Интерфейсы

Интерфейс – это абстрактный тип, позволяющий иметь различные реализации методов в разных классах и обращаться через него к объекту.

Интерфейсы имеют следующие ограничения:

- Модификатор доступа — может быть только public или отсутствовать (тогда, по умолчанию, интерфейс доступен только членам пакета, в котором он объявлен).
- Методы — могут быть абстрактными, или **начиная с java 8** по умолчанию (имеют реализацию, можно переопределить) и статическими (имеют реализацию, нельзя переопределить, нельзя использовать для объектов класса реализации);
- Поля — final, static (константы, не меняющие значений, такие спецификации для них назначаются автоматически, должны быть инициализированы постоянными значениями);
- Нельзя создать объект типа интерфейса (но можно использовать в качестве типа — интерфейсные ссылки).

[модификатор] **interface** ИмяНовогоИнтерфейса

[**extends** список Интерфейсов]

{Тело интерфейса, состоящее из описаний абстрактных методов и констант}

Сравнение абстрактных классов и интерфейсов

Возможность	Абстрактный класс	Интерфейс
Множественное наследование/реализация	-	+
Декларация абстрактных методов	+	+
Реализация конкретных методов	+	- (до Java 8)
Объявление констант	+	+
Объявление полей	+	-
Определение конструкторов	+	-
Создание экземпляров	-	-
Область видимости элементов	любая	public

Интерфейсные константы

Интерфейсы можно использовать для импорта в различные классы совместно используемых констант.

```
public interface MyConstants
        {
    public static final double price = 1450.00;
    public static final int counter = 5;
        }
interface MyColors {
// по умолчанию public static final
    int RED = 1, YELLOW = 2, BLUE = 4;
}
```

Реализация методов интерфейса

```
public interface MyInterface
        {
          void volume(int x,int y, int z);
           default void add(int x, int y) {System.out.println(+(x+y)); }
        }
class Demo1 implements MyInterface
    {
    public void add(int x, int y)
        {
    System.out.println(  +(x-y));
        }

    public void volume(int x, int y, int z)
        {
    System.out.println(      +(x*y*z));
        }

public static void main(String args[])
        {
     MyInterface  d1= new Demo1();
    d1.add(10,20);
    d1.volume(10,10,10);
        } }
```

```
class Demo2 implements MyInterface
{
public void add(int x, int y)
{
     System.out.println(      +(x*y));
  }

public void volume(int x, int y, int z)
{
System.out.println(      +(x-y-z));
}
```

```
MyInterface d2= new Demo2();
d2.add(10,20);
d2.volume(10,10,10);
```

- Интерфейс можно использовать как ссылочный тип при объявлении переменных.
- Переменная или выражение типа интерфейса могут ссылаться на любой объект, который является экземпляром класса, реализующего данный интерфейс.
- Переменную типа интерфейса можно использовать только после присвоения ей ссылки на объект ссылочного типа, для которого был реализован данный интерфейс.

Статические методы в интерфейсах

1. Статические методы в интерфейсе являются частью интерфейса, их нельзя использовать для объектов класса реализации и нельзя переопределить или изменить в классе реализации. Если метод того же имени реализован в классе реализации, то этот метод становится статическим членом этого соответствующего класса.
2. Статические методы в интерфейсе хороши для обеспечения вспомогательных методов, например, проверки на null, сортировки коллекций и т.д.
3. Статические методы в интерфейсе помогают обеспечивать безопасность, не позволяя классам, которые реализуют интерфейс, переопределить их.
4. Можно использовать статические методы интерфейса, чтобы не создавать вспомогательные классы, то есть переместить все статические методы в соответствующий интерфейс. Такой метод легко использовать и быстро находить.

Пример описания статического метода в интерфейсе

```
package pro.java.staticm;

interface MyInt {

    static void prt1() {
        System.out.println("PRT1 in MyInt");
    }

    void prt2();
}

class Prt implements MyInt {
    @Override     // аннотация переопределение метода prt2  в классе Ptr
    public void prt2() {
        System.out.println("PRT2 in Prt");
    }
}

public class Interf01 implements MyInt{

    public static void main(String[] args) {

        // вызов статического метода на интерфейсе
        MyInt.prt1();

        Prt prt = new Prt();
        prt.prt2();
        // prt.prt1(); // ошибка

        Interf01 ifs = new Interf01();
        ifs.prt2();

    }

    @Override
    public void prt2() {
        System.out.println("PRT2 in main()");
    }

}
```

```
PRT1 in MyInt
PRT2 in Prt
PRT2 in main()
```

PTR1 in Interf01

Статические методы видны только для интерфейса, где они объявлены

Interf01.ptr1();

// Переопределение метода prt2 в классе Interf01

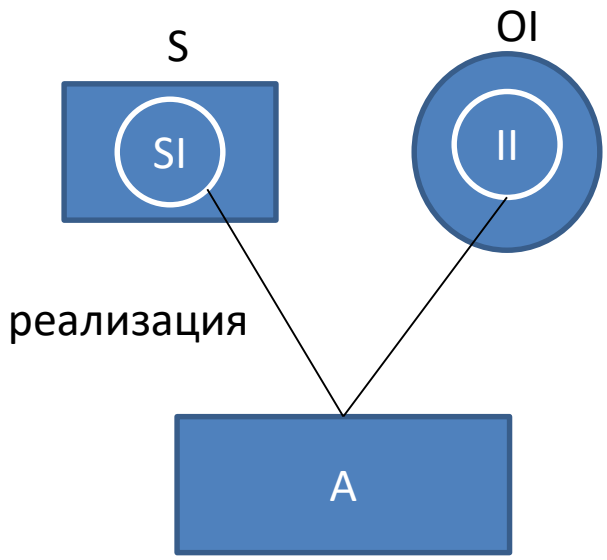
// статический метод класса Interf01
static void ptr1() {
System.out.println(“PTR1 in Interf01”);
}

Вложенные интерфейсы

Можно вкладывать описание интерфейса внутрь описания класса или другого интерфейса.

//описание интерфейса внутри класса

```
class S {
 void MethodS() { }
 interface SI {
 void SIMethod();
 }
}
```



//описание интерфейса внутри другого интерфейса

```
interface OI {
 void OIMethod();
 //описание вложенного интерфейса
 interface II {
 void IIMethod();
 }
}
```

// реализация вложенных интерфейсов

class A implements **OI.II**, **S.SI** {
… // реализация методов IIMethod и SIMethod
}

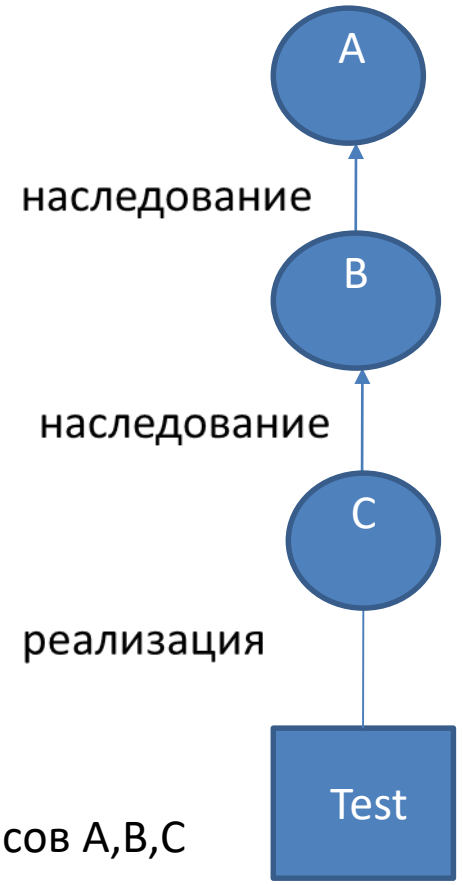
// Использование вложенного интерфейса идет через имя внешнего класса или интерфейса:

```
A a1=new A();
S.SI si = a1;
si.SIMethod();

OI.II ii = a1;
ii.IIMethod();
```

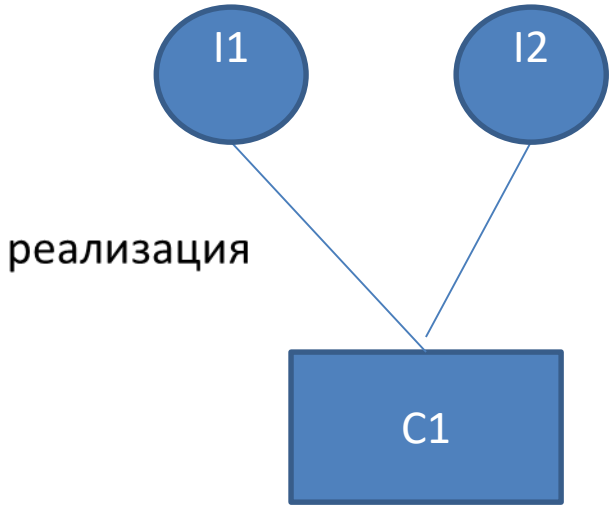
Наследование интерфейсов

```
//суперинтерфейс A
interface A {
 int a_value = 1;
 void A();
}
 //интерфейс B расширяет интерфейс A
interface B extends A{
 int b_value = 2;
 void B();
}
 //интерфейс C расширяет интерфейс B
interface C extends B{
 int c_value = 3;
 void C();
}
// класс Test должен реализовать методы интерфейсов A,B,C
class Test implements C{ . . . }
Test t = new Test();
 t.A();
 t.B();
 t.C();
```



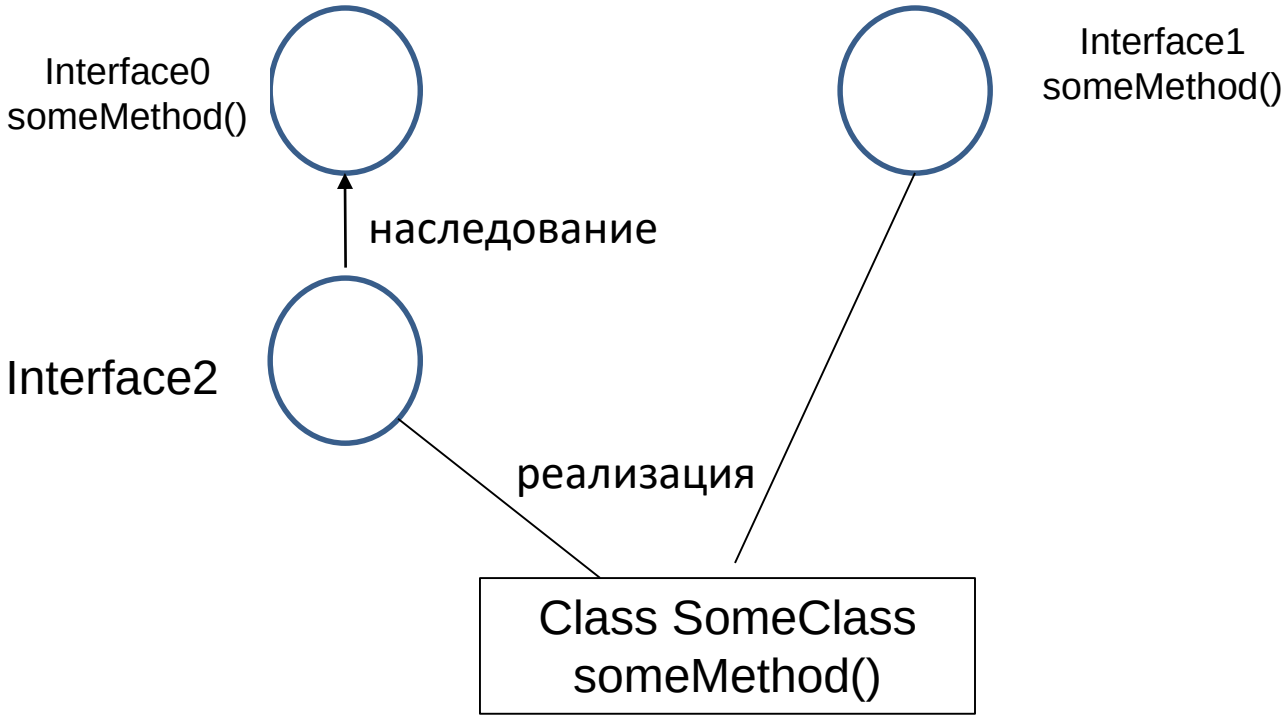
Использование констант при множественном наследовании интерфейсов

```
public interface I1 {
    Double PI=3.14;
}
public interface I2 {
    Double PI=3.1415;
}
class C1 implements I1,I2 {
  void m1(){
    System.out.println("I1.PI="+ I1.PI);
    System.out.println("I2.PI="+ I2.PI);
  };
}
```



Для использования констант с одинаковыми именами из разных интерфейсов необходима квалификация имени константы именем соответствующего интерфейса

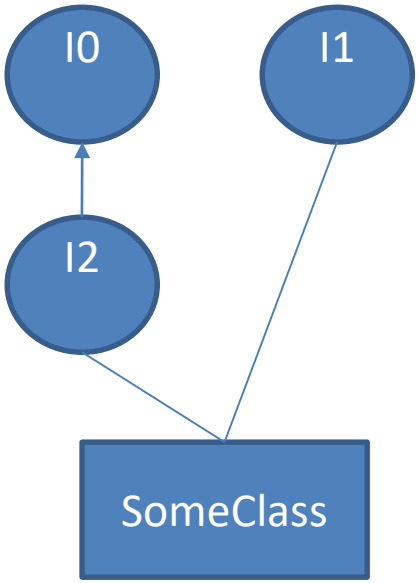
Наследование интерфейсов и реализация интерфейсов



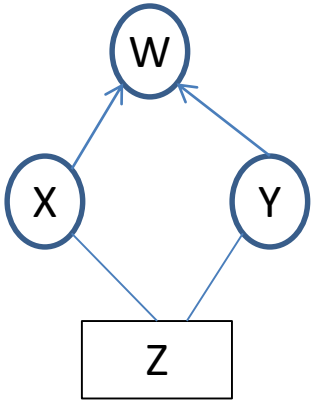
- Класс должен полностью реализовать все методы интерфейса, либо часть методов, но в этом случае должен быть объявлен как абстрактный.
- Если класс реализует несколько интерфейсов, в которых есть одноимённые методы, то в нём может задаваться лишь одна реализация общая для всех этих методов

Использование переменных типа интерфейс

```
interface Interface0 {    int someField = 10;     String someMethod(); }
interface Interface1 {     int someField = 100;     String someMethod(); }
interface Interface2 extends Interface0 {  int someField = 200;     String someMethod(); }
class SomeClass implements Interface1, Interface2 {
    public String someMethod() {    return "Метод";     }
}
public class Main {
      public static void main(String[] args) {
                SomeClass a = new SomeClass();
       Interface1 I1=a;
         System.out.println( a.someMethod() );       // Метод
    //   System.out.println( a.someField );  // неоднозначность ошибка
      System.out.println( ( (Interface1) a).someField );  // 100
       System.out.println( Interface1.someField );        // 100
      Interface2 I2=a;
      System.out.println( I2.someField );             // 200
       System.out.println( I2.someMethod() );     // Метод
       Interface0 I0=a;
       System.out.println( I0.someField );            // 10
       System.out.println( Interface0.someField ); // 10
 System.out.println( I0.someMethod() );           // Метод
      }
}
```



Конфликты имен в реализации интерфейсов



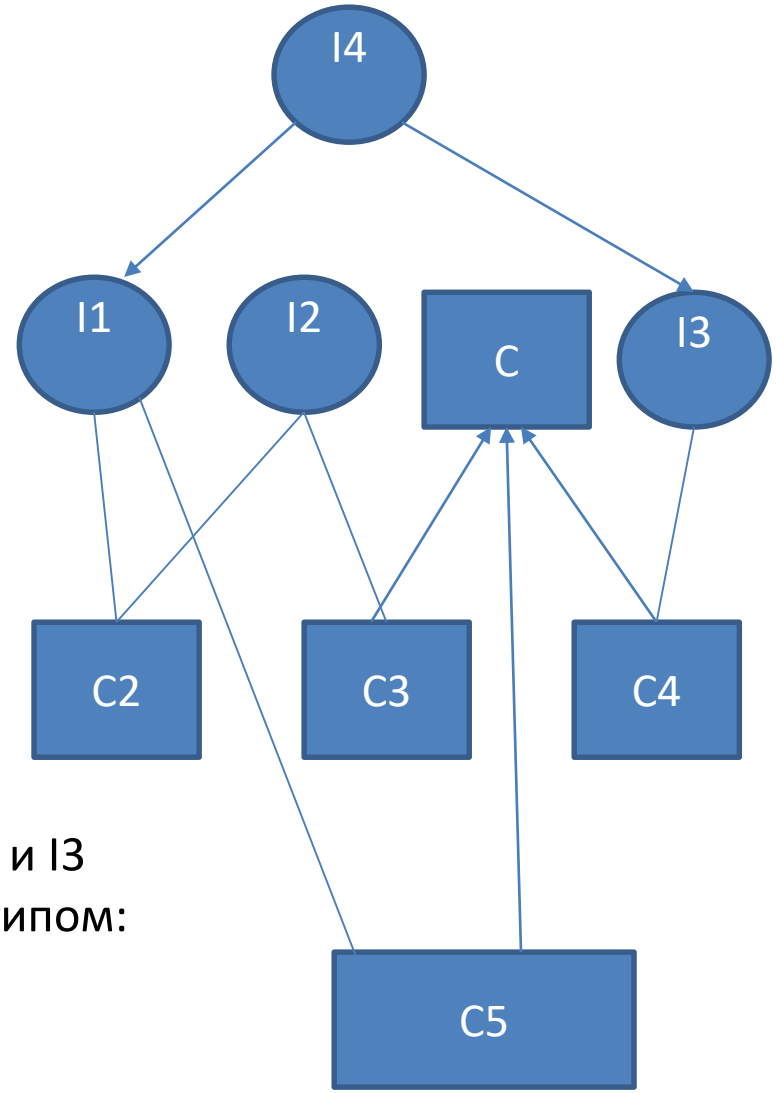

```
interface W { }
interface X extends W { }
interface Y extends W { }
class Z implements X, Y { }
```

Если интерфейсы X и Y содержат одноименные методы с разным количеством или типом параметров, то Z будет содержать два перегруженных метода с одинаковыми именами, но разными сигнатурами.
Если же сигнатуры в точности совпадают, то Z может содержать лишь один метод с данной сигнатурой.
Если методы отличаются лишь типом возвращаемого значения, вы не можете реализовать оба интерфейса.

Если два метода отличаются только типом возбуждаемых исключений, метод класса обязан соответствовать обоим объявлениям с одинаковыми сигнатурами , но может иметь не больший список возможных исключений.

Пример конфликтов имен

```
interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C {
public int f() { return 1; }
}
class C2 implements I1, I2 {
public void f() {} //реализация I1
public int f(int i) { return 1; } //реализация I2
}
class C3 extends C implements I2 {
public int f(int i) { return 1; } // реализация I2
}
class C4 extends C implements I3 {
public int f() { return 2; } } // f () переопределена C и I3
 // Методы различаются только возвращаемым типом:
class C5 extends C implements I1 {} //Ошибка
interface I4 extends I1, I3 {} //Ошибка
```



Функциональные интерфейсы (Java 8)

Это интерфейс, который определяет только **один абстрактный метод** (также может объявлять абстрактные методы из класса java.lang.Object) и дефолтные и static методы (не абстрактные). Аннотация не позволяет добавлять новые абстрактные методы.
@FunctionalInterface

```
public interface Runnable {
    void run();
}
@FunctionalInterface
interface functionalInterface{
   abstract public void abstractMethod();
 //   abstract public void abstractMethod1(); ошибка компиляции
}
```

Если интерфейс наследуется от функционального интерфейса и не содержит в себе новых абстрактных методов, тогда этот интерфейс также является функциональным.

Примеры функциональные интерфейсов

наследует функциональный интерфейс

```
interface Interface1 extends functionalInterface{
}
```

переопределяет метод функционального интерфейса

```
interface Interface2 extends functionalInterface{
    @Override
    abstract public void abstractMethod();
}
```

содержит в себе метод по умолчанию, который абстрактным не является

```
interface Interface3 extends functionalInterface{
    public default void defMethod(){};
}
```

Лямбда-выражения

Это анонимный метод (без имени), представляющий из себя объект, который можно присваивать переменной и передавать как аргумент в другие методы.

(параметры метода)->{тело метода}

```
(int a, int b) -> {  return a + b; }
```

Используются в основном для определения реализации функционального интерфейса

Стандартное использование интерфейса Runnable для создания потока

```
Runnable r = new Runnable() {   // анонимный класс, реализующий интерфейс
        public void run() {
            System.out.println("hello");
        }
    };
    Thread th = new Thread(r);
    th.start();
```

Использование Лямбда-выражения для функционального интерфейса

```
Runnable r = () -> System.out.println("hello");
    Thread th = new Thread(r);   th.start(); // или
new Thread(() -> System.out.println("hello")).start;
```

Параметризованные интерфейсы

Параметризованные типы (классы, интерфейсы и методы), где тип данных указан в виде параметра. Он конкретизируется при объявлении переменной или вызове метода.

Например, встроенный функциональный интерфейс Predicate<T> проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true. В качестве параметра лямбда-выражение принимает объект типа T

```
public interface Predicate<T> {
   boolean test(T t);
}

import java.util.function.Predicate;
public class LambdaApp {

   public static void main(String[] args) {
       Predicate<Integer> isPositive = x -> x > 0; // ссылка на интерфейс, реализующий test
       System.out.println(isPositive.test(5)); // true
      System.out.println(isPositive.test(-7)); // false
   }
}
```

Встроенные функциональные интерфейсы

Встроенные функциональные интерфейсы размещены в пакете java.util.function.

Функциональный интерфейс	Описание
Predicate<T>	проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true.
BinaryOperator<T>	принимает в качестве параметра два объекта типа T, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа T
UnaryOperator<T>	принимает в качестве параметра объект типа T, выполняет над ними операции и возвращает результат операции в виде объекта типа T
Function<T,R>	принимает в качестве параметра объект типа T, выполняет над ними операции и возвращает результат операций в виде объекта типа R
Consumer<T>	выполняет некоторое действие над объектом типа T, при этом ничего не возвращая
Supplier<T>	не принимает никаких аргументов, но должен возвращать объект типа T

Аннотации

Java-аннотация — представляют из себя дескрипторы, включаемые в текст программы, и используются для хранения метаданных программного кода, необходимых на разных этапах жизненного цикла программы., но формально не являются частью кода.

Аннотированы могут быть пакеты, классы, методы, переменные и параметры.

Указывается как **@ИмяАннотации**

Аннотация выполняет следующие функции:

- даёт необходимую информацию для компилятора;
- даёт информацию различным инструментам для генерации другого кода, конфигураций и т. д.;
- может использоваться во время выполнения программы для получения данных о классах, интерфейсах, полях и методах;

Аннотации, используемые компилятором

- **@Override** — проверяет, переопределён ли метод. Вызывает ошибку компиляции, если метод не найден в родительском классе (интерфейсе) или запрещено переопределение;
- **@Deprecated** — отмечает, что метод устарел и не рекомендуется к использованию. Вызывает предупреждение компиляции, если метод используется;
- **@SuppressWarnings** — указывает компилятору подавить предупреждения компиляции, определённые в параметрах аннотации ("unchecked"- игнорировать неявные преобразования), “cast” - игнорировать предупреждения приведения типов);
- **@SafeVarargs** — указывает, что код не осуществляет потенциально опасных операций, связанных с параметром переменного количества аргументов, иначе будет предупреждение. Применяется только к методам и конструкторам с переменным количеством аргументов;
- **@FunctionalInterface** — указывает, что это объявление типа будет функциональным интерфейсом Java 8.

Примеры использования встроенных аннотаций

```
class B extends A {
   @Override /* сообщает компилятору, что мы
    собираемся переопределить метод родительского
    класса. */
   void method() {
      System.out.println("этот метод переопределен");
   }
}
@Deprecated /* помечает метод как устаревший */
static void deprecatedMethod() { }

@Test (timeout = 1000) /* обозначает тестовые методы
    JUnit, задает время (мсек), по истечению которого тест
    считается провалившимся. */
public void test1(){ // код теста }
```

Собственные аннотации

- Аннотация задается описанием интерфейса с символом @ и содержит набор полей, которые описываются как методы (с круглыми скобками).
- Методы в пользовательской аннотации должны быть без параметров.
- Методы в пользовательской аннотации могут возвращать лишь примитивные типы, String, перечисления (Enums), аннотации или массивы из вышеперечисленных типов.
- Методы в пользовательской аннотации могут иметь значения по умолчанию.
- Аннотации могут иметь мета-аннотации, прикрепленные к ним. Мета аннотации используются для предоставления информации об аннотации.

Мета-аннотации
(аннотации, применяемые к другим аннотациям)

- @Retention — определяет, как отмеченная аннотация может храниться — в коде, в скомпилированном классе или во время работы кода;
- @Documented — указывает, что аннотации должны документироваться JavaDoc;
- @Target — указывает какие элементы можно помечать указанной аннотацией;
- @Inherited — аннотация будет наследоваться от базового класса (по умолчанию не наследуются);
- @Repeatable — указанная аннотация может быть применена несколько раз

Мета-аннотация Target

- @Target(ElementType.PACKAGE) – только для пакетов;
- @Target(ElementType.TYPE) – только для классов;
- @Target(ElementType.CONSTRUCTOR) – только для конструкторов;
- @Target(ElementType.METHOD) – только для методов;
- @Target(ElementType.FIELD) – только для атрибутов(*переменных*) класса;
- @Target(ElementType.PARAMATER) – только для параметров метода;
- @Target(ElementType.LOCAL_VARIABLE) – только для локальных переменных.

Пример собственных аннотаций

```
import java.lang.annotation.*;
 public class Main {
    @Target(ElementType.TYPE)      //означает, аннотация применяется к классу, а не к объекту
   @Retention(RetentionPolicy.RUNTIME) //означает, аннотация доступна при работе программы.

 public @interface version {                          //описание аннотации.
private float v(); // номер версии
private String author() default  “Аноним"; // автор, по умолчанию Аноним
}
public static void main(String[] args) {
        // добавляем  аннотацию к классу Cat
 @version(v=1.0f) // автор остаётся “Анонимом”
 // @ version(v=2.0f, author=“Иванов”)) объявление аннотации с указанием версии и автора
       class Cat {… }
        Cat a = new Cat();
        Class cl = a.getClass(); //получить ссылку на класс объекта
        // получаем ссылку на интерфейс  (аннотацию)
       version an = (version)cl.getAnnotation(version.class);
         System.out.println(an.autor);
   }
}
```