Typy generyczne

Typy generyczne w języku Java są wizualnie podobne do szablonów C++. Pozawalają one na definiowanie parametryzowanych klas i funkcji. Podstawowe zalety:

- Mocna kontrola typów w trakcie kompilacji.
- Uniknięcie rzutowania
- Możliwość implementacji generycznych algorytmów.

Typy generyczne są najczęściej wykorzystywane przy implementacji kontenerów.

Przykład kontenera - stos

Klasa `Stack` przechowuje referencje typu `Object`. Potencjalnie, można więc zapisywać na stosie obiekty wszystkich typów.

```
public class Stack {
    static
    class StackFullException extends Exception{}
    static final int STACK_SIZE=1000;
    Object[] data = new Object[STACK_SIZE];
    int count =0;

    void push(Object o)throws StackFullException{
        if(count==STACK_SIZE)
            throw new StackFullException();
        data[count++]=o;
    }
    Object pop(){
        if(count==0)return null;
        return data[--count];
    }

    boolean isEmpty(){return count==0;}
}
```

Zapis

```
Stack s = new Stack();
s.push("koty");
s.push("dwa");
s.push("ma");
s.push("Ala");
```

Odczyt

```
while(!s.isEmpty()){
    System.out.println(s.pop());
}
```

lub

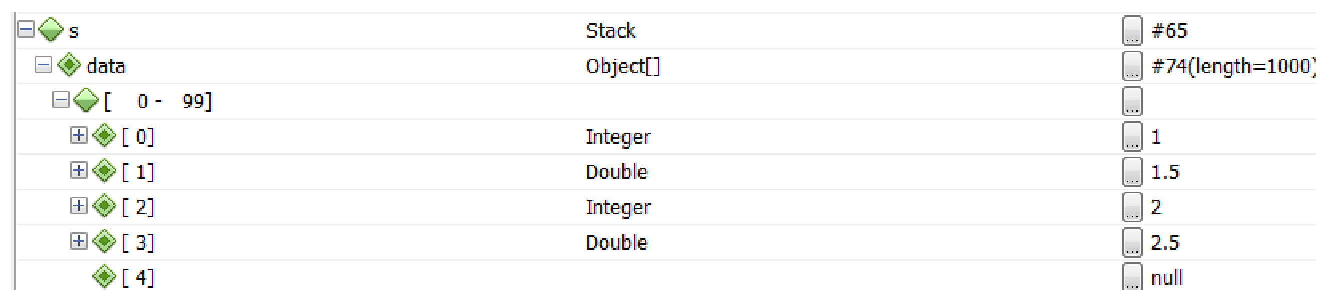
```
while(!s.isEmpty()){
    System.out.printf("%s ",s.pop());
}
```

Inny przykład

```
Stack s = new Stack();
s.push(1);
s.push(1.5);
s.push(2);
s.push(2.5);
while(!s.isEmpty()){
    System.out.printf("%f ",s.pop());
}
```

Wynik:

```
2,500000 Exception in thread "main"
java.util.IllegalFormatConversionException: f != java.lang.Integer
```



Kontenery obiektowe wymagają od programisty przeprowadzania jawnej kontroli typów.

```
while(!s.isEmpty()){
    Object v = s.pop();
    if(v instanceof Double)System.out.printf("%f ",v);
    if(v instanceof Integer)System.out.printf("%d ",v);
}
```

Najczęściej w kontenerze umieszczane są obiekty tego samego typu (np. `Double`). Dostęp do obiektów wymaga często przeprowadzenia rzutowania.

```
        Stack s = new Stack();
        for(double x=1;x<20;x=1.21*x){
            s.push(x);
        }
        double sum=0;
        while(!s.isEmpty()){
            sum+=(Double)s.pop();
        }
        System.out.println(sum);
```

Stos jako typ generyczny

```
public class Stack<T> {
    static
    class StackFullException extends Exception{}
    static final int STACK_SIZE=1000;
    Object[] data = new Object[STACK_SIZE];
    int count =0;

    void push(T t)throws StackFullException{
        if(count==STACK_SIZE)
            throw new StackFullException();
        data[count++]=t;
    }
    T pop(){
        if(count==0)return null;
        return (T)data[--count];
    }
    boolean isEmpty(){return count==0;}
}
```

- Kontener dalej przechowuje dane w postaci tablicy referencji Object.
- Nie jest możliwe użycie wewnątrz kontenera tablicy:
  `T[] data = new T[STACK_SIZE];`
  ponieważ typ generyczny nie jest kompilowany w momencie instancjacji, ale wcześniej (w odróżnieniu od C++). Wobec tego typ T jest nieznany w trakcie kompilacji i nie może powstać tablica **`T[]`**.
- Z drugiej strony – w definicji można użyć referencji do typu będącego parametrem `T`, czy nawet `T[]`.
- Obiekt typu generycznego przechowuje informacje, jaki typ (typy) został użyty jako parametr. Dzięki temu możliwe jest sprawdzanie zgodności typów obiektów użytych w wywołaniach.
- Nie jest konieczne rzutowanie

```
Stack<String> s = new Stack<String>();
s.push("koty");
s.push("dwa");
s.push("ma");
s.push("Ala");
s.push(new Double(1.2)); //> BŁĄD
```

```
incompatible types: Double cannot be converted to String
----
(Alt-Enter shows hints)
s.push(new Double(1.2)); //> BŁĄD
```

Uwaga: począwszy od wersji Java 7 możliwe jest uproszczenie zapisu (ang. diamond operator): `Stack<String> s = new Stack<>();`

Wbudowana kontrola typów chroni przed przypadkowym dodaniem nieodpowiednich wartości (i zarazem zwalnia z rzutowania).

```
Stack<Double> s = new Stack<Double>();
s.push(1.); //nie s.push(1);
s.push(1.5);
s.push(2.); //nie s.push(2);
s.push(2.5);
double sum=0;
while(!s.isEmpty()){
    Double v=s.pop();
    System.out.printf("%f ",v);
    sum+=v;
}
```

Do kontenera możemy dodawać obiekty klas potomnych typu, który był parametrem instancjacji, ale kontener ma jedynie informacje o typie bazowym.

```
class A{}
class B extends A{}
```

```
        Stack<A> s = new Stack<A>();
        s.push(new A());
        s.push(new B());
        s.push(new A());
        s.push(new B());
        while(!s.isEmpty()){
            A v=s.pop();
        }
```

Bardzo prosty kontener generyczny

```
public class Cage<T> {
    T object;
    //T object=new T();
    Cage(){}
    Cage(T t){
        object=t;
    }
    void put(T t){
        object=t;
    }
    T get(){
        return object;
    }
}
```

- `T` – parameter typu

`Cage<String> cage;`

- `String` – argument typu generycznego
- `Cage<String>` – wywołanie (ang. invocation) typu generycznego

```
Cage<String> stringCage = new Cage<String>();
```

albo

```
Cage<String> stringCage = new Cage<>();
```

W odróżnieniu od C++ wewnątrz typu generycznego nie jest możliwe utworzenie obiektu nieznanego typu. Możliwe jest natomiast przekazanie referencji z zewnątrz, np.: w konstruktorze

```
public class Cage<T> {
    T object;
    //T object=new T();
    Cage(T t){
        object=t;
    }
...
}
```

Konwencje dotycząca nazw typów

- `T` – dowolny typ
- `N` – typ liczbowy
- `E` - element kontenera
- `K` - klucz
- `V` – wartość, np. `Map<K,V>`
- `S`,`U`,`V`- dalsze parametry typu generycznego

Podobnie, jak w C++ podczas deklaracji typu parametryzowanego możliwe jest zastąpienie jednego z parametrów typem parametryzowanym:

```
Cage<Cage<Integer>> numberInCages= new Cage<>();
```

Typowe zastosowanie (brak bibliotece kontenerów Java jest kontenera typu multimap):

```
HashMap<String,HashSet<String>> map = new HashMap<>();
                 //new HashMap<String,HashSet<String>>();
```

Surowe typy (ang. raw types)

Surowy typ, to wykorzystywany wewnętrznie typ danych odpowiadający typowi generycznemu pozbawionemu parametrów.

```
Cage rawCage = new Cage();
```

Obiekt `rawCage` nie ma żadnych informacji o typie składowanego obiektu. Stąd nie jest możliwa automatyczna kontrola typów przesyłanych obiektów. Przy odczycie konieczne jest rzutowanie.

①	`Cage rawCage = new Cage();`
②	`rawCage.put(new Double(3.14));`
③	`//rawCage.put("Ala ma kota");`
④	`Double d = (Double)rawCage.get();`

Jeżeli usuniemy komentarz w wierszu ③, w kolejnym podczas rzutowania zostanie wygenerowany wyjątek.

Wzajemne konwersje

Surowe typy i typy parametryzowane można wzajemnie podstawiać

- Poprawne: *raw* ← *parametrized*
- Ostrzeżenie: *parametrized* ← *raw*

Może to jednak prowadzić do braku kontroli nad poprawnością użytych typów:

```
        Cage<String> stringCage = new Cage<>();
        Cage rawCage = stringCage;
        rawCage.put(3.14);
        Double d = (Double)rawCage.get();
        System.out.println(d);
```

```
        Cage<String> stringCage;
        Cage rawCage = new Cage();
        stringCage =  rawCage;
        rawCage.put(3.14);
        String s = stringCage.get();// wyjątek
```

Ostatni przykład wygeneruje wyjątek:

```
java.lang.ClassCastException: java.lang.Double cannot be
cast to java.lang.String
```

Metody generyczne

Możliwe jest zdefiniowanie i wywołanie generycznych metod, zarówno statycznych, jak i niestatycznych.

```
public class GenericMethod {
    public <T> void dump(T[] table){
        for(T t:table)System.out.println(t);
    }
```

```
    static public <T> String join(T[] table){
        StringBuilder b = new StringBuilder();
        for(int i=0;i<table.length;i++){
            if(i!=0)b.append(";");
            b.append(table[i].toString());
        }
        return b.toString();
    }
}
```

Przykład wywołania:

```
        new GenericMethod().<String>dump(
          new String[]{"Raz","dwa","trzy"}
        );
        String result = GenericMethod.<Double>join(
          new Double[]{1.2,2.34,4.5}
        );
        System.out.println(result);
```

Wynik:

```
Raz
dwa
trzy
1.2;2.34;4.5
```

Nie jest konieczne podawanie argumentów instancjacji: **`<String>`** oraz **`<Double>`**. Wystarczy podanie samej nazwy metody. Kompilator ustali użyty argument typu generycznego na podstawie argumentów wywołania metody:

```
GenericMethod.join(new Double[]{1.2,2.34,4.5})
```

Ograniczone paramtery typu

Niejednokrotnie podczas implementacji typów lub metod generycznych zakłada się, że parametry realizują pewien interfejs funkcjonalny. Równocześnie może on zostać zaimplementowany w różny sposób (polimorficznie) w pewnej hierarchii klas.

Przykład

Klasa `Number` jest nadklasą kilku klas reprezentujących wartości numeryczne: `AtomicInteger`, `AtomicLong`, `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short`.

Implementuje ona kilka metod konwersji: `doubleValue()`, `floatValue()`, `longValue()`, itp.

Zaimplementowana metoda generyczna oblicza sumę elementów tablicy, należących do klasy `Number` lub jej potomnych.

```
static public <T extends Number> double sum(T[] table){
    double sum=0;
    for(T t:table){
        System.out.println(t.getClass());
        sum+=t.doubleValue();
    }
    return sum;
}
```

Dla wywołania:

```
double sum = GenericMethod.sum(new Float[]{1f,2f,3f});
System.out.println(sum);
```

obliczona zostanie suma elementów tablicy typu `Float[]`

```
class java.lang.Float
class java.lang.Float
class java.lang.Float
6.0
```

Możliwe jest również dostarczenie tablicy zawierającej elementy różnych typów: potomnych `Number`

```
double sum = GenericMethod.sum(new Number[]{1.1,2.1f,3});
System.out.println(sum);
```

Wyjście:

```
class java.lang.Double
class java.lang.Float
class java.lang.Integer
6.199999904632568
```

Wielokrotne ograniczenia

W przypadku, kiedy wymagane jest, aby parametr implementował zbiór kilku interfejsów, można podać je oddzielając znakiem &:

```
class D <T extends A & B & C> { /* ... */ }
```

Jeśli jeden z elementów jest identyfikatorem klasy, musi pojawić się na pierwszym miejscu.

Przykład:

```
interface MovableForward{ void moveForward(int dist);}
interface Turnable{void turn(double angle); }
interface Traceable{void penUp(); void penDown();}
. . .
    static <T extends MovableForward & Turnable & Traceable>
    void dashedArc(T turtle){
        for(int i=0;i<100;i++){
        turtle.penUp();
        turtle.moveForward(10);
        turtle.turn(10);
        turtle.penDown();
        turtle.moveForward(10);
        turtle.turn(10);
        }
    }
```

Algorytm sortowania

Typowym zastosowaniem ograniczeń jest implementacja generycznych algorytmów. Przykładowo, algorytm sortowania tablicy wymaga, aby możliwe było porównanie elementów. Tym samym muszą one realizować interfejs `Comparable`:

```
    public interface Comparable<T>{
        int  compareTo(T o);
    }
```

```
static<T extends Comparable<T>>void bubbleSort(T[] table){
    boolean swap;
    do{
        swap=false;
        for(int i=0;i<table.length-1;i++){
            if(table[i].compareTo(table[i+1])>0){
                T tmp=table[i];
                table[i]=table[i+1];
                table[i+1]=tmp;
                swap=true;
            }
        }
    }while(swap);
}
```

Wywołanie

```
        Integer[]table={12,-41,4,8,-5,9,3,6,-11,2,8};
        bubbleSort(table);
        for(Integer i:table)System.out.print(i+" ");
```

Wynik: `-41 -11 -5 2 3 4 6 8 8 9 12`

Typy generyczne i dziedziczenie

Pomiędzy typami generycznymi może zachodzić relacja dziedziczenia.

Przykład:

```
public class SpecialCage<T> extends Cage<T>{
    boolean locked=false;
    void lock(){locked=true;}
    void unlock(){locked=false;}
    void put(T t){
        if(!locked)object=t;
    }
    T get(){
        if(!locked)return object;
        else return null;
    }
}
```

Zdefiniujmy trzy klasy:

```
class Animal{}
class Lion extends Animal{}
class Giraffe extends Animal{}
```

W takim przypadku obiekt typu **`SpecialCage<Animal>`** może przypisany do referencji **`Cage<Animal>`**, czyli zachodzi pomiędzy nimi relacja dziedziczenia.

```
Cage<Animal> lockCage = new SpecialCage<Animal>();
```

Niestety, w trakcie wykonania skompilowanego kodu nie jest możliwe sprawdzenie, czy instancja należy do typu parametryzowanego. Poniższy kod nie skompiluje się:

```
if(lockCage instanceof Cage<Animal>){. . .}
```

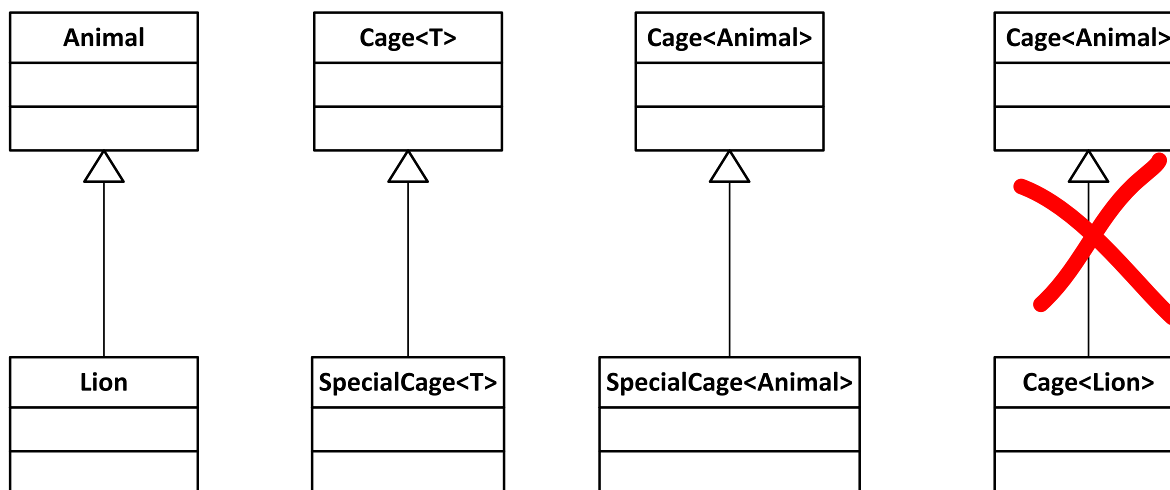
W kontenerze `Cage<Animal>` i kontenerze potomnym `SpecialCage<Animal>` mogą być umieszczane zarówno obiekty klasy `Animal`, jak i `Lion`.

```
Cage<Animal> lockCage = new SpecialCage<Animal>();
lockCage.put(new Animal());
Animal a = lockCage.get();
System.out.println(a.getClass());
lockCage.put(new Lion());
a = lockCage.get();
System.out.println(a.getClass());
```

Wynik:

```
class xxx.Animal
class xxx.Lion
```

Podsumowanie wariantów relacji dziedziczenia



- Specjalna klatka na zwierzęta jest rodzajem klatki na zwierzęta
- Co jest nieco zaskakujące – klatka na lwy nie jest rodzajem klatki na zwierzęta, mimo że lew jest zwierzęciem. Typy `Cage<Animal>` oraz `Cage<Lion>` są niepowiązane. Jedynym ich powiązaniem jest wspólna klasa bazowa: `Object`.

Dlaczego - przykład

```
①        Cage<Lion> lions = new Cage<>();
②        Cage<Animal> animals = lions;
③        animals.put(new Giraffe());
④        Lion leo = lions.get();
```

W `Cage<Animal>` możemy umieszczać dowolne zwierzęta, w tym żyrafę ③. Wbudowana kontrola typów pozwala mieć jednak pewność, że w klatce `Cage<Lion>` są same lwy, wobec tego można je bezpiecznie odczytać w ④, nawet nie przeprowadzając rzutowania. Niestety, w klatce zamiast lwa znajduje się żyrafa, co byłoby przyczyną wygenerowania wyjątku spowodowanego przez wywołanie niejawnego operatora rzutowania.

Niespójność, wywołana(by) została przez instrukcję ②. Ze względu na taki możliwy scenariusz przypisanie `animals = lions` jest zabronione.

Kilka przykładów

- `HashSet` dziedziczy po `Set`

```
Set<String> set = new HashSet<String>();
```

- `HashMap` dziedziczy po `Map`

```
Map<Integer,Integer> map = new HashMap<Integer,Integer>();
```

- Niezgodność typów:

```
Map<Integer,Set<Integer>> mmap
         = new HashMap<Integer,HashSet<Intege>>();
```

- Korekta niezgodności typów:

```
Map<Integer,Set<Integer>> mmap
          = new HashMap<Integer,Set<Integer>>();
mmap.put(1, new HashSet<>(Arrays.asList(2,3,4,5)));
```

Wnioskowanie o typach

Wnioskowanie o typach (ang. *type inference*) to zdolność kompilatora do ustalenia jaki typ miały argumenty wywołania (ang. *invocation*) typu generycznego na podstawie kodu, który się do niego odwołuje, w tym:

- argumentów metod
- typów zwracanych wartości.

Co więcej, dobierany będzie typ najbardziej specyficzny, który można dopasować do wszystkich argumentów.

Wnioskowanie o typach argumentów metod generycznych

Zazwyczaj podczas wywołania metod generycznych typy parametrów metody generycznej ustalane są automatycznie. Stąd nie jest konieczne podawanie użytego typu w ostrych nawiasach.

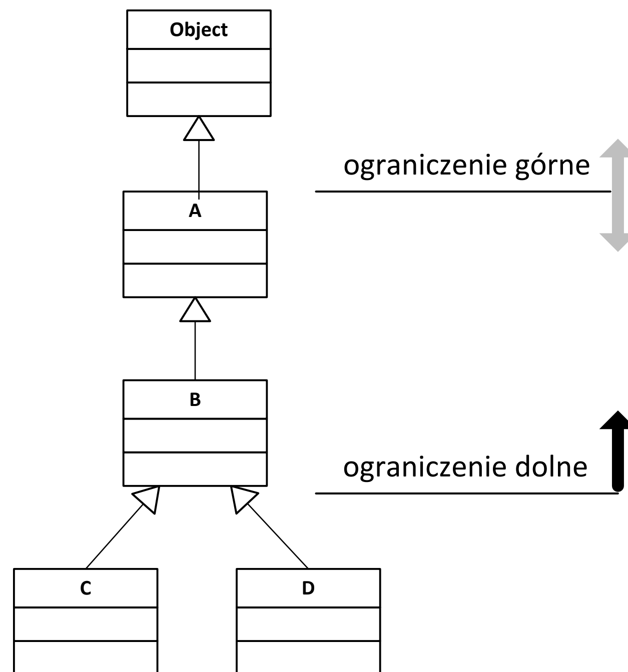
Przykład

```
class A{}
class B extends A{}
class C extends B{}
class D extends B{}


public class Intersection {

    static <T> T typeIntersection(T a,T b){
        return Math.random()<0.5?a:b;
    }
}
```

Podczas wnioskowania o argumentach wywołania metody `typeIntersection()` muszą być brane pod uwagę trzy elementy:

- Argumenty `a` i `b`
- Kontekst, w którym następuje wywołanie, np. typ referencji c w przypisaniu
  `c = typeIntersection(new C(),new D()).`

Występują tu dwa ograniczenia: dolne (na podstawie parametrów wywołania) oraz górne, wynikające z kontekstu.



`Intersection.typeIntersection(new C(),new D());`
`T` jest typu `B`

`A a = Intersection.typeIntersection(new C(),new D());`
`T` jest typu `B` (obiekt typu `B` może być przypisany do A)

`Intersection.<A>typeIntersection(new C(),new D());`
`T` jest typu `A`. Jest to jawne wskazanie typu argumentu dla inwokacji metody generycznej

```
C c = Intersection.typeIntersection(new C(),new D());
```

Błąd: ograniczenie górne (klasa C) leży poniżej dolnego (klasa B)

```
incompatible types: inferred type does not conform to upper bound(s)
    inferred: B
    upper bound(s): C,Object

May split declaration into a declaration and assignment
----
(Alt-Enter shows hints)
        C c = Intersection.typeIntersection(new C(),new D());
```

```
B b = (B)Intersection.<A>typeIntersection(new C(),new D());
```

`T` typu `A`, dlatego zwracaną wartość musimy jawnie rzutować w dół hierarchii na `B`.

Wnioskowanie podczas tworzenia instancji typów generycznych

Kompilator natrafiając na instrukcję, w której następuje wywołanie konstruktora typu generycznego może przeprowadzić wnioskowanie o typach. Począwszy od Java 7 zamiast pisać:

```
    Map<Integer,Set<Integer>> mmap
    = new HashMap< Integer,Set<Integer>>();
```

można użyć operatora <> (ang. diamond operator).

```
Map<Integer,Set<Integer>> mmap = new HashMap<>();
```

Należy zwrócić uwagę, że pominięcie operatora <> wyłączy kontrolę typów – użyta zostanie surowa wersja typu generyczngo. Na ogół nie będzie to źródłem problemów, jeśli konsekwentnie w kodzie używane będą referencje typu parametryzowanego.

Przykład utraty kontroli

```
①      Map<Integer,String> mis = new HashMap();
②      Map rawHM = mis;
③      Map<Integer,Set<Integer>> misi = rawHM;
④      mis.put(1,"cat");
⑤      Set<Integer> set = misi.get(1);
```

Instrukcja ① spowoduje ostrzeżenie podczas kompilacji, ale naprawdę źródłem błędów będą ② i ③.

Target inference (wnioskowanie o typach docelowych)

W Java 8 możliwe jest wnioskowanie o typach w przypadku zagnieżdżonych wywołań typów generycznych. Typ docelowy wyrażenia, to oczekiwany typ ustalany na podstawie kontekstu, w którym wyrażenie występuje.

Przykład

```
public class CageFactory<T> {
    static <T> Cage<T> makeCage(){
        Cage<T> cage = new Cage<>();
        return cage;
    }
}
```

Dla instrukcji

```
Cage<Lion> cage = CageFactory.makeCage();
```

typem wyrażenia `CageFactory.makeCage()` jest `Cage<Lion>`, stąd kompilator wnioskując o typach ustalił, że wywołanie ma postać:

```
Cage<Lion> cage = CageFactory.<Lion>makeCage();
```

Dla instrukcji

```
CageFactory.makeCage().put(new Lion());
```

typem wyrażenia `CageFactory.makeCage()` musi być również `Cage<Lion>`, ponieważ dla wynikowego obiektu `Cage<T>` wykonywana jest instrukcja `put(new Lion())`.

Wywołanie jest więc równoważne:

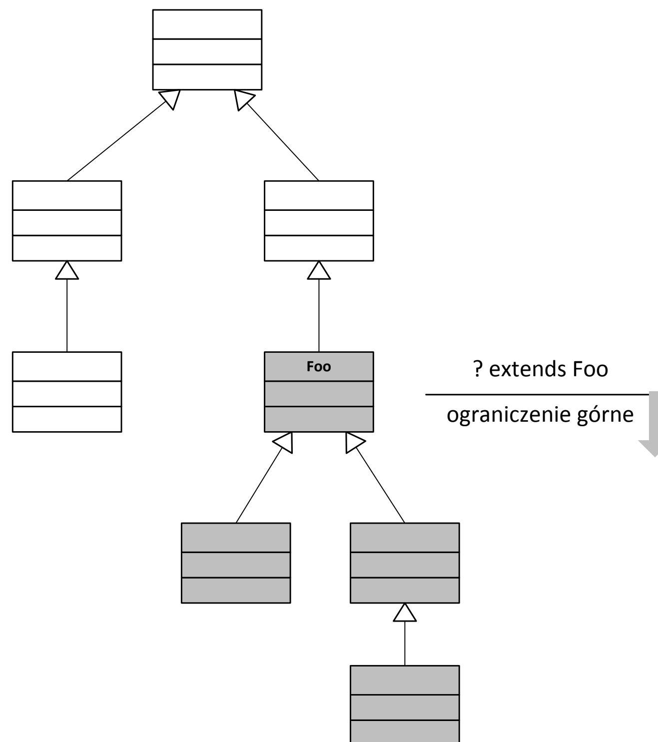
```
CageFactory.<Lion>makeCage().put(new Lion());
```

Symbole wieloznaczne

Definiując typy generyczne możliwe jest użycie wieloznacznego symbolu typu w postaci `?` (ang. *wildcard*). Najczęściej typy wieloznaczne określają miejsce typu w hierarchii klas (lub interfejsów). Są to więc typy ograniczone od góry lub od dołu.

Ograniczenie górne

Ograniczenie górne postaci `<? extends Foo>` specyfikuje zakres typów obejmujący `Foo` i klasy potomne.




```
static double scalarProduct(
    List<? extends Number> a,
    List<? extends Number> b){
    double r=0;
    for(int i=0;i<a.size() & i<b.size();i++){
        r+=a.get(i).doubleValue()*b.get(i).doubleValue();
    }
    return r;
}
```

```
public static void test(){
    List<Integer> li = Arrays.asList(1,2,3);
    List<Double> ld = Arrays.asList(1.1,2.2,3.3,4.);
    double p = scalarProduct(li,ld);
    System.out.println(p);
}
```

Wynik: `15.399999999999999`

Inny przykład

Budujemy poprawioną wersję klatki pozwalającą na bezpieczną przeprowadzkę zwierzęcia.

```
public class BetterCage<T> extends Cage<T>{
    void moveFrom(Cage<T> other){
        object =other.object;
        other.object=null;
    }
}
```

Następnie próbujemy do niej przeprowadzić lwa umieszczonego w zwykłej klatce `Cage<Lion> cl`.

```
① BetterCage<Animal> ca = new BetterCage<>();
② Cage<Lion> cl = new Cage<>();
③ cl.put(new Lion());
④ ca.moveFrom(cl);
```

Niestety, instrukcja ④ podczas kompilacji pokazuje błąd, chociaż oczywiście przenosiny byłyby możliwe: `Lion` dziedziczy po `Animal`.

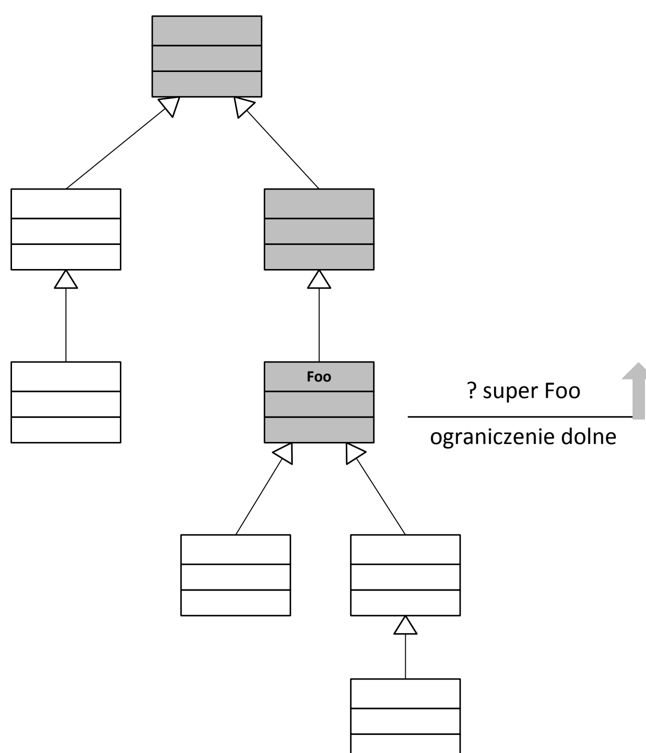
```
error: incompatible types: Cage<Lion> cannot be converted to
Cage<Animal>
        ca.moveFrom(cl);
```

Rozwiązaniem jest zadeklarowanie, że metoda `moveFrom()` przyjmuje argumenty typu `Cage` parametryzowanego dowolną klasą dziedziczącą po `T`, czyli:

```
void moveFrom(Cage<? extends T> other){
. . .
}
```

Ograniczenie dolne

Ograniczenie dolne ma postać `<? super Foo>` i specyfikuje zakres typów obejmujący klasę `Foo` i jej nadklasy. Specyfikacja ograniczenia dolnego ma wykorzystanie w przypadku metod, gdzie referencja jest konwertowana do jednej z nadklas. Typ `Foo` może zostać automatycznie zrzutowany w górę hierarchii.



Przykład

Funkcja `append()` dodaje zawartość listy `src` do `target`. Jednak jej działanie jest ograniczone do list tego samego typu.

```
static <T> void append(List<T> target, List<T> src){
    for(T t:src)target.add(t);
}
```

Dla kodu postaci:

```
① List<Number> ln = Arrays.asList(1,2.1,3.2f);
② List<Double> ld = Arrays.asList(1.1,2.2,3.3,4.);
③ append(ln,ld);
```

w instrukcji ③ zostanie wskazany błąd kompilacji, mimo, że elementy typu `Double` można dodać do listy `ln` (już nawet tam są). Rozwiązaniem jest zadeklarowanie, parametru `target`, jako: `List<? super T> target`. W takim przypadku funkcja przyjmie postać:

```
static <T>
void append(List<? super T> target, List<T> src){
    for(T t:src)target.add(t);
}
```

Idąc dalej, można kopiować elementy typów poniżej ograniczenia górnego do typów powyżej ograniczenia dolnego, stąd najbardziej elastyczną będzie deklaracja:

```
static <T>
void append(List<? super T> target, List<? extends T> src){
    for(T t:src)target.add(t);
}
```

Przykład BetterCage

Analogicznie możemy rozszerzyć przykład `BetterCage`, dodając funkcję przeprowadzki do innej klatki.

```
public class BetterCage<T> extends Cage<T>{
    void moveFrom(Cage<? extends T> other){
        object =other.object;
        other.object=null;
    }
    void moveTo(Cage<? super T> other){
        other.object=object;
        object=null;
    }
}
```

Przykładowy kod przeprowadzki ma postać:

```
        BetterCage<Lion> cl = new BetterCage<>();
        cl.put(new Lion());
        Cage<Animal> ca = new Cage<>();
        cl.moveTo(ca);
```

Nieograniczone symbole wieloznaczne

Nieograniczone symbole wieloznaczne `<?>` mogą być użyte w przypadku, kiedy implementowana funkcjonalność nie zależy od typu.

Przykład 1

```
    public static String toCSV(List<?> list){
        StringBuilder buf = new StringBuilder();
        for(Object o:list){
            buf.append(o);
            buf.append(";");
        }
        return buf.toString();
    }
```

Wywołanie:

```
        List<Number> ln = Arrays.asList(1,2.1,3.2f);
        System.out.println(toCSV(ln));
```

Wynik: `1;2.1;3.2;`

Podobny przykład

```
   public static void print(List<?> list){
        System.out.print("[ ");
        for(Object o:list){
            System.out.print(o+" ");
        }
        System.out.println("]");
    }
```

Przechwytywanie typów i metody pomocnicze

W pewnych sytuacjach kompilator stara się ustalić typ symbolu wieloznacznego na podstawie analizowanego wyrażenia. Ten proces nazywany jest przechwytywaniem typów (ang. *type capture*).

W szczególności dotyczy to sytuacji, kiedy sprawdzana jest zgodność typu referencji umieszczanej w kontenerze.

Przykład

```
public static void shuffle(List<?> list){
    Random r = new Random(System.currentTimeMillis());
    for(int i=0;i<list.size();i++){
        int i1 = r.nextInt(list.size());
        int i2 = r.nextInt(list.size());
        Object tmp = list.get(i1);
        list.set(i1, list.get(i2));
        list.set(i2,tmp);
    }
}
```

Zadaniem funkcji `shuffle(List<?> list)` jest losowa zmiana kolejności elementów na liście. Jest ona niezależna od typu, przestawia pary wylosowanych elementów. Jednakże podana implementacja wykazuje błędy.

Kompilator nie chce pozwolić, na wywołanie metody `set()` umieszczającej w tym przypadku element typu `Object` na liście nieokreślonego typu – nazwanego podczas kompilacji `CAP#1`.

Obejściem tego problemu jest zaimplementowanie metody pomocniczej (ang. *helper method*) działającej na liście dowolnego typu `T` i wywołanie jej w miejscu wskazanych instrukcji. Podczas sprawdzania parametr T zostanie zastąpiony przez `CAP#1` i kompilator stwierdzi zgodność typu obiektu dodawanego w wywołaniu `set()` z typem listy: `List<CAP#1>`.

```
public static void shuffle(List<?> list){
    Random r = new Random(System.currentTimeMillis());
    for(int i=0;i<list.size();i++){
        int i1 = r.nextInt(list.size());
        int i2 = r.nextInt(list.size());
        swap(list,i1,i2);
    }
}
static <T>void swap(List<T> list, int i1, int i2){
        T tmp = list.get(i1);
        list.set(i1, list.get(i2));
        list.set(i2,tmp);
}
```

Przykład wywołania

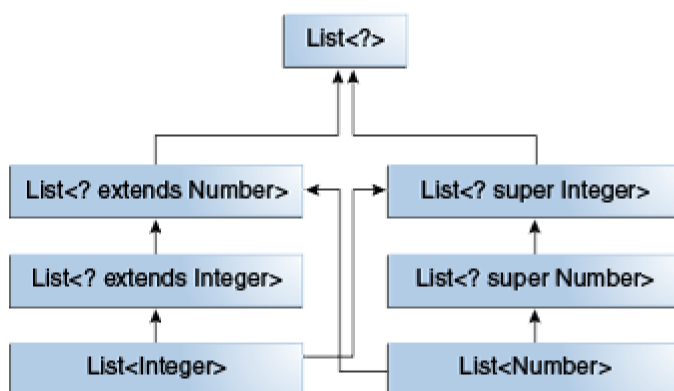
```
List<Integer> ln =
Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12);
shuffle(ln);
print(ln);
```

Wynik

```
[ 1 7 9 5 12 6 3 2 4 11 10 8 ]
```

Symbole wieloznaczne i dziedziczenie

Typy parametryzowane symbolami wieloznacznymi można traktować jako powiązane relacją dziedziczenia.



Rysunek i przykład na podstawie http://docs.oracle.com/javase/tutorial/java/generics/subtyping.html

Nie jest to jednak „prawdziwa" relacja dziedziczenia, której istnienie można np. sprawdzić za pomocą operatora instanceof. Jest to relacja, której obecność może kompilator *wywnioskować* analizując na przykład przepływ argumentów pomiędzy wywołaniami funkcji i możliwe ich konwersje.

Przykłady poprawnych konwersji argumentów.

Wywołania	Relacja dziedziczenie
void f1(**List<?>** arg) {}	Brak
void f2(**List<? extends Number>** arg) { f1(arg); }	List<? extends Number> → List<?>
void f3(**List<? extends Integer>** arg) { f2(arg);f1(arg); }	List<? extends Integer> → List<? extends Number> List<?>
void f4(**List<Integer>** arg) { f3(arg);f5(arg); **f2(arg);f1(arg);** }	List<Integer> → List<? extends Integer> List<? super Integer> List<? extends Number> List<?>

void f5(**List<? super Integer>** arg) { f1(arg); }	List<? super Integer> → List<?>
void f6(**List<? super Number>** arg) { f5(arg); f1(arg); }	List<? super Number> → List<? super Integer> List<?>
void f7(**List<Number> arg**) {f2(arg);f6(arg); **f5(arg); f1(arg);** }	List<Number> arg) → List<? extends Number> List<? super Number> List<? super Integer> List<?>

Wbrew przypuszczeniom (dla maszyny wnioskującej o typach wbudowanej w kompilator Java 8) – typy List<? super Integer> oraz List<? extends Integer> są niepowiązane.

Kompilator usiłuje przechwycić typ List<? extends Integer>, ale nie potrafi go dopasować do List<? super Integer>.

```
90
91  void f5(List<? super Integer> arg){f1(arg);     }
92
93  incompatible types: List<CAP#1> cannot be converted to List<? super Integer>
94    where CAP#1 is a fresh type-variable:
95      CAP#1 extends Integer from capture of ? extends Integer
96  ----
97  (Alt-Enter shows hints)
98
    void nomatch(List<? extends Integer> arg){ f5(arg); }
```

Zmienne wejściowe i wyjściowe

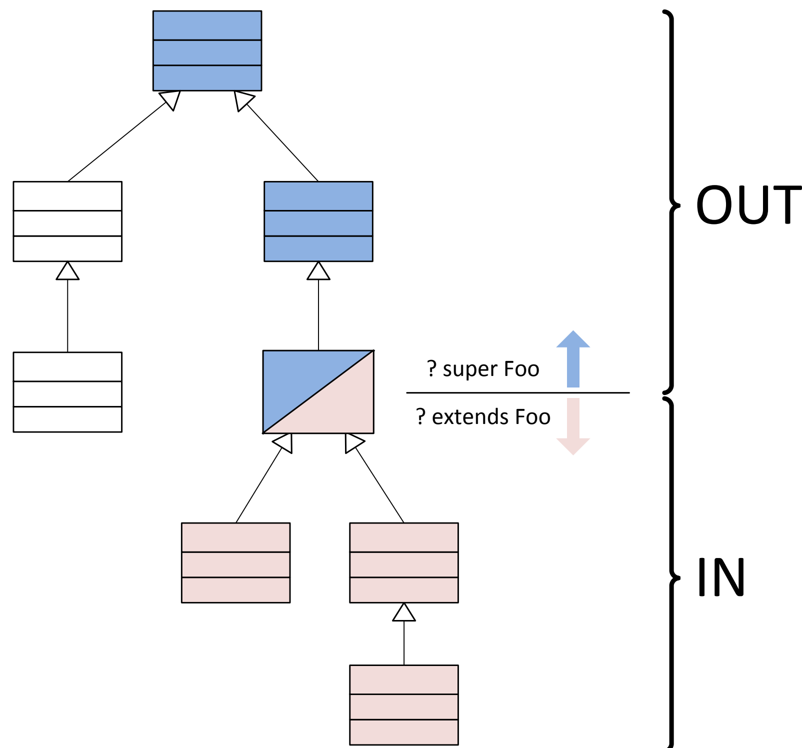
Podczas doboru symboli wieloznacznych pomocne może być rozróżnienie pomiędzy zmiennymi wejściowymi i wyjściowymi

- Zmienne wejściowe (IN) dostarczają danych
- Zmienne wejściowe (OUT) konsumują dane (używają, przechowują, zapisują)

We wcześniejszym przykładzie

```
static <T>
void append(List<? super T> target, List<? extends T> src){
    for(T t:src)target.add(t);
}
```

`target` było zmienną wyjściową, `src` wejściową



Dobór symboli

IN	`<? extends Foo>`
IN dostępna jako `Object`	`<?>`
OUT	`<? super Foo>`
IN i OUT równocześnie	Nie należy używać symboli wieloznacznych (lub użyć metod pomocniczych)

Usuwanie informacji o typach

Implementacji typów generycznych towarzyszyło założenie, że:

- Nie będą generowane instancje parametryzowanych typów, czyli np. obiekty typów `List<String>` i `List<Double>` będą obsługiwane przez wspólny kod posługujący się referencjami typu `Object`.
- Nie będzie zmieniana specyfikacja maszyny wirtualnej Java tak, aby dołączyć dodatkowe informacje o typach parametryzowanych. Kod powstały w wyniku kompilacji typów generycznych będzie zawierał wyłącznie standardowe klasy wraz z metodami oraz interfejsy.

Przyjętym rozwiązaniem jest więc znajomość typów i wnioskowane o typach w trakcie kompilacji oraz usunięcie tej informacji (ang. *type erasure*) w skompilowanym kodzie.

Podczas kompilacji dokonywane są pewne adaptacje:

- Parametry typów generycznych są zastępowane referencjami typu `Object` lub typami wynikającymi z ograniczeń.
- W wygenerowanym kodzie wstawiane są operatory rzutowania
- W niektórych przypadkach dziedziczenia mogą być dodane metody adaptujące typy.

Usuwanie informacji o typach dla parametryzowanych klas

Podczas usuwania informacji o typach kompilator zastępuje nieograniczone parametry referencjami `Object`, natomiast parametry ograniczone typami wynikającymi z ograniczeń.

Przykład 1

```
class Box<T>{
    T t;
    void set(T t){
      this.t=t;
    }
    T get(){return t;}
}
```

```
class Box {
    Object t;
    void set(Object t){
      this.t=t;
    }
    Object get(){return t;}
}
```

Przykład 2

```
class Fraction
<T extends Number>{
  T num;
  T den;
  void set(T num, T den)
  {
      this.num = num;
      this.den = den;
  }
  double asDouble(){ return
     num.doubleValue()
     /den.doubleValue();
  }
}
```

```
class Fraction
{
  Number num;
  Number den;
  void set(Number num,
      Number den){
      this.num = num;
      this.den = den;
  }
  double asDouble(){ return
     num.doubleValue()
     /den.doubleValue();
  }
}
```

Usuwanie typów dla metod generycznych

W przypadku metod generycznych działanie kompilatora jest analogiczne.

Przykład 1

`<T> void print(T[] tab){`
`  for(T t:tab){`
`    System.out.print(t+";");`
`  }`
`}`
	`void print(Object[] tab){`
	`  for(Object t:tab){`
	`    System.out.print(t+";");`
	`  }`
	`}`

Przykład 2

`static <T extends Number>`
`double sum(T[] tab){`
`    double sum=0;`
`    for(T t:tab){`
`        sum+=t.doubleValue();`
`    }`
`    return sum;`
`}`
	`static`
	`double sum(Number[] tab){`
	`    double sum=0;`
	`    for(Number t:tab){`
	`        sum+=t.doubleValue();`
	`    }`
	`    return sum;`
	`}`

Wywołania metod i metody adaptacyjne

Jeżeli wołana metoda zwraca typ będący parametrem dla typu generycznego, dodawane są operatory rzutowania.

`Box<String> fs= new Box <>();`
`fs.set("Test");`
`String v = fs.get();`
	`Box fs= new Box ();`
	`fs.set("Test");`
	`String v = (String)fs.get();`

W szczególnych przypadkach polimorfizmu mogą zostać wygenerowane metody adaptacyjne (ang. *bridge methods*).

<table>
<tr><td><pre>class FixedBox
extends Box<String>{
    void set(String t){
        t=t.trim();
        super.set(t);
    }
}</pre></td><td><pre>class FixedBox
extends Box{
    void set(String t){
        t=t.trim();
        super.set(t);
    }
    void set(Object t){
       set((String)t)
    }

}</pre></td></tr>
<tr><td><pre>Box<String> b = new
FixedBox();
b.set("Ala ma kota");</pre></td><td><pre>Box b = new FixedBox();
b.set("Ala ma kota");</pre></td></tr>
</table>

Klasa `FixedBox` dziedziczy po klasie `Box` parametryzowanej typem `String` i przedefiniowuje metodę `set()` zmieniając typ jej parametru na `String`. W rezultacie metody `set(Object t)` i `set(String t)` stają się niepowiązane.

W przypadku instrukcji

```
b.set("Ala ma kota");
```

nastąpiłoby wywołanie metody `set()` zdefiniowanej w klasie `Box`, a nie metody przedefiniowanej w `FixedBox`.

Informacja o typach w trakcie wykonania

W trakcie wykonania programu dostępna jest informacja o typach klas lub interfejsów w postaci obiektów klasy `Class`. Dla dowolnego obiektu informacja o klasie zwracana jest przez metodę `Class getClass()` klasy `Object`.

Typ nazywany jest reifikowanym (ang. *reifiable*), jeżeli pełna informacja o typie jest reprezentowana przez obiekt klasy `Class` i jest dostępna w trakcie wykonania. Termin pochodzi od łacińskiego słowa *rei*, czyli rzecz, przedmiot.

Reifikowane są: typy wbudowane, niegeneryczne, surowe typy i typy generyczne typy nieograniczone `<?>`.

Pozostałe typy generyczne nie są reifikowane – pełna informacja o typie jest usunięta i niedostępna w trakcie wykonania.

```
System.out. println(new ArrayList<String>().getClass());
System.out. println(new ArrayList<Integer>().getClass());
```

Wynik:

```
class java.util.ArrayList
class java.util.ArrayList
```

Ograniczenia dla typów generycznych

- Nie można utworzyć instancji typu generycznego parametryzowanej typem wbudowanym.

  Nie jest możliwe posługiwanie się typami postaci `List<int>` czy `List<double>`. Muszą być użyte ich opakowania obiektowe: `List<Integer>` lub `List<Double>`

- Nie jest możliwe utworzenie instancji parametrów typu generycznego

```
class Factory<T>{
    T get(){
        return new T(); // BŁĄD
    }
}
```

- Nie jest możliwe zadeklarowanie pól statycznych, których typ jest użyty jako parametr.

```
    class Statics<T>{
        static T common=null;  // BŁĄD
        T get(){
            return common;
        }
    }
```

Gdyby było to możliwe, obiekty typu `Statics<String>` i `Statics<Double>` miałyby wspólne pole `common` o trudnym do określenia typie.

- Nie jest możliwe użycie operatora `instanceof` dla typów parametryzowanych.

Przyczyną jest brak ich reifikacji. Kompilator nie skompiluje kodu, w którym testowana jest przynależność do parametryzowanej klasy.

```
ArrayList<Integer> ai = new ArrayList<>();
boolean ok = (ai instanceof ArrayList<Integer>); BŁĄD
```

- Nie jest możliwe utworzenie tablic typów parametryzowanych

```
①        List<String>[] tab = new List<String>[10];
②        List[] rawtab = new List<String>[10];
③        rawtab[0]=new ArrayList<String>();
④        rawtab[0]=new ArrayList<Integer>();
```

Instrukcja ① nie wydaje się przynosić problemów. Jednakże potencjalnie możliwe jest ② przypisanie tablicy do surowego

typu. W takim przypadku nie byłaby możliwa kontrola nad dodawanymi elementami.

Nie jest to znaczącym ograniczeniem. Można posłużyć się konstrukcją:

```
List<List<String>>tab = new ArrayList<>();
```

- Nie jest możliwe użycie typów parametryzowanych jako wyjątków oraz ich przechwytywanie.

  Podstawą mechanizmu obsługi wyjątków jest dopasowanie handlera do typu wyjątku. W przypadku parametrów typów generycznych oraz typów parametryzowanych informacja o typie jest zamazywana.

- Nie jest możliwe przeciążanie metod, dla których formalne parametry metod po usunięciu informacji o typach sprowadzają się do tych samych surowych typów.