

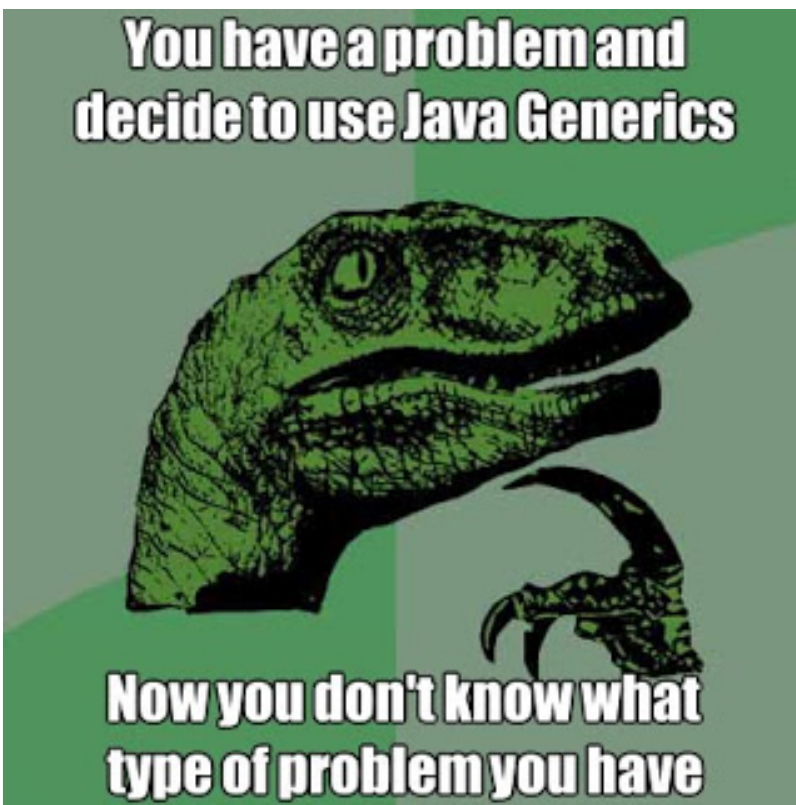
Programování v jazyce Java

```
<link rel="stylesheet" href="http://cdnjs.cloudflare.com/ajax/libs/font-awesome/3.1.0/css/font-awesome.min.css">
```

Generické typy, typové parametry

- Generické typy = něco *obecně použitelného*, zobecnění
- Třídy v Javě mají společného předka, třídu `Object` (každý objekt je instancí třídy `Object`).
- Potřebujeme-li pracovat s nějakými objekty, o kterých neznáme typ, můžeme využít společného předka a pracovat s ním.
- To umožňuje snadnou implementaci kolekcí, ale například i využití reflexe.

Vtip



Deklarace seznamu bez a s generiky

```
// no generics (obsolete)
public interface List { ... }
// generic type E
public interface List<E> { ... }
```

- do špičatých závorek umístíme symbol — seznam bude obsahovat prvky **E** (předem neznámého) typu
- je doporučováno používat velké, jednopísmenné deklarace
- písmeno vystihuje použití — **T** je **t**ype, **E** je **e**lement
- **E** nahradíme jakoukoliv třídou nebo rozhraním

Jednoduché využití v metodách

```
E get1(int index);
Object get2(int index);
```

- **get1** vrací pouze objekty, které jsou typu **E** — je vyžadován speciální typ
- **get2** vrací libovolný objekt, tj. musíme pak přetypovávat

```
boolean add(E o);
```

- přidává do seznamu prvky typu **E**

Výhody generik

```
List numbers1 = new ArrayList();
numbers1.add(1);
numbers1.add(new Object()); // allowed, unwanted
Integer n = (Integer) numbers1.get(0);
```

```
List<Integer> numbers2 = new ArrayList<>();
numbers2.add(1);
numbers2.add(new Object()); // won't compile!
n = numbers2.get(0);
```

- do seznamu **numbers1** lze vložit **libovolný objekt**
- při získávání objektů se spoléháme na to, že se jedná o číslo
- do **numbers2** nelze obecný objekt vložit, je nutné vložit číslo

Motivace

- Chceme seznam různých typů seznamů, tak jej vytvoříme následovně:

```
List<List<Object>> listOfDifferentLists;
```

- Máme problém — seznam čísel není seznamem objektů:

```
List<Number> numbers = new ArrayList<Number>();  
List<Object> general = numbers; // won't compile!  
List<? super Number> general2 = numbers; // solution
```



Do seznamu, který obsahuje nejvýše čísla lze vkládat pouze objekty, které jsou alespoň čísla.

Žolíci (wildcards) I

Generika poskytují nástroj zvaný *žolíček* (wildcard), který se zapisuje jako `<?>`.

```
List<Number> numbers = new ArrayList<Number>();  
List<?> general = numbers; // OK  
general.add("Not a number"); // won't compile!
```

- `List<?>` říká, že jde o seznam **neznámých** prvků.
- Jelikož nevíme, jaké prvky v seznamu jsou, **nemůžeme do něj ani žádné prvky přidávat**.
- Jedinou výjimkou je žádný prvek `null`, který lze přidat kamkoliv.



Abstraktní třída `Number` reprezentuje numerické primitivní typy (`int`, `long`, `double`, ...)

Žolíci (wildcards) II

- Ze seznamu neznámých objektů můžeme prvky číst.
- Každý prvek je alespoň instancí třídy `Object`:

```
public static void printList(List<?> list) {  
    for (Object e : list) {  
        System.out.println(e);  
    }  
}
```

Žolíci a polymorfismus I

Následující metoda dělá sumu ze seznamu čísel:

```
public static double sum(List<Number> numbers) {
    double result = 0;
    for (Number e : numbers) {
        result += e.doubleValue();
    }
    return result;
}
...
List<Number> numbers = List.of(1,2,3);
sum(numbers); // it works
List<Integer> integers = List.of(1,2,3);
sum(integers); // won't compile!
```

Žolíci a polymorfismus II

- `Integer` je `Number` a přesto seznam `List<Integer>` nelze použít!
- Nechceme `List<Number>`, řešením je **seznam neznámých prvků, které jsou nejvýše čísla**.

```
public static double sum(List<? extends Number> numbers) { ... }
```

- Toto použití žolíku má uplatnění i v rozhraní `List<E>`, např. v metodě `addAll`:

```
boolean addAll(Collection<? extends E> c);
```

- Uvědomte si následující — žolík je zkratka pro neznámý prvek rozšiřující `Object`.

Žolíci a dědičnost

Další použití žolíků:

- Parametrem metody je instance třídy, která je v **hierarchii mezi třídou specifikovanou naším obecným prvkem `E` a třídou `Object`**.
- Například chceme setřídít množinu celých čísel.
- Existuje třídění podle:
 - hodnoty metody `hashCode()` — na úrovni třídy `Object`
 - čísla — na úrovni třídy `Number`
 - celého čísla — na úrovni třídy `Integer`
- Konstruktor stromové setříděné mapy:

```
public TreeSet(Comparator<? super E> c);
```

Žolíci a více typů

- Deklarace obecného rozhraní setříděné mapy:

```
public interface SortedMap<K,V> extends Map<K,V> { ... }
```

- Je-li třeba použít více nezávislých obecných typů, zapíšeme je do zobáček jako seznam hodnot oddělených čárkou.
- **K** je **key**, **V** je **value**.
- Je možné použít i žolíků, viz následující příklad konstruktorů stromové mapy:

```
public TreeMap(Map<? extends K, ? extends V> m);  
public TreeMap(SortedMap<K, ? extends V> m);
```

Generické metody

- Pro používání generik a žolíků v metodách platí stále stejná pravidla.
- Generická metoda = metoda **parametrizována alespoň jedním obecným typem**.
- Obecný typ nějakým způsobem váže typy proměnných a/nebo návratové hodnoty metody.
- Příklad statické metody, která přenesou prvky z pole do seznamu (pole i seznam musí mít stejný typ):

```
static <T> void arrayToList(T[] array, List<T> list) {  
    for (T o : array) list.add(o);  
}
```

- Ve skutečnosti nemusí být seznam **list** téhož typu — stačí, aby jeho **typ byl nadtřídou** typu pole **array**.
- Např. `Integer[] array` a `List<Number> list`
 - prvky z pole do seznamu se dají kopírovat (i když typy nejsou stejné!), protože `Integer` je `Number`

Generics metody vs. wildcards

- Chceme, aby typ u generické metody spojoval parametry nebo parametr a návratovou hodnotu.
- Ne úplně správné (funkční) použití generické metody:

```
static <T, S extends T> void copy(List<T> destination, List<S> source);
```

- Lepší zápis, **T** spojuje dva parametry metody a přebytečné **S** je nahrazené žolíkem:

```
static <T> void copy(List<T> destination, List<? extends T> source);
```



Metody jsou **public**, viditelnost je vynechána kvůli lepší přehlednosti.

Pole

- Pro pole nelze použít parametrizovanou třídu.
- Při vkládání prvků do pole runtime systém kontroluje pouze *typ vkládaného prvku*.
- Do pole řetězců bychom pak mohli vložit pole čísel a pod.

```
// generic array creation error
public <T> T[] returnArray() {
    return new T[10];
}
```

- Jde však použít třídu s žolíkem, který není vázaný:

```
List<?>[] pole = new List<?>[10];
```

Vícenásobná vazba generik I

- Uvažujme následující metodu, která vyhledává maximální prvek kolekce.

```
static Object max(Collection<T> c);
```

- Prvky kolekce musí implementovat rozhraní **Comparable**, což není syntaxí vůbec podchyceno.
 - Zavolání této metody proto může vyvolat výjimku **ClassCastException!**
- Chceme, aby prvky kolekce implementovali rozhraní **Comparable**.

```
static <T extends Comparable<? super T>> T max(Collection<T> c);
// if generics are removed
static Comparable max(Collection c); // does not return Object!
```

Vícenásobná vazba generik II

- Signatura metody se změnila — má vracet **Object**, ale vrací **Comparable!**
 - Metoda musí vracet **Object** kvůli zpětné kompatibilitě.

- Využijeme tedy **vícenásobnou vazbu**:

```
static <T extends Object & Comparable<? super T>> T max (Collection<T> c);
```

- Po výmazu má metoda správnou signaturu, protože v úvahu se bere první zmíněná třída.
- Obecně lze použít více vazeb, například když je obecný prvek implementací více rozhraní.

Závěr

- Generiky mají i další využití, například u reflexe.
- Tohle však již překračuje rámec začátečnického seznamování s Javou.
- Slidy vychází z materiálů
 - [Javy firmy Sun](#)
 - *Generics in the Java Programming Language* od Gilada Brachy