

# Generické typy

jsou třídy či interfejsy deklarující tzv. typové parametry jimiž:

- systematizují typovou kontrolu kompilátorem,
- vyjadřují jasněji smysl, zlepšují čitelnost a robustnost programu,
- ulehčují psaní programu v IDE,
- v class-souborech jsou vyznačeny, ale lze je pomínout ( tzv. erasure ),
- v runtime se nijak neuplatňují – tam jsou jen tzv. hrubé ( raw ) typy.
- užívají se zejména v kolekcích k vymezení typů prvků.

Typové parametry lze užít **jen na referenční typy - leč nikoli na:**

- **ve statickém kontextu:** `static void m( ) { List<T> lst= ... }`
- **pole a vytvoření pole:** `E[ ] e = ( E[ ] ) new Object[5]; new E[3]  
new ArrayList<String>[7]`
- **primitivní typy:** `List<int>` - **lze** referenční typy `List<int[ ]>`
- **statické atributy:** `static T t`
- **vytvořit instanci:** `new E( )`
- **potomky třídy Throwable:** `class MyEx<T> extends Exception { }`

Podrobněji: The Java Language Specification ( Third Edition ) § 4.4-4.9, § 18

# Syntax hlaviček

```
... class JménoTřídy [ < Deklarace > ] // generická třída  
    [ extends JménoNadTřídy [ < TypArgument, ... > ] ]  
    [ implements JménoInterfejsu [ < TypArgument, ... > ], ... ] {...}
```

```
... interface JménoInterfejsu [ < Deklarace > ] // generický interfejs  
    [ extends JménoNadInterfejsu [ < TypArgument, ... > ], ... ] {...}
```

```
... [ < Deklarace > ] JménoTřídy // generický konstruktor  
    ( [ [ final ] ( Typ | TypParm ) jménoParametru, ... ] ) {...}
```

```
... [ [ static ] [ final ] | [ abstract ] ] // generická metoda  
    [ < Deklarace > ] ( void | ReturnTyp | TypParm ) jménoMetody  
    ( [ [ final ] ( Typ | TypParm ) jménoParametru, ... ] ) ( {...} | ; )
```

< Deklarace > je seznam typových parametrů:  $\langle T_1, T_2, \dots, T_i, \dots, T_n \rangle$

- $T_i$  ve třídě či interfejsu jsou globální a nelze je užít ve statickém kontextu,
- $T_i$  konstruktoru či metodě jsou lokální a lze je zastínit.

# Typový parametr ( § 4.4 )

deklarovaný v seznamu  $\langle T_1, T_2, \dots, T_i, \dots, T_n \rangle$  může být vymezen takto:

- $T_i$  [ `extends java.lang.Object` ] - je jakákoli třída.
- $T_i$  `extends Z` - Z je již známá typová proměnná.
- $T_i$  `extends ( R | I ) [ & I & I ... ]` - R je třída  
I interfejs  
& - logický součin.

Typové parametry se obvykle značí jednopísmenným identifikátorem:

T, E, K, V, N, S, A pro: Type, Element, Key, Value, Number, Service, Action

Příklady :

```
class Demo < T, E extends Y & K , N extends J & K > { }
```

```
public < Y, K > K demo( Y y, K k, int i ) { ... return k; }
```

kde Y je např. třída – J a K jsou interfejsy.

# Parametrizovaný objekt

vytvoří konstruktor s typovými argumenty v počtu *a* s vymezením formálních parametrů dle definice typu:

```
new Demo < A, B, C > ( ... )
```

kde *A*, *B*, *C* jsou vhodné referenční typy, tj. třídy nebo interfejsy.

Parametrizovaný objekt lze referovat čtyřmi způsoby:

```
Demo          ref0 ;           // raw: Object, Object, Object
Demo < A,B,C > ref1 ;           // ref1: A,B,C

ref1 = new Demo < A,B,C > (...); // decorated type
      = new Demo <> (...);       // inference diamantem <> od v. 7
ref0 = ref1 ;                   // erasure
ref0 = new Demo (...);         // raw type
ref1 = ref0 ;                   // unchecked conversion
```

# Příklad: Parametrizovaný Interval

```
public class Interval <E extends Comparable<E>>
    implements Comparable<Interval<E>> {
    E low, high;

    public Interval( E low, E high ) {
        if ( low.compareTo( high ) >0 ) throw new IllegalArgumentException( );
        this.low=low; this.high=high;
    }
    @Override
    public int compareTo( Interval<E> that ) {
        return this.low.compareTo( that.low );
    }
    @Override
    public String toString( ) { return "Interval["+low+", "+high+"]"; }
}
class XI <E extends Comparable<E>> extends Interval<E> {
    public XI( E a, E b ) { super(a,b); }
}
```

# Typový argument se žolíkem ( wildcard )

Reference, tj. proměnné a parametry metod, lze vymezit jako množinu přijatelných neznámých typů takto:

- `< ? [ extends Typ | extends java.lang.Object ] >`
  - zde `extends` zahrnuje i `implements`  
Typ vyznačuje horní mez včetně.  
Nelze vkládat nebo měnit hodnoty ( kromě `null` )
    - jen k nim přistupovat ( a v kolekcích i odstraňovat ) .
- `< ? super Typ >` - Typ vyznačuje dolní mez včetně.  
Lze vkládat hodnoty.

Např.: pro výpis lze užít neomezený žolík:

```
void printAny( List< ? > c ) {                               // ne < Object >
    for ( Object o : c ) System.out.println( o );
    c.get( 0 );                                             // lze - nemá param. typ
    c.add( new Object( ) );                                 // má param. E - compile error
}
```

## ? v referenci

Budiž věci tříd ...  $\leftarrow A \leftarrow B \leftarrow C \leftarrow D$  referované proměnnými a, b, c, d .

```
class Box< V extends B > { // Tato krabice umožní uložit jednu věc
    V v ; // splňující: v instanceof B
    Box ( V v ) { this.v = v; } // nutno vložit b či c či null
    V get( ) { return v; }
    void set( V v ) { this.v = v; } // vždy lze vložit null
}
```

Krabice slouží k řádnému uložení, reference vymezuje vklad a výběr věcí:

```
Box< B>          x1=new Box<>( b );    x1.set( b... );    B z1=x1.get();
Box< C>          x2=new Box<>( c );    x2.set( c... );    C z2=x2.get();
Box< ? super B > x3=new Box<>( b );    x3.set( b... );    B z3=x3.get();
Box< ? super C > x4=new Box<>( b | c ); x4.set( c... );    B z4=x4.get();
Box< ? extends B > x5=new Box<>( b... );    B z5=x5.get();
Box< ? extends C > x6=new Box<>( c... );    C z6=x6.get();
Box< ?[extends A] > x7=new Box<>( b... );    B z7=x7.get();
```

```
@ SuppressWarnings( "rawtypes" ) // potlačení před metodou či class
```

```
Box x= new B<>( b | c );    x.set( b | c );    B z= x.get( );
```

# Příklad syntaxe

```
class Demo < X , Y extends Number, Z extends AbstractMap < K, V >
    implements NavigableMap < K, V > ... {

    private Comparator< ? super K > comparator = null ;

    public TreeMap( Comparator< ? super K > c ) { comparator = c ; ...

    public TreeMap( Map< ? extends K, ? extends V > m ) { putAll( m ); ...
    ...
}

Map < String, Integer > m = new TreeMap < String, Integer > ( ) ;
```



# TreeMap jako příklad

```
class TreeMap< K, V > extends AbstractMap < K, V >
    implements NavigableMap < K, V > ... {

    private Comparator < ? super K > comparator = null ;

    public TreeMap( Comparator < ? super K > c ) { comparator = c ; ...

    public TreeMap( Map < ? extends K, ? extends V > m ) { putAll( m ); ...
    ...
}
```

```
Map < String, Integer > m = new TreeMap< String, Integer > ( ) ;
```

```
    m.put ( "AAA" , 1);
```

```
    int i = m.get ( "AAA" );
```

# Příklad parametrizované metody

Parametrizovat metody lze i v neparametrizovaných třídách i interfejsích.

```
public static < U extends Double, V > U m( U u, V v ) {  
    U x = u;  
    V y = v;  
    System.out.println( u.getClass( ) + " " + v.getClass( ) );  
    System.out.println( x + " " + y );  
    return x;  
}
```

....

```
Number s = m ( 666.999, new Stack( ) ); // čísla převádí autoboxing  
Double d = m ( .777, "BBB" );  
Object o = m ( 3.14, 3L );
```

Parametrizovaný konstruktor má smysl jen v parametrizovaných třídách.

# Generika a subtypy

Důležitá, byť neintuitivní, zábrana proti narušení vymezení:

Pro vztah Subtyp --> Supertyp ( extends / implements )

neplatí: L<Subtyp> --> L<Supertyp>

Příklad: `class Driver extends Person { ... }`

```
List< Driver > drivers = new ArrayList<> ( );
```

```
List< Person > people = drivers; // error
```

```
people.add( new Person( ) ); // non-driver in drivers
```

# Erasure

```
Collection<String>    cs = new ArrayList <>( );  
                      cs.add( "AAA" );  
Collection            cr = cs;                // raw type  
                      cr.add( 666 );         // unsafe operation  
Collection<?>        cx = cs;  
                      cx.add( "BBB" );      // error  
Collection<? super String >    cx = cs;  
                      cx.add( "CCC" );  
Collection<? extends String > cy = cs;  
                      cy.add( "DDD" );      // error  
Collection<Object>    co = cs;              // error incompatible types
```

# Příklad řazení heterogenní kolekce

```
List< Comparable< ? > > list = new ArrayList<>( );
```

```
// vložení objektů různých tříd avšak splňující Comparable.
```

```
Collections.sort( list, new MyComp( ) ); // řazení
```

```
@ SuppressWarnings( { "rawtypes" } )
```

```
class MyComp implements Comparator< Comparable > {
```

```
    @ Override
```

```
    @ SuppressWarnings( { "unchecked" } )
```

```
    public int compare( Comparable o1, Comparable o2 ) {
```

```
        Class c1 = o1.getClass( ), c2 = o2.getClass( );
```

```
        return c1 == c2 ?
```

```
            - o1.compareTo( o2 ) // téže třídy
```

```
            : - c1.getName( ).compareTo( c2.getName( ) ); // různých tříd
```

```
    }
```

```
}
```