Generické programování

- generika umožňují používání generických typů při deklaraci tříd, rozhraní a metod
- cílem je:
    - nebýt omezen datovými typy (typ Object komplikuje datovou kontrolu)
    - rozšíření typové kontroly
    - využít kód opakovaně s různými datovými typy

- jsou to třídy/rozhraní deklarující tzv. typové parametry, jimiž:
    - systematizují typovou kontrolu kompilátorem
    - vyjasňují smysl, zlepšují čitelnost a robustnost programu
    - ulehčují psaní programu v IDE
    - v class-souborech jsou vyznačeny, ale lze je pominuout (erasure)
    - v runtime se nijak neuplatňují
    - užívají se zejména v kolekcích k vymezení typů prvků

- **nelze užít:**
    - ve statickém kontextu
    - statické atributy (private static T name)
    - k vytvoření pole
    - primitivní typy (List<int>)
    - k vytvoření instance (new E())
    - potomky třídy throwable (Class MyEx<T> extends Exception)

- nutno deklarovat generický typ před jeho použitím
    - např. v názvu třídy – public class Box<T>
    - pak se na něj stačí odkazovat jen pomocí T
    - další příklad - public < Y, K > K demo(Y Y, K K, INT I) { … RETURN K; }

```
LinkedList<String> list = new LinkedList<String>();
list.add("abc");
String s = list.get(0); // není nutnost přetypovat


LinkedList list = new LinkedList();
list.add("abc");
String s = (String)list.get(0); // nutné přetypování
```

```
Pair<Int, char> p = new Pair<>(8, 'a'); // compile-time error

Pair<Integer, Character> p = new Pair<>(8, 'a'); //OK
```

```
CLASS PAIR<K, V> {

   PRIVATE K KEY;
   PRIVATE V VALUE;

   PUBLIC PAIR(K KEY, V VALUE) {
      THIS.KEY = KEY;
      THIS.VALUE = VALUE;
   }

   // ...
}
```

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

... interface  JménoInterface [ < Deklarace > ]        // generický interface
        [ extends JménoNadInterface [ < 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: < T_1 , T_2, ... T_i, ... T_n >
T_i ve třídě či interface jsou globální a nelze je užít ve statickém kontextu,
T_i konstruktoru či metodě jsou lokální a lze je zastínit.

Typový parametr

- TP deklarovaný v seznamu < T_1 , T_2, ... T_I, ... T_N > může být vymezen takto:
  - **T_I [EXTENDS JAVA.LANG.OBJECT]** … jakákoliv třída
  - **T_I EXTENDS Z** … Z je již známá typová proměnná
  - **T_I EXTENDS (R | I) [& I & I ...]**
    - R = třída
    - I = interface
    - & = logický součin

- TP se obvykle značí **jednopísmenným identifikátorem**:
  - T … Type
  - E … Element
  - K … Key
  - V … Value
  - N … Number
  - S … Service
  - A … Action

- parametrizovaný objekt lze **referovat** čtyřmi způsoby
  - Demo ref0; // RAW: Object, Object, Object
  - Demo<A,B,C> ref1; // REF1: A,B,C

1.) ref1 = new Demo<A,B,C>(…); // decorated type
  a. new Demo<>(…); // inference diamantem <>
2.) ref0 = ref1; // erasure
3.) ref0 = new Demo(…); // raw type
4.) 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); }
}
```

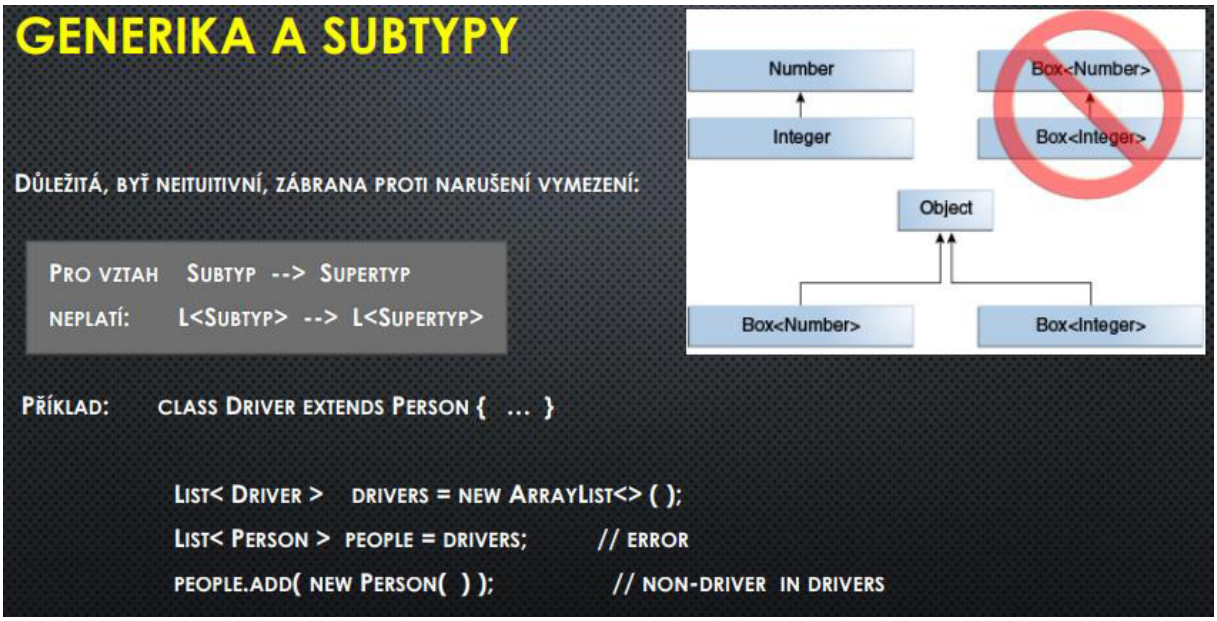
Typový argument se žolíkem (wildcard)

- žolík = ? = neznámý typ, může být cokoliv
- reference (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 či měnit hodnoty (kromě null)
        - lze k nim jen přistupovat (a v kolekcích i odstraňovat)
    - **< ? super TYP >**
        - typ vyznačuje dolní mez včetně
        - lze vkládat hodnoty

- **příklad (neomezený žolík pro výpis):**

```
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
 }
```

Generika a subtypy



Erasure

- proces vynucování typových omezení jen při kompilaci, při běhu se informace o typu vyřadí

```
public static <E> boolean containsElement(E [] elements, E element){
   for (E e : elements){
      if(e.equals(element)){
         return true;
      }
   }  return false;
}
```

Před kompilací

```
PUBLIC STATIC BOOLEAN CONTAINSELEMENT(OBJECT [] ELEMENTS, OBJECT ELEMENT){
   FOR (OBJECT E : ELEMENTS){
      IF(E.EQUALS(ELEMENT)){
         RETURN TRUE;
      }
   }  RETURN FALSE;
}
```

Po kompilaci

```
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 – kolekce objektů různých tříd, ale splňující Comparable

```
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
  }
}
```

Co generika neumí

- jsou vytvořeny nad kompilátorem, v bytekódu neexistují
- nemůžeme používat typové parametry za běhu programu (již neexistují)

Reflexe (jak obejít zapouzdření)

- možnost inspekce generik v runtime
- extrémně pomalá

```
private static void set(Object object, String var, Object value)
    throws NoSuchFieldException, IllegalArgumentException, IllegalAccessException {

    Field f = object.getClass().getDeclaredField(var);
    f.setAccessible(true);
    f.set(object, value);

    // TODO: můžeme odchytávat výjimky a vyhazovat vlastní výjimku případně běhovou
}
```