



Язык программирования Java

Мультимедийный курс

автор: Васильев А.Н.

`www.vasilev.kiev.ua`

Киев 2017



Обобщенные типы данных

- Знакомство с обобщенными классами
- Обобщенные методы
- Обобщенные классы и наследование
- Обобщенные интерфейсы
- Обобщенные подстановки



Обобщенные классы

3

- В обобщенных классах тип данных является параметром
- Обобщенный класс описывается как и обычный, но с формальным идентификатором, обозначающим тип данных. При создании объекта класса идентификатор типа передается как параметр
- Параметры, которые отождествляются в описании обобщенного класса с типом данных (обобщенные параметры), указываются в угловых скобках в описании класса после имени класса
- При создании объекта на основе обобщенного класса в угловых скобках в команде объявления объектной переменной указывают идентификаторы типа, которые следует использовать вместо обобщенных параметров
- Также идентификаторы указываются в угловых скобках в инструкции создания объекта - между названием класса и круглыми скобками с аргументами конструктора
- Значениями обобщенных параметров могут быть названия классов, но не могут быть идентификаторы для базовых типов: значениями обобщенных параметров идентификаторы `int` или `char` быть не могут, вместо них используют классы-оболочки `Integer` и `Character`



Обобщенные классы

4

Параметры, которые отождествляются в описании обобщенного класса с типом данных (обобщенные параметры), указываются в угловых скобках в описании класса после имени класса

Описание обобщенного класса

```
class MyClass<X,Y>{  
    // Описание обобщенного класса  
}
```

В команде объявления объектной переменной и в инструкции создания объекта в угловых скобках указывают идентификаторы типа

Создание объекта обобщенного класса

```
MyClass<Integer,Character> obj=new MyClass<Integer,Character>(аргументы) ;
```

Также приемлемо использовать упрощенный синтаксис: идентификаторы типов в команде вызова конструктора можно не указывать (но при этом угловые скобки должны присутствовать)

Создание объекта обобщенного класса

```
MyClass<Integer,Character> obj=new MyClass<>(аргументы) ;
```

Использование обобщенного класса

```
class MyClass<X>{           // Обобщенный класс с одним параметром типа
    X data;                 // Поле обобщенного типа
    MyClass(X data){       // Конструктор с аргументом обобщенного типа
        this.data=data;    // Присваивание значения полю
    }
    void show(){           // Метод для отображения значения поля
        System.out.println("Значение поля: "+data);
    }
} // Главный класс:
class UsingGenClassDemo{
    public static void main(String[] args){
        // Создание объекта на основе обобщенного класса.
        // Вместо обобщенного параметра используется идентификатор Integer:
        MyClass<Integer> A=new MyClass<Integer>(100);
        // Создание объекта на основе обобщенного класса.
        // Вместо обобщенного параметра используется идентификатор Character:
        MyClass<Character> B=new MyClass<Character>('B');
        // Создание объекта на основе обобщенного класса.
        // Вместо обобщенного параметра используется идентификатор String:
        MyClass<String> C=new MyClass<String>("Зеленый");
        // Вызов метода show() из объектов обобщенного класса:
        A.show();
        B.show();
        C.show();
    }
}
```

Результат

Значение поля: 100

Значение поля: B

Значение поля: Зеленый



Обобщенные классы

6

Использование обобщенного класса с несколькими параметрами - 1

```
// Обобщенный класс с одним параметром:
class Alpha<X>{
    private X first;        // Закрытое поле обобщенного типа
    Alpha(X first){          // Конструктор с аргументом обобщенного типа
        set(first);         // Вызов метода с аргументом обобщенного типа
    } // Метод с аргументом обобщенного типа для присваивания значения полю:
    void set(X first){
        this.first=first;
    } // Метод возвращает результатом значение поля:
    X get(){
        return first;
    }
}

// Обобщенный класс с двумя параметрами:
class Bravo<X,Y>{
    Alpha<X> obj;            // Поле - объект обобщенного класса
    Y second;               // Поле обобщенного типа
    Bravo(X first,Y second){ // Конструктор с двумя аргументами обобщенного типа
        obj=new Alpha<X>(first); // Создание объекта на основе обобщенного класса
        this.second=second;     // Присваивание значения полю
    } // Метод для отображения значений полей:
    void show(){
        System.out.println("Значения "+obj.get()+" и "+second);
    }
}

// Продолжение на следующем слайде!!!
```

Использование обобщенного класса с несколькими параметрами - 2

```
// Главный класс:
class UsingGenericsDemo{
    // Главный метод:
    public static void main(String[] args){
        // Создание объекта на основе обобщенного класса:
        Bravo<Integer,Character> A=new Bravo<Integer,Character>(100,'A');
        // Отображение значений полей объекта:
        A.show();
        // Создание объекта на основе обобщенного класса:
        Bravo<String,Double> B=new Bravo<>("BRAVO",12.345);
        // Отображение значений полей объекта:
        B.show();
    }
}
```

Результат

Значения 100 и A

Значения BRAVO и 12.345



Обобщенные методы

8

- Помимо обобщенных классов, можно создавать обобщенные методы
- В описании обобщенного метода тип определяется через обобщенный параметр (или параметры), а при вызове метода значения обобщенных параметров определяются на основе типов аргументов, переданных методу
- Обычно (но не обязательно) обобщенные методы являются статическими

Описание статического обобщенного метода

```
static <параметр> тип_результата имя_метода (аргументы) {  
    // Код метода  
}
```

Описание нестатического обобщенного метода

```
<параметр> тип_результата имя_метода (аргументы) {  
    // Код метода  
}
```

Использование обобщенного статического метода - 1

```
// Класс со статическими обобщенными методами:
class UsingStaticGenMethodDemo{
    // Метод с аргументом обобщенного типа:
    static <X> void show(X arg){
        System.out.println(arg);
    }
    // Аргумент метода - обобщенный массив:
    static <X> void show(X[] array){
        System.out.print("| ");
        // Отображение значений элементов массива:
        for(int k=0;k<array.length;k++){
            System.out.print(array[k]+" | ");
        }
        System.out.println("");
    }
    // Методу аргументом передается обобщенный массив и
    // целочисленный индекс, а результатом возвращается
    // значение элемента с соответствующим индексом:
    static <X> X getElement(X[] array,int index){
        return array[index];
    }
    // Продолжение на следующем слайде!!!
```

Использование обобщенного статического метода - 2

```
public static void main(String[] args){           // Главный метод
    Integer[] nums={1,3,7,2};                     // Целочисленный массив
    Character[] syms={'A','W','L','O','B'};        // Символьный массив
    System.out.println("Вызов метода show()");
    System.out.print("С текстовым аргументом: ");
    show("обобщенный метод");
    System.out.print("С int-аргументом: ");
    show(123);
    System.out.print("С double-аргументом: ");
    show(123.45);
    System.out.print("С char-аргументом: ");
    show('A');
    System.out.print("Целочисленный массив: ");
    show(nums);
    System.out.print("Символьный массив: ");
    show(syms); // Поэлементное отображение массивов:
    System.out.println("Вызов метода getElement()");
    System.out.print("Целочисленный массив: *");
    for(int k=0;k<nums.length;k++){
        System.out.print(getElement(nums,k)+"*");
    }
    System.out.print("\nСимвольный массив: *");
    for(int k=0;k<syms.length;k++){
        System.out.print(getElement(syms,k)+"*");
    }
    System.out.println("");}}
```

Использование обобщенного статического метода - результат

Вызов метода `show()`

С текстовым аргументом: обобщенный метод

С `int`-аргументом: 123

С `double`-аргументом: 123.45

С `char`-аргументом: A

Целочисленный массив: | 1 | 3 | 7 | 2 |

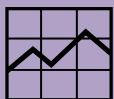
Символьный массив: | A | W | L | O | B |

Вызов метода `getElement()`

Целочисленный массив: *1*3*7*2*

Символьный массив: *A*W*L*O*B*

Детали



Работа с обобщенными типами полна сюрпризов. Причина "неожиданностей" во многом связана с механизмом "стирания", который используется при компиляции программного кода с обобщенными методами и/или классами. Ситуация такая, что значения, используемые для обобщенных параметров, известны на момент компиляции, но после компиляции программы в байт-код эта информация "стирается". Проще говоря, на этапе выполнения программного кода в рамках обобщенного класса или метода нельзя явно идентифицировать используемое значение обобщенного параметра (но это можно сделать на этапе компиляции). Поэтому, например, мы можем объявить в теле метода или в классе объектную переменную, относящуюся к обобщенному типу, но не можем создать объект на основе обобщенного типа. Причина в том, что переменная создается на этапе компиляции, а объект - в процессе выполнения программы. По той же причине мы можем использовать переменные массива с элементами обобщенного типа, но не можем создать массив с элементами обобщенного типа.

Мы можем описать обобщенный класс, в котором используется метод с обобщенными параметрами типа. Но можно поступить и иначе: класс описывать как обычный, но при этом один или более методов описать как обобщенные

Описание статического обобщенного метода

```
// Класс с обобщенным методом:
class MyClass{
    String name;                // Текстовое поле
    <X> void show(X arg){        // Обобщенный метод
        System.out.println(name+": "+arg);
    }
    MyClass(String txt){        // Конструктор
        name=txt;
    }
} // Главный класс:
class UsingGenMethodsDemo{
    public static void main(String[] args){ // Главный метод
        MyClass A=new MyClass("Объект А"); // Создание объектов
        MyClass B=new MyClass("Объект В");
        A.show(123);              // Вызов обобщенных методов из разных объектов
        A.show("Alpha");
        A.show('A');
        B.show(123);
        B.show("Bravo");
        B.show('B');
    }
}
```

Результат

Объект А: 123
Объект А: Alpha
Объект А: A
Объект В: 123
Объект В: Bravo
Объект В: B

Суперкласс на основе обобщенного класса - 1

```
// Обобщенный класс:
class Base<X>{
    // Обобщенное поле:
    X data;
    // Конструктор:
    Base(X data){
        this.data=data;
    } // Метод для отображения значения поля:
    void show(){
        System.out.println(data);
    }
}
// Подкласс на основе обобщенного класса с целочисленным
// типом вместо обобщенного:
class Alpha extends Base<Integer>{
    // Конструктор:
    Alpha(Integer n){
        // Вызов конструктора суперкласса:
        super(n);
    } // Переопределение метода:
    void show(){
        System.out.print("Целочисленное поле: ");
        // Вызов версии метода из суперкласса:
        super.show();
    }
} // Продолжение на следующем слайде!!!
```

Суперкласс на основе обобщенного класса - 2

```
// Подкласс на основе обобщенного класса с текстовым типом вместо обобщенного:
class Bravo extends Base<String>{
    // Конструктор:
    Bravo(String txt){
        // Вызов конструктора суперкласса:
        super(txt);
    } // Переопределение метода:
    void show(){
        System.out.print("Текстовое поле: ");
        // Вызов версии метода из суперкласса:
        super.show();
    }
}

// Подкласс на основе обобщенного класса с символьным типом вместо обобщенного:
class Charlie extends Base<Character>{
    // Конструктор:
    Charlie(Character s){
        // Вызов конструктора суперкласса:
        super(s);
    } // Переопределение метода:
    void show(){
        System.out.print("Символьное поле: ");
        // Вызов версии метода из суперкласса:
        super.show();
    }
} // Продолжение на следующем слайде!!!
```

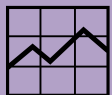
Суперкласс на основе обобщенного класса - 3

```
// Главный класс:
class ExtendingGenClassDemo{
    // Главный метод:
    public static void main(String[] args){
        // Создание объектов:
        Alpha A=new Alpha(123);
        Bravo B=new Bravo("объект В");
        Charlie C=new Charlie('C');
        // Вызов метода show() из разных объектов:
        A.show();
        B.show();
        C.show();
    }
}
```

Результат

Целочисленное поле: 123
Текстовое поле: объект В
Символьное поле: C

Детали



Допустим, что обобщенный суперкласс описан следующим образом:

```
class Base<X>{  
    X data;  
}
```

Рассмотрим два подкласса. Подкласс **Alpha** описан так:

```
class Alpha extends Base<Integer>{}
```

Еще один подкласс **Bravo** опишем следующим образом:

```
class Bravo extends Base{}
```

Принципиальное отличие классов **Alpha** и **Bravo** в том, что при создании **Alpha** для обобщенного суперкласса **Base** указано значение **Integer** для обобщенного типа, а при создании класса **Bravo** значение для обобщенного типа не указано. В результате у объектов класса **Alpha** поле **data** будет целочисленным (типа **Integer**), а у объектов класса **Bravo** поле **data** будет относиться к типу **Object** (суперкласс в вершине иерархии всех классов).

В обобщенных классах и методах в инструкции объявления обобщенного типа может использоваться выражение вида X extends Суперкласс, означающее, что обобщенный тип X должен быть подклассом (прямым или опосредованным) класса Суперкласс

Ограничение наследования для обобщенного типа - 1

```
class Base{                                // Суперкласс
    String name;                           // Текстовое поле
    Base(String txt){                      // Конструктор
        name=txt;
    } // Метод для отображения значения поля:
    void show(){
        System.out.println("Текстовое поле: "+name);
    }
} // Подкласс суперкласса Base:
class Alpha extends Base{
    int number;                            // Целочисленное поле
    Alpha(String txt,int n){              // Конструктор
        super(txt);                      // Вызов конструктора суперкласса
        number=n;                        // Присваивание значения целочисленному полю
    } // Переопределение метода:
    void show(){
        super.show();                    // Вызов версии метода из суперкласса
        // Отображение значения целочисленного поля:
        System.out.println("Целочисленное поле: "+number);
    }
} // Продолжение на следующем слайде!!!
```

Ограничение наследования для обобщенного типа - 2

```
// Подкласс суперкласса Alpha:
class Bravo extends Alpha{
    // Символьное поле:
    char symbol;
    // Конструктор:
    Bravo(String txt,int n,char s){
        // Вызов конструктора суперкласса:
        super(txt,n);
        // Присваивание значения символьному полю:
        symbol=s;
    }
    // Переопределение метода:
    void show(){
        // Вызов версии метода из суперкласса:
        super.show();
        // Отображение значения символьного поля:
        System.out.println("Символьное поле: "+symbol);
    }
}
// Продолжение на следующем слайде!!!
```

Ограничение наследования для обобщенного типа - 3

```
// Обобщенный класс:
class MyClass<X extends Base>{
    // Поле обобщенного типа:
    X obj;
    // Конструктор:
    MyClass(X obj){
        // Значение поля:
        this.obj=obj;
    } // Метод обобщенного класса:
    void show(){
        System.out.println("Объект класса MyClass");
        // Вызов метода из поля обобщенного типа:
        obj.show();
    }
} // Главный класс:
class GenTypeExtendingDemo{
    // Главный метод:
    public static void main(String[] args){
        // Объекты создаются на основе обобщенного класса:
        MyClass<Alpha> A=new MyClass<>(new Alpha("Alpha",123));
        MyClass<Bravo> B=new MyClass<>(new Bravo("Bravo",321,'B'));
        // Вызов метода show() из объектов:
        A.show();
        B.show();
    }
}
```

Результат

Объект класса MyClass
Текстовое поле: Alpha
Целочисленное поле: 123
Объект класса MyClass
Текстовое поле: Bravo
Целочисленное поле: 321
Символьное поле: B

☒ На заметку

Инструкция `super.show()` в классе `Alpha` означает вызов версии метода `show()` из класса `Base`, поскольку он является суперклассом для класса `Alpha`. Инструкция `super.show()` в классе `Bravo` означает вызов версии метода `show()` из класса `Alpha`, поскольку для класса `Bravo` суперклассом является класс `Alpha`.

Детали



Полю `obj` присваивается значение, переданное аргументом конструктору класса `MyClass`. Учитывая, что значением параметра `X` может быть имя класса (наследующего `Base`), легко понять, что аргументом конструктору передается объектная переменная, значением которой является ссылка на некоторый объект и, следовательно, полю `obj` присваивается ссылка на этот объект. Важно здесь то, что объект для поля `obj` не создается в конструкторе, а ссылка на него лишь передается в конструктор. Это принципиально важный момент, поскольку в обобщенном классе нельзя создать объект на основе обобщенного типа.

- Обобщенным может быть не только класс или метод, но и интерфейс
- Объявляется обобщенный интерфейс аналогично к обобщенному классу: после имени интерфейса в угловых скобках указываются названия для обобщенных типов
- Обобщенный интерфейс может использоваться как для создания обобщенного класса, так и для создания класса с конкретными значениями для обобщенных параметров типа

Обобщенный интерфейс - 1

```
interface MyMethods<X>{           // Обобщенный интерфейс
    X get();
    void set(X arg);
} // Создание обобщенного класса на основе обобщенного интерфейса:
class MyClass<X> implements MyMethods<X>{
    private X value;               // Закрытое поле обобщенного типа
    public X get(){                 // Описание методов из интерфейса
        return value;
    }
    public void set(X arg){
        value=arg;
    } // Метод для отображения значения поля:
    void show(){
        System.out.println("Значение поля: "+get());
    } // Конструктор:
    MyClass(X arg){
        value=arg;
    } // Продолжение на следующем слайде!!!
```

Обобщенный интерфейс - 2

```
// Главный класс:
class GenInterfaceDemo{
    // Главный метод:
    public static void main(String[] args){
        // Интерфейсная переменная:
        MyMethods ref;
        // Создание объектов на основе обобщенного класса:
        MyClass<Integer> A=new MyClass<>(123);
        MyClass<Character> B=new MyClass<>('A');
        // Вызов метода из объекта класса:
        A.show();
        // Интерфейсной переменной присваивается значение:
        ref=A;
        // Вызов метода через интерфейсную переменную:
        ref.set(321);
        // Вызов методов из объектов класса:
        A.show();
        B.show();
        // Интерфейсной переменной присваивается значение:
        ref=B;
        // Вызов метода через интерфейсную переменную:
        ref.set('B');
        // Вызов метода из объекта класса:
        B.show();
    }
}
```

Результат

Значение поля: 123
Значение поля: 321
Значение поля: A
Значение поля: B

Создание обычного класса на основе обобщенного интерфейса - 1

```
// Обобщенный интерфейс:
interface MyMethods<X>{
    X get();
    void set(X arg);
}

// Создание первого класса на основе
// обобщенного интерфейса:
class Alpha implements MyMethods<Integer>{
    // Закрытое поле целочисленного типа:
    private Integer value;
    // Описание методов из интерфейса:
    public Integer get(){
        return value;
    }
    public void set(Integer arg){
        value=arg;
    }
    // Метод для отображения значения поля:
    void show(){
        System.out.println("Целочисленное поле: "+get());
    }
    // Конструктор:
    Alpha(Integer arg){
        value=arg;
    }
} // Продолжение на следующем слайде!!!
```



Обобщенные интерфейсы

25

Создание обычного класса на основе обобщенного интерфейса - 2

```
// Создание второго класса на основе
// обобщенного интерфейса:
class Bravo implements MyMethods<Character>{
    // Закрытое поле символьного типа:
    private Character value;
    // Описание методов из интерфейса:
    public Character get(){
        return value;
    }
    public void set(Character arg){
        value=arg;
    }
    // Метод для отображения значения поля:
    void show(){
        System.out.println("Символьное поле: "+get());
    }
    // Конструктор:
    Bravo(Character arg){
        value=arg;
    }
}
// Продолжение на следующем слайде!!!
```



Обобщенные интерфейсы

26

Создание обычного класса на основе обобщенного интерфейса - 3

```
// Главный класс:
class MoreGenInterfaceDemo{
    // Главный метод:
    public static void main(String[] args){
        // Интерфейсная переменная:
        MyMethods ref;
        // Создание объектов:
        Alpha A=new Alpha(123);
        Bravo B=new Bravo('A');
        // Вызов метода из объекта класса:
        A.show();
        // Интерфейсной переменной присваивается значение:
        ref=A;
        // Вызов метода через интерфейсную переменную:
        ref.set(321);
        // Вызов методов из объектов класса:
        A.show();
        B.show();
        // Интерфейсной переменной присваивается значение:
        ref=B;
        // Вызов метода через интерфейсную переменную:
        ref.set('B');
        // Вызов метода из объекта класса:
        B.show();
    }
}
```

Результат

Целочисленное поле: 123
Целочисленное поле: 321
Символьное поле: A
Символьное поле: B



Обобщенные интерфейсы

27

Интерфейс `MyMethods` является обобщенным. Интерфейсная переменная `ref` объявляется командой `MyMethods ref`, в которой указано только основное имя интерфейса без указания значения для обобщенного типа. Благодаря этому значением переменной `ref` можно присвоить ссылку как на объект `A`, так и на объект `B` - несмотря на то, что объект `A` создавался на основе класса `Alpha`, реализующего интерфейс `MyMethods` со значением `Integer` для обобщенного типа, а объект `B` создается на основе класса `Bravo`, реализующего интерфейс `MyMethods` со значением `Character` для обобщенного типа. У такой "универсальности" есть цена: обобщенный тип при обращении к объектам через интерфейсную переменную `ref` не специфицирован. Поэтому, например, метод `get()`, который в соответствии со своим объявлением в интерфейсе, возвращает значение обобщенного типа, а для объектов `A` и `B` описан возвращающим соответственно значения типа `Integer` и `Character`, при вызове его через переменную `ref` интерпретируется как возвращающий значение типа `Object`. Это не мешает, скажем, отображать в консольном окне значение, возвращаемое методом `get()`, но накладывает ограничения на допустимые операции с результатом метода (правда, обычно на выручку может прийти явное приведение типа).

С другой стороны, мы могли бы, скажем, объявить переменную `ref` с помощью команды `MyMethods<Integer> ref`, в которой для интерфейса `MyMethods` явно указано значение `Integer` для параметра обобщенного типа. Но в таком случае значением переменной `ref` можно было бы присвоить лишь ссылку на объект `A`, но не на объект `B`. Зато при вызове метода `get()` через переменную `ref` результат метода интерпретируется как значение типа `Integer`



Обобщенные подстановки

28

Идея обобщенной подстановки состоит в том, что при определении обобщенного параметра используют инструкцию вида `<?>`. Это сообщает компилятору, что в соответствующем месте используется некоторый неизвестный обобщенный тип

Реализуем такую программу:

В программе определен обобщенный класс `MyClass` с одним обобщенным параметром (обозначим его как `T`)

В классе объявлено поле `value` обобщенного типа `T` и конструктор с одним аргументом обобщенного типа. Аргумент конструктора присваивается значением полю `value`

В классе `UsingWildcardDemo` описывается главный метод `main()` и два статических не возвращающих результат метода: `show()` и `display()`. Они выполняют практически одинаковую "работу", но описаны по-разному

Оба метода предназначены для отображения значения поля `value` объекта, созданного на основе обобщенного класса `MyClass` и переданного аргументом методу

В главном методе программы приведены примеры создания объектов на основе обобщенного класса `MyClass` с последующей их передачей аргументами методам `show()` и `display()`



Использование обобщенных подстановок - 1

```
// Обобщенный класс:
class MyClass<T>{
    // Поле обобщенного типа:
    T value;
    // Конструктор:
    MyClass(T val){
        value=val;
    }
}

// Главный класс:
class UsingWildcardDemo{
    // Статический обобщенный метод
    // (используется параметр обобщенного типа):
    static <T> void show(MyClass<T> obj){
        System.out.println("Вызов метода show():");
        // Отображение значения поля:
        System.out.println(obj.value);
    }
    // Статический метод, в котором используется
    // обобщенная подстановка:
    static void display(MyClass<?> obj){
        System.out.println("Вызов метода display():");
        // Отображение значения поля:
        System.out.println(obj.value);
    }
}

// Продолжение на следующем слайде!!!
```

Использование обобщенных подстановок - 2

```
// Главный метод:
public static void main(String[] args){
    // При создании объекта явно указано значение
    // для обобщенного типа:
    MyClass<Integer> A=new MyClass<>(100);
    // При создании объекта не указано значение
    // для обобщенного типа:
    MyClass B=new MyClass<>('B');
    // При создании объекта использована
    // обобщенная подстановка:
    MyClass<?> C=new MyClass<>("Объект C");
    // Примеры вызова методов show() и display()
    // с разными аргументами:
    show(A);
    display(A);
    show(B);
    display(B);
    show(C);
    display(C);
}
```

Во всех трех случаях мы можем отобразить значение поля, но операции, допустимые для выполнения с полем value, зависят от способа создания объекта и различны для объектов A, B и C

Результат

Вызов метода show():
100

Вызов метода display():
100

Вызов метода show():
B

Вызов метода display():
B

Вызов метода show():
Объект C

Вызов метода display():
Объект C



Обобщенные подстановки

31

Разница между способами объявления объектов **A**, **B** и **C**:

- При создании объекта **A** мы явно указано значение **Integer** для параметра обобщенного типа, поэтому поле **value** созданного объекта интерпретируется как **Integer**-значение
- При создании объекта **B** для объектной переменной указан только основной тип (имя класса **MyClass**). Поэтому поле **value** созданного объекта интерпретируется как значение типа **Object**
- При создании объекта **C** использована обобщенная подстановка. Для этого объекта значение обобщенного параметра не идентифицируется. Технически в таком случае используется формальное "рабочее" обозначение для значения параметра обобщенного типа - то есть для параметра обобщенного типа выполняется "подстановка", которая играет роль типа. **Такие подстановки используются, если выполнение кода не зависит от значения обобщенного параметра**

При использовании обобщенных подстановок можно использовать ограничения: с помощью ключевого слова `extends` определяют подстановки, обозначающие подклассы для данного класса, а с помощью ключевого слова `super` определяют подстановки для суперклассов данного класса

- Выражение вида `<? extends SuperClass>` означает, что для параметра обобщенного типа используется неустановленный тип, являющийся подклассом (прямым или опосредованным) класса `SuperClass`
- Выражение вида `<? super SubClass>` означает, что в качестве параметра обобщенного типа может быть класс, являющийся (прямо или через цепочку наследования) суперклассом для класса `SubClass`

Использование обобщенных подстановок с ограничениями - 1

```
// Первый класс:
class Alpha{
    // Закрытое текстовое поле:
    private String name;
    // Конструктор:
    Alpha(String txt){
        name=txt;
    }
    // Переопределение метода toString():
    public String toString(){
        return name;
    }
} // Второй класс:
class Bravo extends Alpha{
    // Конструктор:
    Bravo(String txt){
        // Вызов конструктора суперкласса:
        super(txt);
    }
} // Третий класс:
class Charlie extends Bravo{
    Charlie(String txt){
        super(txt);
    }
}
// Продолжение на следующем слайде!!!
```



Обобщенные подстановки с ограничениями

34

Использование обобщенных подстановок с ограничениями - 2

```
// Четвертый класс:
class Delta extends Charlie{
    Delta(String txt){
        super(txt);
    }
}

// Пятый класс:
class Echo extends Delta{
    Echo(String txt){
        super(txt);
    }
}

// Обобщенный класс:
class MyClass<T>{
    // Закрытое поле обобщенного типа:
    private T obj;
    // Переопределение метода toString():
    public String toString(){
        return obj.toString();
    }
    // Конструктор:
    MyClass(T arg){
        obj=arg;
    }
}

// Продолжение на следующем слайде!!!
```

Использование обобщенных подстановок с ограничениями - 3

```
class BoundedWildcardsDemo{    // Главный класс
    // Статический метод для отображения текстового представления объекта,
    // созданного с использованием подкласса для класса Charlie:
    static void show(MyClass<? extends Charlie> obj){
        System.out.println(obj);
    } // Статический метод для отображения текстового представления объекта,
    // созданного с использованием суперкласса для класса Charlie:
    static void display(MyClass<? super Charlie> obj){
        System.out.println(obj);
    } // Главный метод:
    public static void main(String[] args){// Создание объектов:
        MyClass<Alpha> A=new MyClass<>(new Alpha("Объект А"));
        MyClass<Bravo> B=new MyClass<>(new Bravo("Объект В"));
        MyClass<Charlie> C=new MyClass<>(new Charlie("Объект С"));
        MyClass<Delta> D=new MyClass<>(new Delta("Объект D"));
        MyClass<Echo> E=new MyClass<>(new Echo("Объект Е"));
        // Вызов методов display() и show() с передачей
        // аргументом одного из созданных объектов:
        display(A);
        display(B);
        display(C);
        show(C);
        show(D);
        show(E);
    }
}
```

Результат

Объект А
Объект В
Объект С
Объект С
Объект D
Объект Е

- Проанализировать программный код примеров, представленных в презентации
- Набрать код и проверить его функциональность

Продолжение следует...