



11

Generika

- Generika
- Syntax hlaviček
- Typový parametr
- Parametrizované objekty
- ? v referenci
- Erasure
- Relfexe



List<T>

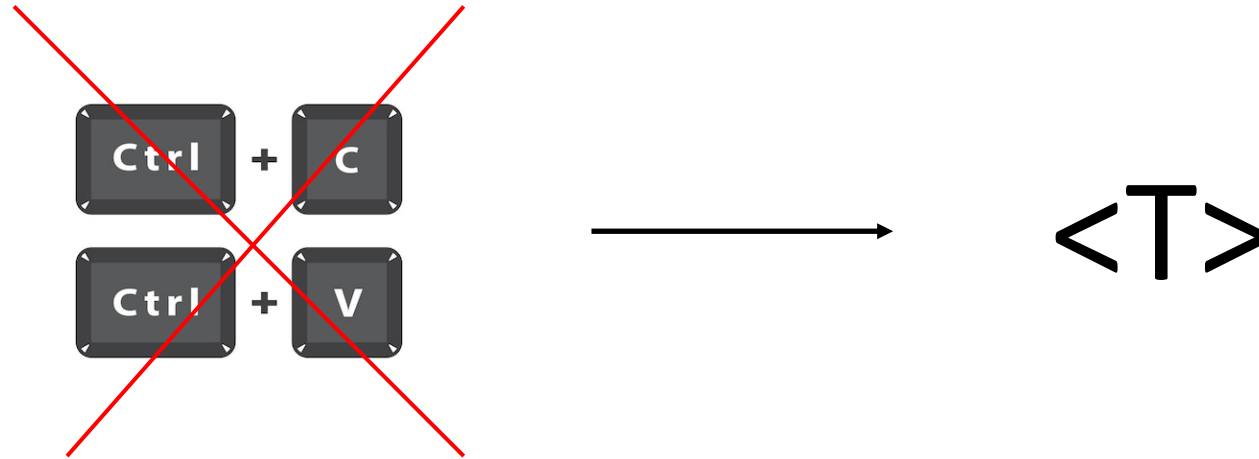
11 Co jsou generika?

- Umožňují používání generických typů při deklaraci tříd, rozhraní a metod.
- K dispozici od JDK 1.5

11 Motivace

Cílem je:

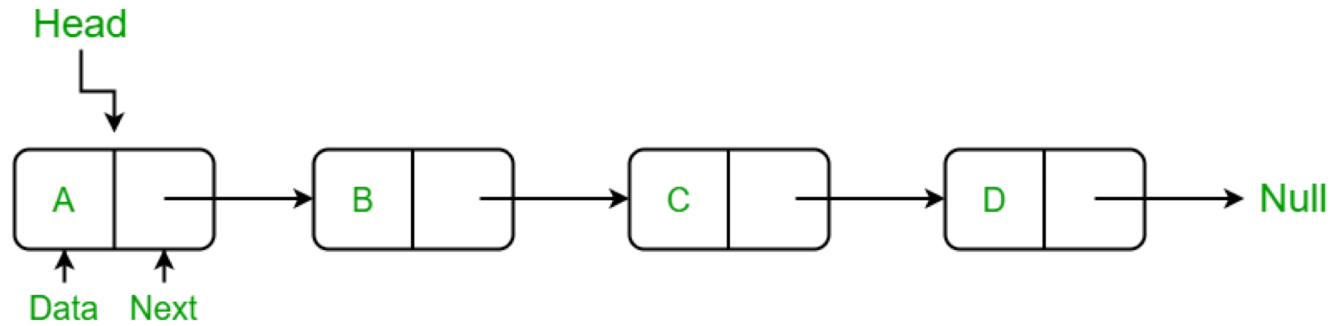
- Nebýt omezen datovými typy
 - Typ *Object* komplikuje/znemožňuje datovou kontrolu
- Rozšíření typové kontroly
- Využít kód opakovaně s různými datovými typy



11 Příklady co všichni známe

- Spojové seznamy (jednosměrný, obousměrný, cyklický...)
- Mapy (*TreeMap*, *HashMap*...)
- Fronty
- Sety
- ...

11 Spojový seznam bez generik



Co když budu chtít udělat list integerů, doublů nebo dalších typů?

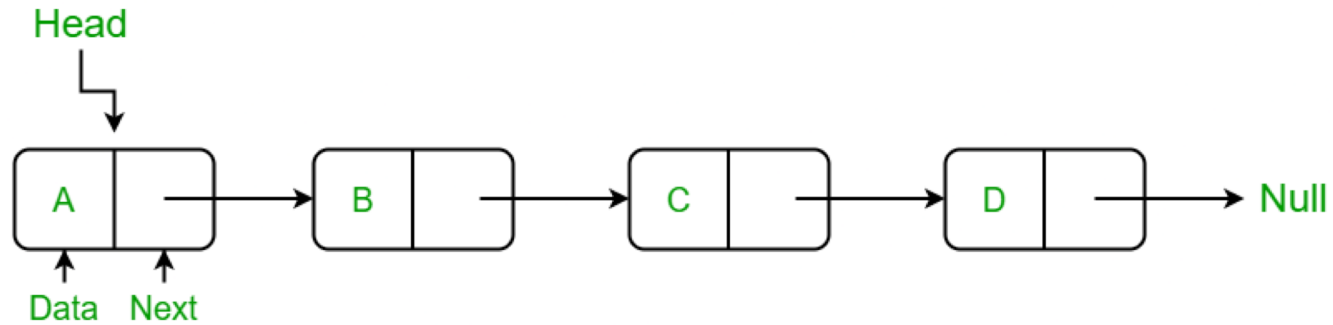
A co hůř, co jednotlivé objekty v aplikaci?

```
class Node {
    String value;
    Node next;

    Node (String value, Node next) {
        this.value = value;
        this.next = next;
    }
}

Node tmp = new Node ("D", null);
tmp = new Node ("C", tmp);
tmp = new Node ("B", tmp);
Node linkedList = new Node ("A", tmp)
```

11 Spojový seznam s generiky



Všechno lze snadno vyřešit pomocí generik.
Je možné vytvořit samostatné listy pro integer,
double a další.

Dokonce ani objekty nebudou takový pain.

```
class Node<T> {
    T value;
    Node next;

    Node (T value, Node next) {
        this.value = value;
        this.next = next;
    }
}

Node<String> tmp = new Node ("D", null);
tmp = new Node ("C", tmp);
tmp = new Node ("B", tmp);
Node<String> linkedList = new Node ("A", tmp)
```

11 Další příklady

ArrayList

```
● ● ●  
  
List<String> wordList = new ArrayList<>();  
wordList.add("Hi");  
wordList.add("Hello");  
wordList.add("Hola");
```

HashMap

```
● ● ●  
  
Map<String, String> wordMap = new HashMap<>();  
wordMap.put("Ahoj", "Hi");  
wordMap.add("Ahoj", "Hello");  
wordMap.add("Nashledanou", "Bye");
```

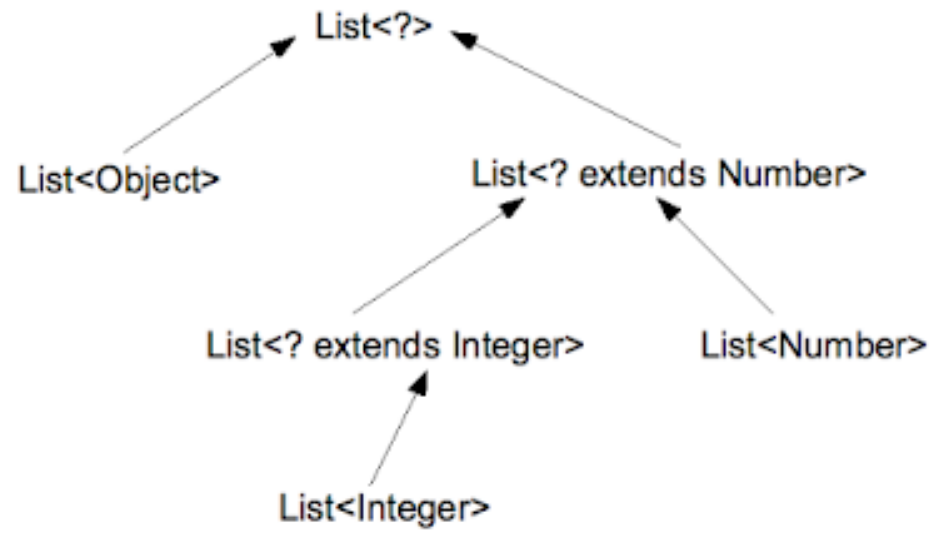
Co když nedefinuju typ?

```
● ● ●  
  
Map map = new HashMap();  
map.put("1", 1);  
Integer x = map.get("1"); // Object cannot be converted to Integer
```

Je potřeba hodnotu přetypovat, jinak je to Object

```
● ● ●  
  
Map map = new HashMap();  
map.put("1", 1);  
Integer x = (Integer) map.get("1");
```

11 Dědičnost



Taxonomy of the generic *List* types

11 Generické typy

Jsou třídy či interface 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 je pomínout (tzv. erasure)
- V runtime se nijak neuplatňují – tam jsou jen tzv. hrubě (raw) typy
- Užívají se zejména v kolekcích k vymezení typů prvků



List<T>

11 Generické typy

Typové parametry lze užít jen na referenční typy, ale nikoliv:

- Ve statickém kontextu

```
static void M() { List<T> list = ...}
```

- Na statické atributy

```
static T t = ...
```

- V poli pro vytvoření pole

```
E[] e = (E[]) new Object[5]
```

- Na vytvoření instance

```
E e = new E()
```

- Na primitivní typy

```
List<int> list = ...
```

- Na potomky třídy throwable

```
class MyException<T> extends Exception {}
```

11 Generické typy - příklad

```
class Pair<K, V> {  
    private K key;  
    private V value;  
  
    public Pair (K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    // ...  
}  
  
Pair<int, char> p = new Pair<8, 'a'>; // compile-time error  
Pair<Integer, Character> p = new Pair<8, 'a'>; // OK
```

Důvod compile erroru?

Viz předchozí slide, v jako hodnoty nelze použít primitivní datové typy.

11 Syntax hlaviček

```
...class JménoTřidy [<Deklarace>] // generická třída  
    [extends JménoNadTřidy[<TypArgument,...>]  
    [implements JménoInterface[<TypArgument,...>, ...] {...}]
```

```
... interface JménoInterface [ < Deklarace > ] // generický interface  
    [ extends JménoNadInterface [ < TypArgument, ... > ] , ... ] {...}
```

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

< Deklarace > je seznam typových parametrů: < T₁ , T₂ , ... 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.

11 Typový parametr

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] - **T_i** je jakákoliv třída
- **T_i** extends **Z** - **T_i** musí být potomkem již definované třídy **Z**
- **T_i** extends (**R** | **I**) [**&** | **&** | ...] - **R** je třída
- **I** je interface
- **&** je logický součin

▪ Typové parametry se obvykle značí jednopísmenným identifikátorem:

- **T** – type
- **E** – element
- **K** – key
- **V** – value
- **N** – number
- **S** – service
- **A** - action



```
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 interfacy.
```

11 Parametrizovaný objekt

- Vytvoří konstruktor s typovými argumenty v počtu a s vymezením formálních parametrů dle definice typu:
 - New Demo <A, B, C> (...)
- 4 možné způsoby parametrizace:

```
Demo ref; // generic objects are Object, Object, Object
Demo<A, B, C> genericRef;

genericRef = new Demo<A, B, C>(...); // Decorated typ
           = new Demo<> (...); // Inference diamantem <> od verze 7
ref = genericRef; // Erasure
ref = new Demo(...); // Raw type
genericRef = ref; // Unchecked conversion
```

- Příklad:

```
public class Interval TODO {
    TODO low, high;

    public Interval( TODO ) {
        if ( low.compareTo( high ) > 0 )
            throw new IllegalArgumentException( );
        this.low=low; this.high=high;
    }

    @Override
    public int compareTo( TODO {
        return this.low.compareTo( that.low );
    }

    @Override
    public String toString( ) {
        return "Interval["+low+", "+high+"]";
    }
}
```

11 Parametrizovaný objekt

- Řešení:

```
public class Interval <E extends Comparable<E>> implements Comparable<Interval<E>> {
    private 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); }
}
```

11 Wildcard – Typový argument se žolíkem

- Reference (proměnné a parametry metod) lze vymezit jako množinu přijatelných neznámých typů
 - `< ? [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

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


11 Wildcard – Typový argument se žolíkem

```
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 nebo C nebo null

    V get( ) {
        RETURN V;
    }

    void set( V v ) {
        this.v = v;
    }
    // vždy lze vložit null
}
```

- Máme definovanou hierarchii
A <- B <- C <- D (B je potomek A, ...)
- 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();`

11 Příklad

- Upravte následující kód tak, aby šel spustit:

```
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>();
m.put("AAA", 1);
int i = m.get("AAA");
```

11 Příklad - řešení

```
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;
        ...
    }

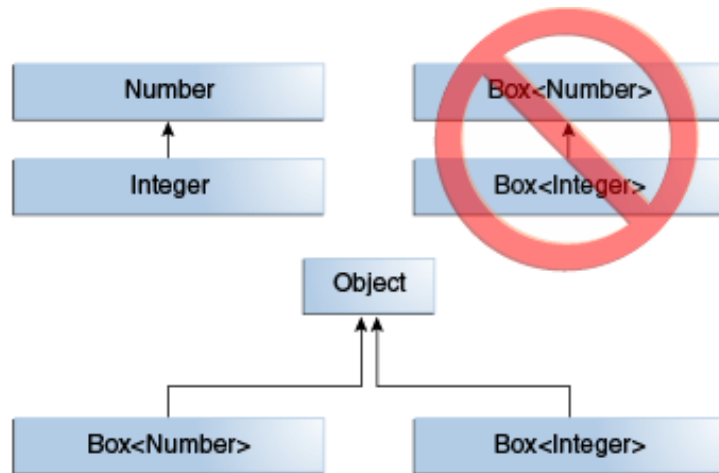
    public TreeMap(Map<? extends K, ? extends V> m) {
        putAll(m);
        ...
    }
}

Map<String, Integer> m = new TreeMap<String, Integer>();
m.put("AAA", 1);
int i = m.get("AAA");
```

- Co kdybychom chtěli dát podmínku na String a Integer?

11 Generika a subtypy

- Důležitá, byť neintuitivní zábrana proti narušení vymezení:
 - Pro vztah SubTyp → SuperTyp neplatí L<SubTyp> → L<SuperTyp>



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

List<Driver> drivers = new ArrayList<>();
List<Person> people = drivers; // error
people.add(new Person()); // non-driver in drivers
```

11 Erasure

- Proces vynucování typových omezení pouze při kompilaci a vyřazení informací o typu prvku při běhu

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

// Po kompilaci
public static boolean containsElement(Object [] elements, Object element){
    for (Object e: elements) {
        if(e.equals(element)) {
            return true;
        }
    }
    return false;
}
```

11 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: Napište třídu heterogenní kolekce, do které můžeme vložit různé objekty, ale bude splňovat Comparable (má metodu compare ;)).

11 Erasure – řešení příkladu

```
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
    }
}
```

11 Co generika neumí?

- Jedná se o „+“ nad kompilátorem (v bytekódu neexistují).
- Nemůžeme používat typové parametry za běhu programu (v daný okamžik již neexistují).
 - Dané typy předat konstruktoru (při konstrukci objektu je znát musíme).

11 Reflex – jak obejít zapouzdření

```
private static void set(Object object, String varr, Object value) throws NoSuchFieldException, IllegalArgumentException, IllegalAccessException {
    Field f = Object.getClass().getDeclaredField(varr);
    f.setAccessible(true);
    f.set(object, value);
    // TODO: MŮŽEME ODCHYTÁVAT VÝJIMKY A VYHAZOVAT VLASTNÍ VÝJIMKU PŘÍPADNĚ BĚHOVOU
}
```

Rychlost ☹️

- Přímý přístup: celkový čas = 5ms (100%)
- Setter: celkový čas = 5ms (100%)
- Reflexe: celkový čas = 151ms (3020%)
- Připravená reflexe: celkový čas = 16ms (320%)
 - Pokud najdeme field jen poprvé a poté ho jen využíváme

Absolutní časy jsou pro milion opakování