

    float floatValue = Calculator.addFloat(2f, 3f);
    System.out.println("Float Value: " + floatValue);

    int intValue = Calculator.addInteger(3, 4);
    System.out.println("Integer Value: " + intValue);

    double genericValue1 = Calculator.add(3, 3f);
    System.out.println("The int + float result: " + genericValue1);

    double genericValue2 = Calculator.add(7.54, 174);
    System.out.println("The double + int result: " + genericValue2);

    // Causes a ClassCastException because String is not a subtype of java.lang.Number
    //double genericValue3 = Calculator.add("Not valid", 3f);
    //System.out.println("The invalid result: " + genericValue3);
  }
}
```

```
Float Value: 5.0
Integer Value: 7
The int + float result: 6.0
The double + int result: 181.54
```

ГЕНЕТИЧНИ КОНСТРУКТОРИ

- Един конструктор може да бъде генеричен, дори неговият клас да не е
- Напр., в следващата програма, класът Summation не е генетичен, но неговият конструктор е

ПРИМЕР

```
class Summation {
      private int sum;
      <T extends Number> Summation(T arg) {
            sum = 0;
            for (int i = 1; i <= arg.intValue(); i++)
                  sum += i;
      }
      int getSum() {
            return sum;
      }
}

public class GenConsDemo {
        public static void main(String[] args) {
            Summation ob = new Summation(4.0);
            System.out.println("Sumation of 4.0 is " + ob.getSum());
        }
}
```

ПРИМЕР

```
class Summation {
      private int sum;
      <T extends Number> Summation(T arg) {
            sum = 0;
            for (int i = 1; i <= arg.intValue(); i++)
                  sum += i;
      }
      int getSum() {
            return sum;
      }
}

public class GenConsDemo {
        public static void main(String[] args) {
            Summation ob = new Summation(4.0);
            System.out.println("Sumation of 4.0 is " + ob.getSum());
        }
}
```

Класът изчислява сумата на числовата стойност, предадена на неговия конструктор
Сумирането на N е сумата на всички цели числа между 0 и N

ПРИМЕР

```
class Summation {
      private int sum;
      <T extends Number> Summation(T arg) {
            sum = 0;
            for (int i = 1; i <= arg.intValue(); i++)
                  sum += i;
      }
      int getSum() {
            return sum;
      }
}

public class GenConsDemo {
        public static void main(String[] args) {
            Summation ob = new Summation(4.0);
            System.out.println("Sumation of 4.0 is " + ob.getSum());
        }
}
```

Тъй като Summation() специфицира параметризиран тип, ограничен от Number, сумираният обект може да се конструира с помощта на който и да е цифров тип, включително int, float или double

ПРИМЕР

```
class Summation {
      private int sum;
      <T extends Number> Summation(T arg) {
            sum = 0;
            for (int i = 1; i <= arg.intValue(); i++)
                  sum += i;
      }
      int getSum() {
            return sum;
      }
}

public class GenConsDemo {
        public static void main(String[] args) {
            Summation ob = new Summation(4.0);
            System.out.println("Sumation of 4.0 is " + ob.getSum());
        }
}
```

Независимо какъв е използваният цифров тип, стойността му се преобразува в int от intValue() Следователно, не е необходимо класът да бъде генетичен – достатъчно е само генетичен конструктор

ПРИМЕР

```
class Summation {
      private int sum;
      <T extends Number> Summation(T arg) {
            sum = 0;
            for (int i = 1; i <= arg.intValue(); i++)
                  sum += i;
      }
      int getSum() {
            return sum;
      }
}

public class GenConsDemo {
        public static void main(String[] args) {
            Summation ob = new Summation(4.0);
            System.out.println("Sumation of 4.0 is " + ob.getSum());
        }
}
```

? Резултат

Sumation of 4.0 is 10

ГЕНЕТИЧНИ ИНТЕРФЕЙСИ

- В Java може да използват генетични интерфейси
- Задават се както генетичните класове

ПРИМЕР

```
interface Containment<T> {
      boolean contains(T o);
}

class MyClass<T> implements Containment<T> {
      T[] arrayRef;
      MyClass(T[] o) {  arrayRef = o;  }
      public boolean contains(T o) {
            for(T x : arrayRef)
              if(x.equals(o))return true;
            return false;
      }
}
```

```
public class GenIFDemo {
        public static void main(String[] args) {
            Integer x[] = { 1, 2, 3 };
            MyClass<Integer> ob = new MyClass<Integer>(x);
            if(ob.contains(2))
                System.out.println("2 is in ob");
            else
                System.out.println("2 is NOT in ob");

            if(ob.contains(5))
                System.out.println("5 is in ob");
             else
                System.out.println("5 is NOT in ob");

             //if(ob.contains(9.25))
             // System.out.println("9.25 is in ob");
        }
}
```

ПРИМЕР

```
interface Containment<T> {
      boolean contains(T o);
}

class MyClassN<T> implements Containment<T> {
      T[] arrayRef;
      MyClassN(T[] o) {  arrayRef = o;  }
      public boolean contains(T o) {
            for(T x:arrayRef)
              if(x.equals(o))return true;
            return false;
      }
}
```

? Резултат

```
public class GenIFDemo {
        public static void main(String[] args) {
              Integer x[] = { 1, 2, 3 };
              MyClassN<Integer> ob = new MyClassN<Integer>(x);
              if(ob.contains(2))
                  System.out.println("2 is in ob");
              else
                   System.out.println("2 is NOT in ob");
               if(ob.contains(5))
                   System.out.println("5 is in ob");
               else
                   System.out.println("5 is NOT in ob");
               //if(ob.contains(9.25))
               // System.out.println("9.25 is in ob");
          }
}
```

2 is in ob
5 is NOT in ob

МАСКИ

- В някои случаи е полезно да се пише код, който специфицира неизвестни типове
- Въпросителният знак “?” (символ за маска) може да се използва за представяне от неизвестен тип в генетичен код
- Маски могат да бъдат използвани с:
  - Параметри
  - Полета
  - Локални променливи
  - Типове на резултати

ПРИМЕР

```
import java.util.ArrayList;
import java.util.List;
public class WildcardExample  {
   public static void main(String[] args)  {
      List<Integer> intList = new ArrayList<Integer>();
      intList.add(2);
      intList.add(4);
      intList.add(6);

      List<String> strList = new ArrayList<String>();
      strList.add("two");
      strList.add("four");
      strList.add("six");

      List<Object> objList = new ArrayList<Object>();
      objList.add("two");
      objList.add("four");
      objList.add(strList);

      printList(intList);
      printList(strList);
      printList(objList);

      checkList(intList, 3);
      checkList(objList, strList);
      checkList(strList, objList);
      checkNumber(intList, 3);
}
```

ПРИМЕР

```
public static <T> void printList(List<T> myList)  {
    for(Object e : myList)  {
        System.out.println(e);
    }
}

public static <T> void checkList(List<?> myList, T obj)  {
    if(myList.contains(obj))  {
        System.out.println("The list " + myList + " contains the element: " + obj);
    } else {
        System.out.println("The list " + myList + " does not contain the element: " + obj);
    }
}

public static <T> void checkNumber(List<? extends Number> myList, T obj)  {
    if(myList.contains(obj)) {
        System.out.println("The list " + myList + " contains the element: " + obj);
    } else {
        System.out.println("The list " + myList + " does not contain the element: " + obj);
    }
}
}
```

ПРИМЕР

```
  public static <T> void printList(List<T> myList){
     for(Object e:myList){
        System.out.println(e);
     }
  }

  public static <T> void checkList(L
     if(myList.contains(obj)){
        System.out.println("The list " +
     } else {
        System.out.println("The list " +
     }
  }

  public static <T> void checkNuml
     if(myList.contains(obj)){
        System.out.println("The list " +
     } else {
        System.out.println("The list " + myList + " does not contain the element: " + obj);
     }
  }
}
```

```
2
4
6
two
four
six
two
four
[two, four, six]
The list [2, 4, 6] does not contain the element: 3
The list [two, four, [two, four, six]] contains the element: [two, four, six]
The list [two, four, six] does not contain the element: [two, four, [two, four, six]]
The list [2, 4, 6] does not contain the element: 3
```

ОЩЕ ПРИМЕРИ

```
public class GenericMethodTest
{
  // generic method printArray
  public static < E > void printArray( E[] inputArray )  {
    // Display array elements
      for ( E element : inputArray ){
        System.out.printf( "%s ", element );
      }
      System.out.println();
  }

  public static void main( String args[] )  {
    // Create arrays of Integer, Double and Character
    Integer[] intArray = { 1, 2, 3, 4, 5 };
    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
    Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

    System.out.println( "Array integerArray contains:" );
    printArray( intArray  ); // pass an Integer array

    System.out.println( "\nArray doubleArray contains:" );
    printArray( doubleArray ); // pass a Double array

    System.out.println( "\nArray characterArray contains:" );
    printArray( charArray ); // pass a Character array
  }
}
```

? Резултат

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

Array doubleArray contains:
1.1 2.2 3.3 4.4

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

ОЩЕ ПРИМЕРИ

? Резултат

```
public class MaximumTest  {
  // determines the largest of three Comparable objects
  public static <T extends Comparable<T>> T maximum(T x, T y, T z)   {
    T max = x; // assume x is initially the largest
    if ( y.compareTo( max ) > 0 ){
      max = y; // y is the largest so far
    }
    if ( z.compareTo( max ) > 0 )  {
      max = z; // z is the largest now
    }
    return max; // returns the largest object
  }
  public static void main( String args[] )   {
    System.out.printf( "Max of %d, %d and %d is %d\n\n",
              3, 4, 5, maximum( 3, 4, 5 ) );

    System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",
              6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

    System.out.printf( "Max of %s, %s and %s is %s\n","pear",
      "apple", "orange", maximum( "pear", "apple", "orange" ) );
  }
}
```

Max of 3, 4 and 5 is 5

Maxm of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear

ОБОБЩЕНИЕ

- Генетичните методи и генеричните класове в Java позволяват на програмистите да специфицира с една декларация на метод набор от свързани с тях методи или с декларация на един клас
  - Група от свързани типове респективно
- Generics осигуряват също безопасност по отношение на типовете по време на компилация
  - Позволява на програмистите да установяват невалидни видове по време на компилация

БЛАГОДАРЯ ЗА ВНИМАНИЕТО!

КРАЙ “ГЕНЕТИЧНО ПРОГРАМИРАНЕ”

Distributed @learning center