Иформация о типах

Механизм *RTTI* (*Runtime Type Information*) предназначен для получения и использования информации о типах во время выполнения программы. *RTTI* освобождает разработчика от необходимости выполнять всю работу с типами на стадии компиляции и открывает немало замечательных возможностей. Потребность в *RTTI* вскрывает целый ряд интересных (и зачастую сложных) аспектов объектно-ориентированного проектирования.

При помощи *RTTI* можно узнать точный тип объекта, на который указывает ссылка базового типа

Объект Class

Информация о типе хранится во время выполнения программы в специальном объект типа ***Class***, который и содержит описание класса. Каждый класс, задействованный в программе, представлен своим объектом ***Class***. Иначе говоря, при написании и последующей компиляции нового класса для него создается объект ***Class***

```
class Candy {static { System.out.print(“InitCandy"); }}
class Gum {static { System.out.print(«Init Gum"); }}
class Cookie {static { System.out.print(“Init Cookie"); }}
public class SweetShop {
  public static void main(String[] args) {
   System.out.print("в методе main()");
    new Candy();
    System.out.print("После создания объекта Candy");
    try {
      Class.forName("testRTTI.Gum");
    } catch(ClassNotFoundException e) {
    System.out.print("Класс Gum не найден");
    }}}
```

Объект Class

```
package testRTTI;
class A{
  static{System.out.println("Init A");}
  public void print(){System.out.println("Print A");}
}
class B extends A{
  static{System.out.println("Init B");}
  public void print(){System.out.println("Print B");}
}
public class TestRTTI {
  public static void main(String[] args)
    throws InstantiationException, IllegalAccessException,
                  ClassNotFoundException {
      Class cl =Class.forName("testRTTI.B");
      A gm = (A)cl.newInstance();
      gm.print();
  }
}
```

Объект Class

```
public class TestRTTI {
  static void printInfo(Class cc) {
    System.out.println("Имя класса: " + cc.getName() +
      " это интерфейс?[" + cc.isInterface() + "]");
    System.out.println("Простое имя: " +
            cc.getSimpleName());
    System.out.println("Каноническое имя: " +
            cc.getCanonicalName());
  }
  public static void main(String[] args) {
      printInfo(String.class);
     printInfo(String.class.getSuperclass());}}
```

В *Java* существует еще один способ получения ссылок на объект ***Class*** — посредством литерала ***class***. Интересно заметить, что создание ссылки на объект ***Class*** с использованием записи ***.class*** не приводит к автоматической инициализации объекта ***Class***.

```
Class cl = Class.forName("testRTTI.A");
Class cl = A.class;
```

Параметризация Class

```
public class GenericClassReferences {
  public static void main(String[] args) {
    Class intClass = int.class;
    Class<Integer> genericIntClass = int.class;
    genericIntClass = Integer.class; // То же самое
    intClass = double.class;
    // genericIntClass = double.class; // Недопустимо
  }}
```

Преобразование типов

```
class Building {}
class House extends Building {}

public class ClassCasts {
  public static void main(String[] args) {
    Building b = new House();
    Class<House> houseType = House.class;
    House h = houseType.cast(b);
    h = (House)b;
  }}
```

Проверка перед приведением типов

В языке *Java*, который при приведении проверяет соответствие типов, такое преобразование часто называют «безопасным нисходящим приведением типов».

Ключевое слово ***instanceof***, проверяет, является ли объект экземпляром заданного типа. Результат возвращается в логическом (***boolean***) формате

```
if(x instanceof Dog) ((Dog)x).bark()
```

Команда ***if*** сначала проверяет, принадлежит ли объект к классу ***Dog***, и только после этого выполняет приведение объекта к типу ***Dog***. Настоятельно рекомендуется использовать ключевое слово ***instanceof*** перед проведением нисходящего преобразования, особенно при недостатке информации о точном типе объекта; иначе возникает опасность исключения ***ClassCastException***.

Проверка перед приведением типов

```
class Base {}
class Derived extends Base {}

public class TestInstance {
public static void main(String[] arg){
  Base base = new Base();
  Derived derived = new Derived();
  println(derived instanceof Base);
  println(base instanceof Derived);
  println(derived.getClass().isInstance(base));
  println(base.getClass().isInstance(derived));
  println(derived.getClass().equals(base.getClass()));
  println(base.getClass().equals(derived.getClass()));
  println(derived.getClass().equals(Derived.class));
}}
```

В соответствии с концепцией типа ***instanceof*** дает ответ на вопрос: «Объект принадлежит этому классу или производному от него?» С другой стороны, сравнение объектов ***Class*** не затрагивает наследования — либо тип точно совпадает, либо нет.

Рефлексия

Отражение или **рефлексия** означает процесс, во время которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения.

Reflection API в Java используется для просмотра информации о классах, интерфейсах, методах, полях, конструкторах, аннотациях во время выполнения java программ.

При этом знать названия исследуемых элементов заранее не обязательно.

Все классы для работы с reflection расположены в пакете java.lang.reflect. Это метод (Method), конструктор (Constructor), массив (Array), поле (Field) и многие другие.

```
// Без рефлексии
new Foo().hello();
// С рефлексией
Class cls = Class.forName("Foo");
cls.getMethod("hello", null).invoke(cls.newInstance(),
null);
```

Рефлексия

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
public class TestReflection {
public static void main(String[] args) {
try {
    Class<?> c = Class.forName("java.util.ArrayList");
    Method[] methods = c.getMethods();
    Constructor[] ctors = c.getConstructors();
    for(Method method : methods)
       System.out.println(method.toString());
    for(Constructor ctor : ctors)
       System.out.println(ctor.toString());
}catch(ClassNotFoundException e)
        {System.out.println("No such class: " + e);}
}
}
```

Исследование модификаторов класса

```
Class c =  obj.getClass();
int mods = c.getModifiers();
if (Modifier.isPublic(mods)) {
    System.out.println("public");
}
if (Modifier.isAbstract(mods)) {
    System.out.println("abstract");
}
if (Modifier.isFinal(mods)) {
    System.out.println("final"); }
```

Чтобы узнать, какие модификаторы были применены к заданному классу, сначала нужно с помощью метода getClass получить объект типа Class, представляющий данный класс. Затем нужно вызвать метод getModifiers() для объекта типа Class, чтобы определить значение типа int, биты которого представляют модификаторы класса. После этого можно использовать статические методы класса java.lang.reflect.Modifier, чтобы определить, какие именно модификаторы были применены к классу.

Нахождение суперклассов

```
Class c = myObj.getClass();
Class superclass = c.getSuperclass();
```

Можно также использовать метод getSuperclass() для объекта Class, чтобы получить объект типа Class, представляющий суперкласс рефлексированного класса. Нужно не забывать учитывать, что в Java отсутствует множественное наследование и класс java.lang.Object является базовым классом для всех классов, вследствие чего если у класса нет родителя то метод getSuperclass вернет null. Для того чтобы получить все родительские суперклассы, нужно рекурсивно вызывать метод getSuperclass().

Определение интерфейсов, реализуемых классом

```
Class c =  LinkedList.class;
Class[] interfaces = c.getInterfaces();
for(Class cInterface : interfaces) {
    System.out.println( cInterface.getName() );
}
```

Работа с полями

```
Class c = obj.getClass();
Field[] publicFields = c.getFields();
for (Field field : publicFields) {
    Class fieldType = field.getType();
    System.out.println("Имя: " + field.getName());
    System.out.println("Тип: " + fieldType.getName());
}
```

Чтобы исследовать поля принадлежащие классу, можно воспользоваться методом getFields() для объекта типа Class. Метод getFields() возвращает массив объектов типа java.lang.reflect.Field, соответствующих всем открытым полям объекта. С помощью класса Field можно получить имя поля, тип и модификаторы.

В классе Class присутствуют пары методов, как например getFields и getDeclaredFields. Метод getFields возвращает только те поля, которые объявлены как public + public поля родительских классов, в то время как getDeclaredFields возвращает все поля текущего класса не зависимо от их видимости.

Работа с полями

Если известно имя поля, то можно получить о нем информацию с помощью метода getField()

```
Class c = obj.getClass();
Field nameField = c.getField("name");
```

Чтобы получить значение поля, нужно сначала получить для этого поля объект типа Field затем использовать метод get(). Метод принимает входным параметром ссылку на объект класса.

```
Class c = obj.getClass();
Field field = c.getField("name");
String nameValue = (String) field.get(obj)
```

Так же у класса Field имеются специализированные методы для получения значений примитивных типов: getInt(), getFloat(), getByte() и др.. Для установки значения поля, используется метод set().

```
Class c = obj.getClass();
Field field = c.getField("name");
field.set(obj, "New name");
```

Исследование конструкторов класса

```
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
for (Constructor constructor : constructors) {
   Class[] paramTypes = constructor.getParameterTypes();
   for (Class paramType : paramTypes) {
      System.out.print(paramType.getName() + " ");
   }
   System.out.println();
}
```

Чтобы получить информацию об открытых конструкторах класса, нужно вызвать метод getConstructors() для объекта Class. Этот метод возвращает массив объектов типа java.lang.reflect.Constructor. С помощью объекта Constructor можно затем получить имя конструктора, модификаторы, типы параметров и генерируемые исключения.

Исследование конструкторов класса

Можно также получить по отдельному открытому конструктору, если известны типы его параметров.

```
Class[] paramTypes =
        new Class[] { String.class, int.class};
Constructor aConstrct =
        c.getConstructor(paramTypes);
```

Методы getConstructor() и getConstructors() возвращают только открытые конструкторы. Если требуется получить все конструкторы класса, включая закрытые можно использовать методы getDeclaredConstructor() и getDeclaredConstructors() эти методы работают точно также, как их аналоги getConstructor() и getConstructors().

Исследование информации о методе

```
Class c = obj.getClass();
Method[] methods = c.getMethods();
for (Method method : methods) {
    System.out.println("Имя: " + method.getName());
    System.out.println("Возвращаемый тип: "
        + method.getReturnType().getName());

    Class[] paramTypes = method.getParameterTypes();
    System.out.print("Типы параметров: ");
    for (Class paramType : paramTypes) {
        System.out.print(" " + paramType.getName());
    }
    System.out.println(); }
```

Чтобы получить информацию об открытых методах класса, нужно вызвать метод getMethods() для объекта Class. Этот метод возвращает массив объектов типа java.lang.reflect.Method. Затем с помощью объекта Method можно получить имя метода, тип возвращаемого им значения, типы параметров, модификаторы и генерируемые исключения.

Исследование информации о методе

Также можно получить информацию по отдельному методу если известны имя метода и типы параметров.

```
Class c = obj.getClass();
Class[] paramTypes = new Class[] {int.class,String.class};
Method m = c.getMethod("methodA", paramTypes);
```

рассмотрим вызов метода зная его имя. Например метод getCalculateRating:

```
Class c = obj.getClass();
Class[] pTypes = new Class[] {String.class,int.class };
Method method=c.getMethod("getCalculateRating",pTypes);
Object[] args = new Object[]
    { new String("First Calculate"), new Integer(10) };
Double d = (Double) method.invoke(obj, args);
```

Метод invoke принимает два параметра, первый - это объект, класс которого объявляет или наследует данный метод, а второй - массив значений параметров, которые передаются вызываемому методу.

Применение рефлексии

```
import java.lang.reflect.Field;
 class WithPrivateFinalField {
    private int i = 1;
    private final String s = "String S";
    private String s2 = "String S2";
    public String toString() {
        return "i = " + i + ", " + s + ", " + s2; } }
public class ModifyngPrivateFields {
 public static void main(String[] args) throws Exception {
  WithPrivateFinalField pf = new WithPrivateFinalField();
  Field f = pf.getClass().getDeclaredField("i");
  f.setAccessible(true);  f.setInt(pf, 47);
  System.out.println(pf);
  f = pf.getClass().getDeclaredField("s");
  f.setAccessible(true);  f.set(pf, "MODIFY S");
  System.out.println(pf);
  f = pf.getClass().getDeclaredField("s2");
  f.setAccessible(true);  f.set(pf, "MODIFY S2");
  System.out.println(pf); } }
```

Применение рефлексии

```
class SomeAction {
  public void doSomething() {System.out.println("done");}}
class ActionHandler {
 public static void main(String[] args) {
 // у нас есть коллекция действий (action)
   List<Object> actions = new ArrayList<Object>() {
    { add(new SomeAction()); }};
  // а также путь запроса (request path)
  String path = "/SomeAction.doSomething";
  // получаем имя класса и метода
    String className =
       path.substring(path.lastIndexOf("/") +1,
                                    path.indexOf("."));
    String methodName =
       path.substring(path.indexOf(".") + 1);
    Object action = null;
    for (Object a : actions)
     if (a.getClass().getSimpleName().equals(className))
                  action = a;
```

Применение рефлексии

```
if (action == null){ /* это равносильно HTTP ERROR 404
*/;}
  // находим нужный метод
  Method method = null;
  try {
        method = action.getClass().getMethod(methodName);
  } catch (NoSuchMethodException e) { /* HTTP ERROR 404 */
}
  // вызываем метод
  try {
         method.invoke(action);
  } catch (IllegalAccessException e) {
// не возникнет если следовать контракту публичных методов
     throw new RuntimeException(e);
  } catch (InvocationTargetException e) {
     throw new RuntimeException(e.getCause());
        }
    }
}
```

Аннотации

Аннотации представляют из себя дескрипторы, включаемые в текст программы, и используются для хранения метаданных программного кода, необходимых на разных этапах жизненного цикла программы.

Информация, хранимая в аннотациях, может использоваться соответствующими обработчиками для создания необходимых вспомогательных файлов или для маркировки классов, полей и т.д.

В общем виде выглядит как @ + имя аннотации, написанное перед именем переменной, параметра, метода, класса, пакета, либо перед их модификаторами.

Аннотация выполняет следующие функции:

- дает необходимую информацию для компилятора;
- дает информацию различным инструментам для генерации другого кода, конфигураций и т. д.;
- может использоваться во время работы кода;

Аннотации

Аннотации, применяемые к java-коду:

@Override — проверяет, переопределен ли метод.

@Deprecated — отмечает, что метод устарел. Вызывает предупреждение компиляции, если метод используется;

@SuppressWarnings — указывает компилятору подавить предупреждения компиляции, определенные в параметрах аннотации;

Аннотации, применяемые к другим аннотациям:

@Retention — определяет, как отмеченная аннотация может храниться — в коде, в скомпилированном классе или во время работы кода;

@Documented — отмечает аннотацию для включения в документацию;

@Target — отмечает аннотацию как ограничивающую, какие элементы java-аннотации могут быть к ней применены;

@Inherited — отмечает, что аннотация может быть расширена подклассами аннотируемого класса;

Аннотации

```
import java.lang.annotation.*;
@Target(value=ElementType.FIELD)
@Retention(value= RetentionPolicy.RUNTIME)
public @interface Name {
    String name(); String type() default “string”;
}
```

RetentionPolicy.SOURCE - аннотация используется на этапе компиляции и должна отбрасываться компилятором;
RetentionPolicy.CLASS - аннтоация будет записана в class-файл компилятором, но не должна быть доступна во время выполнения (runtime);
RetentionPolicy.RUNTIME - аннотация будет записана в class-файл и доступна во время выполнения через reflection.

Аннотации

Аннотация @Inherited помечает аннотацию, которая будет унаследована потомком класса, отмеченного такой аннотацией. Сделаем для примера пару аннотаций и пометим ими класс.

```
@Inherited
@interface PublicAnnotate { }

@interface PrivateAnnotate { }

@PublicAnnotate @PrivateAnnotate
class ParentClass { }

class ChildClass extends ParentClass { }
```

Аннотации

```
package ncedu.annotations;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface SecretField{};
```

```
package ncstudying.testExceptions;
import ncedu.annotations.SecretField;
class DataOnly {
  int k;
  int p;
  @SecretField
  private String secretNameField="It's a secret";
 }
```

Аннотации

```
package testreflection;
import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import ncedu.annotations.SecretField;

Class cl =
      Class.forName("ncstudying.testExceptions.DataOnly");
   Object o = cl.newInstance();
   Field[] fl = cl.getDeclaredFields();
   for(Field f: fl){
        System.out.println(f.getName());
        Annotation a = f.getAnnotation(SecretField.class);
        f.setAccessible(true);
        if (a!=null){
             String st =(String)f.get(o);
             System.out.println(st);
        }

   }
```

Параметризация

Обычные классы и методы работают с конкретными типами: либо, примитивами, либо с классами. Если ваш код должен работать с разными типами, такая жесткость может создавать проблемы.

```
public class Holder2 {
  private Object a;
  public Holder2(Object a) { this.a = a; }
  public void set(Object a) { this.a = a; }
  public Object get() { return a; }
  public static void main(String[] args) {
    Holder2 h2 = new Holder2(new Automobile());
    Automobile a = (Automobile)h2.get();
    h2.set("Not an Automobile");
    String s = (String)h2.get();
    h2.set(1);
    Integer x = (Integer)h2.get();
  }
}
```

Параметризация

```
public class Holder3<T> {
  private T a;
  public Holder3(T a) { this.a = a; }
  public void set(T a) { this.a = a; }
  public T get() { return a; }
  public static void main(String[] args) {
    Holder3<Automobile> h3 =
      new Holder3<Automobile>(new Automobile());
    Automobile a = h3.get(); //Преобразование не требуется
    // h3.set("Not an Automobile");// Ошибка
    // h3.set(1); // Ошибка}}
```

```
public class TwoTuple<A,B> {
  public final A first; public final B second;
  public TwoTuple(A a, B b) { first = a; second = b; }
  public String toString() {
    return "(" + first + ", " + second + ")";}}

static TwoTuple<String,Integer> f() {
    return new TwoTuple<String,Integer>("hi", 47);
  }
```

```
public class LinkedStack<T> {
  private static class Node<U> {
    U item;
    Node<U> next;
    Node() { item = null; next = null; }
    Node(U item, Node<U> next) {
      this.item = item;
      this.next = next;
    }
    boolean end() { return item == null && next == null; }
  }
  private Node<T> top = new Node<T>();
  public void push(T item) {
    top = new Node<T>(item, top);
  }
  public T pop() {
    T result = top.item;
    if(!top.end())
      top = top.next;
    return result;
  }}
```

Параметризация

```
public static void main(String[] args) {
    LinkedStack<String> lss = new LinkedStack<String>();
    for(String s : "Phasers on stun!".split(" "))
      lss.push(s);
    String s;
    while((s = lss.pop()) != null)
      System.out.println(s);
  }
```

Параметризация работает и с интерфейсами. Например, класс, создающий объекты, называется генератором.

```
public interface Generator<T> {
   T next(); }
```

Параметризация методов

```
public class GenericMethods {
  public <T> void f(T x) {
    System.out.println(x.getClass().getName());
  }
  public static void main(String[] args) {
    GenericMethods gm = new GenericMethods();
    gm.f("");
    gm.f(1);
   }}
```

```
public class BasicGenerator<T> implements Generator<T> {
  private Class<T> type;
  public BasicGenerator(Class<T> type){ this.type = type;}
  public T next() {
    try {
          return type.newInstance();
    } catch(Exception e) {throw new RuntimeException(e);}}
  public static <T> Generator<T> create(Class<T> type) {
    return new BasicGenerator<T>(type);
  }}
```

Стирание

```
import java.util.*;

public class ErasedTypeEquivalence {
  public static void main(String[] args) {
    Class c1 = new ArrayList<String>().getClass();
    Class c2 = new ArrayList<Integer>().getClass();
    System.out.println(c1 == c2);
}
}
```

```
import java.util.*;

class Frob {}
class Fnorkle {}
class Quark<Q> {}
class Particle<POSITION,MOMENTUM> {}

public class LostInformation {
  public static void main(String[] args) {
    List<Frob> list = new ArrayList<Frob>();
    Map<Frob,Fnorkle> map = new HashMap<Frob,Fnorkle>();
    Quark<Fnorkle> quark = new Quark<Fnorkle>();
    Particle<Long,Double> p = new Particle<Long,Double>();
    System.out.println(Arrays.toString(
      list.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      map.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      quark.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      p.getClass().getTypeParameters()));
  }}
```

Стирание

Согласно документации *JDK*, ***Class.getTypeParameters()*** «возвращает массив объектов ***TypeVariable***, представляющих переменные типов, указанные в параметризованном объявлении...» Т.е. можно лишь узнать какие идентификаторы использовались в качестве заполнителей. Информация о параметрах типов недоступна внутри параметризованного кода. Таким образом, вы можете узнать идентификатор параметра типа и ограничение параметризованного типа, но фактические параметры типов, использованные для создания конкретного экземпляра, остаются неизвестными.

Параметризация в *Java* реализуется с применением стирания (*erasure*). Это означает, что при использовании параметризации вся конкретная информация о типе утрачивается. Внутри параметризованного кода вы знаете только то, что используется некий объект. Таким образом, ***List<String>*** и ***List<Integer>*** действительно являются одним типом во время выполнения; обе формы «стираются» до своего низкоуровневого типа ***List***.

Стирание

```
class GenericBase<T> {
  private T element;
  public void set(T arg) { arg = element; }
  public T get() { return element; }}
class Derived1<T> extends GenericBase<T> {}
class Derived2 extends GenericBase {}
public class ErasureAndInheritance {
  public static void main(String[] args) {
    GenericBase gb = new GenericBase();
    Derived2 d2 = new Derived2();
    Object obj = d2.get();
    d2.set(obj); }}
```

```
public class Erased<T> {
  private static final int SIZE = 100;
  public  void f(Object arg) {
   if(arg instanceof T) {}
   T var = new T();
   T[] array = new T[SIZE];
   T[] array = (T[])new Object[SIZE];
  }}
```

Стирание

```
class Building {}
class House extends Building {}

public class ClassTypeCapture<T> {
  Class<T> kind;
  public ClassTypeCapture(Class<T> kind) {
    this.kind = kind;
  }
  public boolean f(Object arg) {
    return kind.isInstance(arg);
  }
  public static void main(String[] args) {
    ClassTypeCapture<Building> ctt1 =
      new ClassTypeCapture<Building>(Building.class);
    System.out.println(ctt1.f(new Building()));
    System.out.println(ctt1.f(new House()));
    ClassTypeCapture<House> ctt2 =
      new ClassTypeCapture<House>(House.class);
    System.out.println(ctt2.f(new Building()));
    System.out.println(ctt2.f(new House()));}}
```

```
class ClassFactory<T> {
  T x;
  public ClassFactory(Class<T> kind) {
    try {
      x = kind.newInstance();
    } catch(Exception e) {
      throw new RuntimeException(e);
    }
}}
class Employee {}
public class InstantiateGenericType {
  public static void main(String[] args) {
    ClassFactory<Employee> fe =
      new ClassFactory<Employee>(Employee.class);
    print("ClassFactory<Employee> успех");
    try {
      ClassFactory<Integer> fi =
        new ClassFactory<Integer>(Integer.class);
    } catch(Exception e) {
      print("ClassFactory<Integer> неудача");
    }
  }}
```

Массивы параметризованных типов

```
public class Erased<T> {
  private static final int SIZE = 100;
  public T[] array;
  public  void f(Object arg) {
    array = (T[])new Object[SIZE];
   }
  public void put(int index,T value){
      array[index]=value;}
  public static void main(String[] arg){
      Erased<String> e = new Erased<String>();
      e.f(null);
      e.put(1,"asfsf");
      e.array[1]="asfsf";}}
```

```
import java.util.*;
 public class ListOfGenerics<T> {
  private List<T> array = new ArrayList<T>();
  public void add(T item) { array.add(item); }
  public T get(int index) { return array.get(index); }
}
```

Массивы параметризованных типов

```
class Generic<T> {}
class ArrayOfGenericReference {
  static Generic<Integer>[] gia;
//static Generic<Integer>[] gia=new Generic<Integer>[100];
static Generic<Integer>[] gia = new Generic[100];
}
```

Ограничения

У ограничений важный эффект: возможность вызова методов, определенных в ограничивающих типах. Поскольку стирание уничтожает информацию о типе, при отсутствии ограничений для параметров типов могут вызываться только методы ***Object***. Но, если ограничить параметр подмножеством типов, вы сможете вызвать методы из этого подмножества. Для установления ограничений в *Java* используется ключевое слово ***extends***.

```
interface HasIntData{ int getData();}
class DoSomething{
    void doSomething(int i){};
}

class SomeDoerWithInt<T extends DoSomething & HasIntData>{
    public void doSomething(T arg){
        arg.doSomething(arg.getData());
    }
}
```

Метасимволы

```
class A{}
class B extends A{}
class C extends B{}
class D extends A{};
public class TestRestrictions {
    public static void main(String[] arg){
        ArrayList<A> test = new ArrayList<B>();
    }    }
```

«Параметризованный тип, в котором задействован тип ***B***, нельзя присвоить параметризованному типу, в котором задействован тип ***A***». Компилятор отказывается выполнить «восходящее преобразование». Впрочем, это и не является восходящим преобразованием — ***List*** с элементами ***B*** не является «частным случаем» ***List*** с элементами ***A***.

Иногда между двумя разновидностями параметризованных типов все же требуется установить некоторую связь, аналогичную восходящему преобразованию. Именно это и позволяют сделать метасимволы.

Метасимволы

```
ArrayList<? extends A> test = new ArrayList<B>();
test.add(new A());
test.add(new B());
A a = test.get(0);
```

test относится к типу ***List<? extends A>***, что можно прочитать как «список с элементами любого типа, производного от ***A***». Однако в действительности это не означает, что ***test*** будет содержать именно типы из класс ***A***. Метасимвол обозначает «некоторый конкретный тип, не указанный в ссылке». Таким образом, присваиваемый ***test*** должен содержать некий конкретный тип (например, ***A*** или ***B***), но для восходящего преобразования этот тип несущественен.

Метасимволы

```
ArrayList<? super B> test = new ArrayList<A>();
test.add(new A());
test.add(new B());
A a = test.get(0);
```

Test является контейнером для некоторого типа, являющегося базовым для ***B***; из этого следует, что ***B*** и производные от ***B*** типы могут безопасно включаться в контейнер. Но, поскольку нижним ограничением является ***B***, мы не знаем, безопасно ли включать ***A*** в такой ***Test***, так как это откроет ***Test*** для добавления типов, отличных от ***B***, с нарушением статической безопасности типов.

Метасимволы

```
public class TreeMap<K,V>
{
private final Comparator<? super K> comparator;
public TreeMap(Comparator<? super K> comparator) {
        this.comparator = comparator;
    }
public void putAll(Map<? extends K, ? extends V> map) {….}
}

TreeMap<B,Integer> t = new TreeMap<B,Integer>(
                  new   Comparator<A>(){

            @Override
            public int compare(A o1, A o2) {

            }
        });
t.put(new C(), 1);
t.put(new B(), 2);
```

<?>

Литература

1. Брюс Эккель Философия Java. 4-е издание
2. Хорстманн К. С., Корнелл Г. -- Java 2. Том 1. Основы
3. Habrahabr.ru
4. Sql.ru
5. http://grepcode.com/project/repository.grepcode.com/java/root/jdk/openjdk/
6. !!! Google.com