Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Generické programování v jazyce Java

Generics programming in Java

Bakalářská práce

Autor: Pavel Říha

Vedoucí bakalářské práce: RNDr. Jaroslav Icha

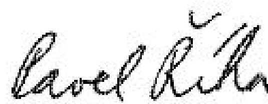
Katedra Informatiky

Rok 2008/2009

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.



V Českých Budějovicích dne *14. 2. 2009*

…………………………..…

Pavel Říha

Anotace

Tato bakalářská práce má za úkol seznámit čtenáře s generickým programováním v Javě a formou praktických příkladů jej seznámit s jejím využitím v praxi. Práce zahrnuje jak popis využití vybraných generických tříd z knihoven obsažených v JDK, tak i ukázky deklarace vlastních generických tříd a metod.

Abstract

This diploma introduces for readers Java generics with using examples and discuses its uses in real and most common problems. Work includes examples of using the Java libraries in JDK, as well as own examples of declaration of generic classes and methods.

Poděkování

Rád bych poděkoval panu RNDr. Jaroslavu Ichovi za cenné připomínky k praktickým příkladům v bakalářské práci a dalším podnětným připomínkám při její realizaci.

Obsah:

1. Úvod 7
   1.1 Co nabízí nového generické programování v jazyku Java? 7
   1.2 Jaký je primární účel Java genericity? 8
   1.3 Jaké jsou výhody užívání Java genericity? 8
2. Generické typy 10
   2.1 Co jsou syrové typy (raw types) 10
   2.3 Deklarace generických a parametrizovaných typů 11
   2.4 Deklarace generického typu 12
   2.5 Ohraničení parametrizace generických typů 12
   2.6 Vytváření instancí generických typů. 13
   2.7 Proč instance generického typu sdílejí stejný runtime typ? 14
3. Parametrické typy 15
   3.1 Konkrétní parametrizovaný typ 15
   3.2 Typy parametrizované pomocí žolíků 15
   3.3 Práce s parametrizovanými typy 16
   3.4 Statické typy uvnitř generických typů. 17
   3.5 Problém dědičnosti a kompatibility parametrů 17
4. Typový argument 19
   4.1 Definice typového argumentu 19
   4.2 Povolené typy typových argumentů 19
   4.3 Primitivní typy jako typové argumenty 20
   4.4 Žolíky jako typové argumenty 22
   4.5 Typové parametry jako typové argumenty 22
   4.6 Omezující typové parametry pro typové argumenty 22
   4.7 Specifikace typového argumentu při použití generického typu 23
   4.8 Specifikace typového argumentu při volání generické metody 23
5. Žolíky 24
   5.1 Definice žolíku 24
   5.2 Neohraničený žolík 24
   5.3 Ohraničený žolík 24
   5.4 Víceúrovňové žolíky 27
   5.5 Vícenásobný žolík 29
6. Meze žolíků 32
   6.1 Mez žolíků 32
   6.2 Povolené meze žolíků 32
   6.3 Rozdíly mezi mezí žolíku a mezí typového parametru 33
7. Generické metody 35
   7.1 Co je to generická metoda 35
   7.2 Jak použít generickou metodu 35
   7.3 Volání generických metod 36
   7.4 Přetěžování generických metod 37
8. Parametrizace a očišťování 40
   8.1 Proces překladu 40
   8.2 Typové očišťování (type erasure): 40
   8.6 Můstkové metody 43
   8.7 Přetypování při překladu 46
9. Nejednoznačnosti, zakázané operace a kolize 49
   9.1 Parametrizované typy jako součást pole 49
   9.2 Hodnoty polí parametrizovaného typu 50

9.3 Použití žolíka při deklaraci objektu 52
10. Příklady vlastních generických typů....54
10.1 Definice vlastního generického typu 54
10.2 Typová bezpečnost a dědičnost 57
10.3 Ohraničení typových hodnot parametrizovaných typů 60
11. Vybrané generické třídy v Java Development Kit 61
11.1 Třída java.lang.Class....61
11.2 Rozhraní java.lang.Comparable a java.util.Comparator 61
11.3 Třída java.lang.Enum<E extends Enum<E>>....64
11.4 Rozhraní java.lang.Iterable<E>, java.util.Iterator<E> 66
11.5 Rozhraní java.util.Collection<E>, rozšiřující rozhraní a implementující třídy....69
11.6 Rozhraní java.util.Map<E>, rozšiřující rozhraní a implementující třídy....74
11.7 Balíčky java.util.concurrent, java.util.concurrent.atomic 80
12. Závěr 81
12.1 Jaké jsou výhody parametrizace generických typů?....81
12.2 Nevýhody parametrizovaných a generických typů:....81
12.3 Výhody použití syrových typů: 82
12.4 Shrnutí:....83

1.Úvod

Generické programování v jazyce Java je novinka, která se objevuila ve verzi JDK 1.5 a umožňuje používání generických typů při deklaraci tříd, rozhraní a metod. To znamená, že můžeme nyní definovat, jaký typ nebo množina typů bude použita v souvislosti s deklarovanou třídou či metodou. Pokud jsme v předchozích verzích Javy deklarovali třídu, byli jsme omezeni datovými typy: použití obecného datového typu Object komplikovalo (místy až znemožňovalo) typovou kontrolu, a deklarace jediného nebo malé množiny specifických datových typů zase omezovalo použitelnost třídy v další aplikaci. Generické třídy a metody byly tedy zavedeny za účelem rozšíření typové kontroly a s cílem využít kód opakovaně s různými datovými typy.

1.1 Co nabízí nového generické programování v jazyku Java?

Generická Java podporuje definice a použití generických typů a metod. Nabízí jazykové vlastnosti pro následující účely:

- *deklarace generického typu* – generický typ je typ s formálními typovými parametry.
- *deklarace parametrizovaného typu* - typ s konkrétními typovými argumenty
- *deklarace generické metody* – jak se deklaruje generická metoda a kdy se využívá
- *typové parametry*-typové parametry se uvádějí v hranatých závorkách za typem, jsou oddělé čárkou.
    - *formální typový parametr* – obecný typový parametr, označuje se velkým písmenem, používá se při deklaraci typu nebo metody
    - *konkrétní typový argument* – typový argument dosazený jako konkrétní datový typ, používá se při vytváření instance generického typu nebo volání generické metody
    - *meze typových parametrů* – hranice typového parametru uvedená klíčovým slovem extends nebo super
- *typové argumenty-zástupce pro typový argument*
    - *žolíky* – označujeme znakem „?“, syntaktický pojem označující množinu typů.
    - *ohraničené žolíky* – existují dva druhy: první, s horní mezí označujeme „? extends X“ a druhý s dolní mezí označujeme „? super X“..
    - *zástupce žolíku (wildcard capture)* – anonymní typová proměnná reprezentující určitý neznámý typ zastupovaný žolíkem. Kompilátor používá zástupce vnitřně pro vyhodnocení výrazu a termín „capture of“ se příležitostně zobrazí v příslušné chybové zprávě
- konkretizace generického typu
    - *syrový typ* – datový typ mimo primitivní typy, který nemá typový parametr .
    - *konkrétní instance* – generický typ parametriozvaný určitým konkrétním argumentem - datovým typem
    - *žolíková instance* – generický typ parametrizovaný žolíkem; není-li zadána žádná mez, považuje se jako přípustný parametr jakýkoliv objekt typu Object.
- konkretizace generické metody
    - *automatické odvození typu* – automatické odvození typu argumentu, které se provede během kompilace
    - *specifikace explicitního typového argumentu* – poskytnutí konkrétního datového typu pro generickou metodu

Účelem této práce bude seznámit čtenáře s těmito rysy a s většinou problémů, se kterými by se mohli potýkat.

1.2 Jaký je primární účel Java genericity?

Genericita Javy byla primárně vyvinuta pro práci s generickými kolekcemi. Ovšem neomezuje se pouze na kolekce, používá se i v dalších třídách: práce s vlákny, vlastními třídami a dalším.

Potřeba generických typů vyvstala primárně z implementace a použití kolekcí, především z těch implementovaných v Java knihovnách. Programátoři chtěli zajistit, že kolekce budou obsahovat pouze určitý typ, například seznam číselných hodnot nebo seznam slov či vět. Kolekce v Javě nenabízely pro prvky stejného typu homogenní kolekce. Všechny kolekce místo toho ukládaly prvky jako typ Object a byly potenciálně heterogenní - což je směs objektů různých typů. Což bylo také viditelné v API pro kolekce: negenerické kolekce přijímaly prvky libovolného typu a vracely jej při výběru z kolekce jako typ Object. (viz. dokumentace pro java.util ve verzi Javy 1.4). V negenerické Javě homogenní kolekce vyžadovaly implementaci různých tříd pro různé typy objektů, třeba IntegerList nebo StringList pro správu čísel a slov.

Samozřejmě psát implementaci každé třídy pro každý datový typ není ani praktické, ani reálné.

Rozumnější cíl bude vytvoření jednotné implementace třídy kolekce pro použití s prvky různých typů. Jinak řečeno, spíše než implementace třídy IntegerList nebo StringList, chceme jednu generickou kolekci List použitelnou v obou případech.

Proto je zde genericita: implementace jedné generické třídy, jež může představovat instance pro různé typy. V knihovnách Javy existuje jedna generická třída List (viz. java.util 5.0). Dovoluje specifikaci instancí List<Integer>, List<String> a dalších, z nichž je každá homogenní kolekcí prvků svého parametru: čísel Integer, slov String a dalších. V generické Javě je generická třída List nazývána generickou třídou, která má svůj typový parametr. Použití jménem List<Integer>, List<String> jsou takzvané parametrizované typy. Jsou instancemi generické třídy, kde je typový parametr nahrazený konkrétním typovým argumentem Integer nebo String.

Použití generických schopností jazyka Java bylo původně motivováno potřebou existence mechanismu pro kontrolu homogenních kolekcí. Ovšem jazykové vlastnosti nejsou omezeny jen na kolekce. Platformní knihovny J2SE 5.0 obsahují mnohé příklady generických typů a metod, jež nemají cokoliv společného s kolekcemi. Příkladem mohou být třídy WeakReferences a SoftReferences v balíčku java.lang.ref, jež jsou speciální typ odkazů na objekty určitého typu reprezentované typovým parametrem. Nebo rozhraní Callable v balíčku java.util.concurrent, reprezentující úlohu a metodu call, která vrací hodnotu určitého typu reprezentovaný parametrem. Dokonce sama třída Class v balíčku java.lang je od verze 5.0 generická třída , jehož typový parametr odkazuje na typ, který jej označuje.

1.3 Jaké jsou výhody užívání Java genericity?

Výhodou použití Java genericity je především možnost detekce dalších chyb při kompilaci.

Použitím parametrizovaného typu třeba LinkedList<String> místo LinkedList umožňuje kompilátoru provést více typových kontrol a vyžaduje méně dynamických přetypování. Tímto způsobem lze dříve nalézt chyby, ve smyslu, že jsou ohlášeny už během kompilace, místo vyvolání výjimky až za běhu programu.

Uvažujme příklad s typem LinkedList<String>. Typ LinkedList<String> vyjadřuje skutečnost, že LinkedList je homogenní seznam prvků typu String. Díky rozsáhlejším informacím provádí kompilátor typovou kontrolu pro zajištění , že LinkedList<String> obsahuje opravdu pouze prvky typu String. Jakýkoliv pokus o vložení cizího prvku je zamítnut a ohlášen chybovou zprávou kompilátoru.

Příklad vytvoření parametrizovaného typu:

```
LinkedList<String> list = new LinkedList<String>();
list.add("abc");      // ok
list.add(new Date());  // chyba
```

Použitím prostého (syrového) typu LinkedList kompilátor by nic nezaznamenal a oba prvky by byly bez varování vloženy. Důvodem je, že neparametrizovaný LinkedList nevyžaduje, aby oba prvky byly stejného typu nebo určitých podtypů. Neparametrizovaný seznam je sekvencí prvků typu Object a tudíž libovolný.
Stejný příklad s použitím neparametrizovaného typu:

```
LinkedList list = new LinkedList();
list.add("abc");      // ok
list.add(new Date());  // zpracuje bez chyby
```

Jelikož je zajištěno, že LinkedList<String> obsahuje pouze řetězce typu String, není nutné přetypovávat z něj obdržené prvky.

Příklad s využitím parametrizovaného typu:

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

S prostým typem LinkedList zde není žádné povědomí o tom, jaký typ prvku skutečně objekt obsahuje. Všechny metody vracejí reference na typ Object, který je nutné přetypovat na požadovaný typ prvku.
Stejný příklad s neparametrizovaným typem:

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

Přetypování skončí výjimkou ClassCastException, pokud obdržený typ není typu String. K stejné chybě za běhu při použití parametrizovaného typu nemůže dojít, protože již kompilátor zabrání vložení cizího prvku.

2.Generické typy

2.1 Co jsou syrové typy (raw types)

Anglicky ‚raw types', typy, které česky nazýváme „syrové typy", nebo také „neozdobené typy", stejně tak jako mohou být nazývané „neparametrizované typy", jsou základní typy, které existovaly v Javě až do verze 1.5. Od této verze, která již ale není nazývána jako Java 1.5, ale jako Java 5.0, jsou zavedeny tzv. generické typy. Co rozumíme pod pojmem „generický typ", se dozvíme v kapitole „Deklarace generických a parametrizovaných typů". . Syrové typy jsou například Collection, List, ArrayList..

Pokud například definujeme

```
ArrayList seznam=new ArrayList();
```

tak zde je ArrayList nazýván „syrovým typem". Tzv. ozdobený, neboli parametrizovaný typ je tedy například :

```
ArrayList<String> seznam=new ArrayList<String>();
```

Toto je ArrayList, který bude schopen pracovat pouze s položkami typu String. O tom, jak lze parametrizovat datové typy se dozvíme v kapitole **Vytváření instancí generických typů.**

Je velmi dobré podotknout, že pokud generický typ v Javě 5 a vyšší deklarujete bez parametrů jako syrový, dostanete varování překladače: „*Note, Your File C:\bluej\parametrickeTypy.java uses unchecked or unsafe operations Note: recompile with –Xlint:unchecked for details.* V následujícím textu bude označován zkráceně názvem ‚varování „unchecked"' , „unchecked warning" nebo jen zkráceně „unchecked".

Příklad:

```
import java.util.*;
public class ParametrickeTypy
{
   ArrayList< Integer > parametrickySeznam=new ArrayList<Integer>();
   ArrayList  syrovySeznam  = new ArrayList();
   public ParametrickeTypy()
   {
      parametrickySeznam.add(158);
      syrovySeznam.add(158); // na tuto řádku budeme upozorněni
   }

}
```

Pokud provedeme rekompilaci pomocí výše zadaného parametru, kompilátor nás upozorní právě na objekt syrovySeznam. Nemůže zde zaručit, že další prvky budou typu Integer, a že nedojde k výjimce při běhu.

2.2 Proč jsou syrové typy povoleny?

Syrové typy jsou povoleny především pro umožnění spolupráce s negenerickým, starším kódem Javy. Pokud máte například negenerickou starou metodu s argumentem typu List, můžete použít parametrizovaný typ List<String>, zaručující, že vrátí seznam instancí typu String.

Příklad použití ‚staršího' (negenerického) kódu s použitím syrových typů:

```
class SomeLegacyClass {
  public void setNames(List c) { &hellip; }
  public List getNames() { &hellip; }
}

final class Test {
  public static void main(String[] args) {
    SomeLegacyClass obj = new SomeLegacyClass();
    List<String> names = new LinkedList<String>();
    &hellip;  fill list &hellip;

    obj.setNames(names);

    names = obj.getNames();    // unchecked warning – varování překladače
  }
}
```

Parametr typu List<String> je předán metodě setNames s argumentem přijímající syrový typ List.
Konverze z List<String> do List je typově bezpečná, jelikož metoda schopná zpracovat jakýkoliv heterogenní seznam objektů umí bez problémů přijmout i seznam objektů typu String. Metoda getNames vrací syrový typ List, který přiřadíme typu List<String>.
Kompilátor zde nemůže zaručit, že výsledný List je opravdu List instancí typu String. i přesto, kompilátor povolí konverzi ze syrového typu List na specifičtější typ List<String>, za účelem spojení generického a negenerického kódu. Protože konverze z List na List<String>, není typově bezpečná, je označena kompilátorem hláškou „unchecked assignment".
Použití syrových typů v psaném kódu po zavedení genericity programovaní Java není doporučováno. S přihlédnutím ke specifikaci jazyka Java, je možné, že budoucí verze Javy zcela zakáží použití syrových typů.

2.3 Deklarace generických a parametrizovaných typů

Generický typ je referenční typ, který má jeden nebo více typových parametrů. Tyto typové parametry jsou nahrazeny typovými argumenty při vytvoření instance generického typu.
Příklady generických typů:

```
interface Collection<E>  {
  public void add (E x);
  public Iterator<E> iterator();
}
```

Rozhraní Collection má jeden typový parametr E. Parametr E je zástupce, který bude při kompilaci nahrazen typovým argumentem při vytvoření instance Collection. Instance generického typu s typovými argumenty se nazývá *parametrizovaný typ*.

```
class MojeTrida<E>{

public mojeTrida<E>(){ private ArrayList mujSeznam<E>=new ArrayList<E>(); }
```

```
public void add(E x) { mujSeznam.add(x); }

}
```

Třída MojeTrida má jediný typový parametr E, který určuje, jak bude parametrizován vnitřní seznam mujSeznam.
Instance toho typu pak může vypadat následovně:

```
MojeTrida<String> seznamSlov=new MojeTrida<String>();
```

2.4 Deklarace generického typu

V předchozích kapitolách jsme se dozvěděli, že generické typy jsou typy, které mají svůj parametr. Jak je tedy můžeme deklarovat? Lze je deklarovat podobným způsobem jako běžné, syrové typy.

V deklaraci generického typu za jménem typu následuje typový parametr, seznam identifikátorů odděluje čárka a tento je ohraničen špičatými závorkami.
Příklad deklarace generického typu:

```
public class Dvojice<X,Y>  {
  private X prvni;
  private Y druha;

  public Dvojice(X a1, Y a2) {
    prvni  = a1;
    druha = a2;
  }
  public X getPrvni()  { return prvni; }
  public Y getDruha() { return druha; }
  public void setPrvni(X arg)  { prvni = arg; }
  public void setDruha(Y arg) { druha = arg; }
}
```

Třída Dvojice má dva typové parametry, X a Y. Jsou nahrazeny typovými argumenty při vytváření instance této třídy. Například v deklaraci Dvojice<String,Date> je typový parametr X nahrazen typovým argumentem String a Y je nahrazen argumentem typu Date.

Rozsah identifikátorů X a Y je celá deklarace této třídy. V tomto rozsahu jsou použity dva typové parametry jako by byly typy. V příkladu výše jsou použity typové parametry jako argumenty a návratové typy metod instance a typů instančních polí.

Typové parametry lze také deklarovat s mezemi. Meze dávají přístup k metodám neznámého typu, který typový parametr zastupuje. V našem případě nevolám žádné metody neznámého typu X a Y. Z tohoto důvodu jsou oba tyto typové parametry bez meze.

2.5 Ohraničení parametrizace generických typů

Téměř všechny referenční typy mohou být generické. To zahrnuje třídy, rozhraní, vnořené statické třídy, vnitřní (nestatické) třídy a místní třídy.
Následující typy nemohou být generické:

Anonymní vnitřní třídy. Anonymní vnitřní třídy mohou implementovat generické rozhraní nebo rozšířit generickou třídu, ale samy nemohou být generické. Generická anonymní třída by totiž neměla smysl: anonymní třídy nemají jméno, ale při deklaraci parametrizovaného typu je pro její instanci třeba uvést její jméno a zadat parametry. Proto zde generické anonymní třídy postrádají smysl.

Třídy výjimek. Generická třída nesmí být odvozena přímo ani nepřímo od třídy Throwable. Generické výjimky nebo chybové typy nejsou dovoleny, jelikož mechanismus práce s výjimkami je runtime mechanismus na úrovni kódu a virtuální stroj Java nezná generické parametry Javy. JVM neumí rozlišit mezi jednotlivými instancemi generické výjimky. Tudíž generické typy výjimek nemají význam. Generické typy se mohou vyskytovat s klauzulí throws (což může být jakýkoliv generický objekt či metoda), ale nikdy už v sekci catch.

Výčtové typy Výčtové typy (typy enum) nemohou mít typové parametry. Jako pojem jsou totiž výčtové typy a jejich hodnoty statické. Jelikož typové parametry nelze použít ve statickém kontextu, parametrizace výčtových typů postrádá jakýkoliv význam.

2.6 Vytváření instancí generických typů.

Instance vytváříme dosazením typových parametrů místo každého typového argumentu.
Při použití generického typu musíme nahradit každý typový parametr uvedený v deklaraci typovým argumentem. Seznam typových argumentů je seznam prvků oddělených čárkou, který je uzavřený do špičatých závorek za jménem typu. Výsledek je takzvaný parametrický typ.

Příklad generického typu:

```
class Dvojice<X,Y>  {
  private X prvni;
  private Y druha;

  public Dvojice(X a1, Y a2) {
    prvni  = a1;
    druha = a2;
  }
  public X getPrvni()  { return prvni; }
  public Y getDruha() { return druha; }
  public void setPrvni(X arg)  { prvni = arg; }
  public void setDruha(Y arg) { druha = arg; }
}
```

Pokud chceme použít generický typ Dvojice, musíme specifikovat typové argumenty, které nahradí paranetry X a Y. Typový argument může být konkrétní referenční typ jako např. String, Long, Date, atd.

Příklad konkrétního parametrizovaného typu :

```
public void vypisDvojici( Dvojice<String,Long> dvojice) {
  System.out.println("("+dvojice.getPrvni()+","+dvojice.getDruha()+")");
}

Dvojice<String,Long> limit = new Dvojice<String,Long> ("maximum",1024L);
vypisDvojici(limit);
```

Instance Dvojice<String,Long> je konkrétní parametrizovaný typ a může být použit jako běžný referenční typ (s několika omezeními, jež si ukážeme dále). V tomto příkladu jsme použili konkrétní parametrizovaný typ jako typový argument metody jako typ referenční proměnné, v novém výrazu pro vytvoření objektu.

Ovšem existují zde i takzvané žolíkové instance. Nemají konkrétní typ jako typové argumenty, ale takzvané žolíky. Žolík je syntaktická konstrukce se znakem „?“ která neznamená jen typ, ale přímo množinu typů. ve své nejjednodušší formě je znakem pro „všechny typy“.
Příklad typu parametrizovaného žolíkem:

```
public void vypisDvojici( Dvojice<?,?> dvojice ){
  System.out.println("("+dvojice.getPrvni()+","+dvojice.getDruha()+")");
}

Dvojice<?,?> limit = new Dvojice<String,Long> ("maximum",1024L);
vypisDvojici(limit);
```

Deklarace Dvojice<?,?> je příklad typu parametrizovaného žolíkem kde oba typy jsou zároveň žolíky. Každý otazník značí oddělenou reprezentaci z rodiny „všech typů“. Výsledná množina instancí se skládá ze všech instancí generického typu Dvojice. (Poznámka: konkrétní typové argumenty z členů množiny nepotřebují být identické, každý z nich je nezávislým typem). Referenční proměnná nebo metoda parametru, jejíž typ je parametrizován žolíkem, může zastupovat jakéhokoliv člena z této množiny.

Je dovoleno vynechat typové argumenty úplně, pokud jsou opa typové argumenty žolíky. Generický typ bez typových argumentů se nazývá „syrový typ“ a je dovolen pouze z důvodů kompatibility se starším, negenerickým Java kódem. Použití syrových typů se obecně jinak nedoporučuje.

2.7 Proč instance generického typu sdílejí stejný runtime typ?

Svůj runtime typ sdílejí kvůli procesu nazývaným „typové čištění“.

Kompilátor překládá generické a parametrizované typy technikou nzývanou „typové čištění“. tento proces primárně odstraňuje veškeré informace vztahující se k typovým parametrům a argumentům. Například, parametrizovaný typ List<String> je přeložen na typ List, což je zde takzvaný „syrový typ“. Toto samé se děje i pro parametrizovaný typ List<Long>, který se v bajtovém kódu objevuje jako List.
Po překladu typovým čištěním zmizí veškeré informace vztahující se k typovým parametrů, a typovým argumentům. ve výsledku, veškeré instance stejného generického typu sdílejí shodný runtime typ, přesněji syrový typ.

Příklad vypisující runtime typ dvou parametrizovaných typů:

```
System.out.println("runtime typ pro ArrayList<String>: "+ne w
ArrayList<String>().getClass());
System.out.println("runtime typ pro ArrayList<Long>  : "+new ArrayList<Long>().g
etClass());

---------------------------------------------------------------------------------------------
vypíše:   runtime typ pro ArrayList<String> : class java.util. ArrayList
     runtime typ pro ArrayList<Long>   : class java.util. ArrayList
```

Příklad ilustruje, že ArrayList<String> a ArrayList<Long> sdílejí stejný runtime typ ArrayList .

3.Parametrické typy

3.1 Konkrétní parametrizovaný typ

Zde je jako parametr uveden jediný konkrétní datový typ, se kterým může třída pracovat skrz parametr:

```
Collection<String> seznam = new LinkedList<String>();

public void vypisDvojici( Dvojice<String,Long> pair) {
  System.out.println("("+dvojice.getPrvni()+","+pair.getDruha()+")");
}

Dvojice<String,Long> limit = new Dvojice<String,Long> ("maximum",1024L);
vypisDvojici(limit);
```

Definice Collection<String> vytváří parametrizovaný typ, který je instancí generického typu Collection, a kde je argument E nahrazen konkrétním typem String. Stejně tak i třída Dvojice, kde jsou žolíky nahrazeny konkrétními parametry.
Tímto se také dostáváme i k parametrizovaným typů, které jsou parametrizovány pomocí žolíků.

3.2 Typy parametrizované pomocí žolíků

Instance generického typu, kde je typový argumentem žolík místo konkrétního typu

Typ parametrizovaný žolíkem je instance generického typu, kde je alespoň jedním typovým argumentem žolík, znak ‚?'. (Podrobnější definici pojmu žolík najdete v kapitole 5.) Příklady typů parametrizovaných žolíky jsou Collection<?> , List<? extends Number> , Comparator<? super String> a Dvojice<String,?>. Typ parametrizovaný žolíkem označuje množinu typů zahrnující konkrétní instance generického typu. Druh použitého žolíku určuje, jaký konkrétní parametrizovaný typ náleží do které množiny. Například, parametrizovaný typ s žolíkem Collection<?> označuje množinu všech instancí rozhraní Collection nezávisle na typovém argumentu. Typ se žolíkem List<? extends Number> zahrnuje množinu všech typů, které jsou potomky třídy Number.
Parametrický typ s žolíkem Comparator<? super String> je množina typů, které jsou instancemi typu Comparator a jsou rodičem typu String.

Parametrický typ se žolíkem není konkrétní typ, který by se mohl objevit s operátorem new. Parametrický typ se žolíkem je podobný typu rozhraní ve smyslu, že lze deklarovat referenční proměnné žolíkem parametrizovaného typu, ale nelze vytvořit instanci parametrizovaného typu žolíkem. Referenční proměnné parametrizovaného typu žolíkem mohou odkazovat na objekt, jež je typu náležící do množiny typů označených parametrizovaným typem žolíkem.
Příklady:

```
Collection<?> kolekce = new ArrayList<String>();
List<? extends Number> seznam = new ArrayList<Long>();
Comparator<? super String> komparator = new RuleBasedCollator("< a< b< c< d");
Dvojice<String,?> dvojice = new Dvojice<String,String>();
```

Nesprávné použití:

```
List<? extends Number> list = new ArrayList<Date>();  // chyba
```

Typ Date není podtypem třídy Number, tudíž ArrayList<Date> nenáleží do množiny typů zahrnovaných typem List<? Extends Number>. Proto také kompilátor ohlásí chybu.

3.3 Práce s parametrizovanými typy

Můžeme provést přetypování typu na parametrizovaný typ?

Ano, ovšem za jistých okolností to není typově bezpečné a kompilátor ohlásí varování hláškou „unchecked warning“.

Všechny instance generického typu sdílejí reprezentaci stejného runtime typu, přesněji reprezentaci svého syrového typu. Například, instance generického typu List, List<String>, List<Long>, atd., mají různé statické typy při kompilaci, ale stejný dynamický typ při běhu, tj typ List.

Přetypování se skládá ze dvou částí:

- kontrola statického typu prováděná kompilátorem během kompilace
- kontrola dynamického typu prováděná virtuálním strojem za běhu programu.

Statická část vytřídí nesmyslná přetypování, které nemohou uspět, jako např. String do Date nebo List<String> do List<Date>.

Dynamická část používá binární typové informace a provádí typovou kontrolu za běhu programu. Vyvolá výjimku ClassCastException, pokud dynamický typ objektu není cílovým typem (nebo podtypem cílového typu) při přetypování. Příklady přetypování s dynamickou částí jsou přetypování z Object na String nebo z Object na List<String>. Toto jsou tzv. přetypování z rodiče na příslušného potomka. Ne všechna přetypování mají dynamickou část. Některá přetypování jsou pouhá statická přetypování a nevyžadují žádnou typovou kontrolu za běhu. Například přetypování mezi primitivními typy, třeba z long na int nebo byte na char.

Další příklad statického přetypování je takzvané nadtypování, z potomku na rodiče, například přetypování ze String na Object nebo z LinkedList<String> na List<String>. Nadtypování (převedení na nadřazený typ) jsou dovolená přetypování, ale nikoliv však povinná. Jsou zde automatické konverze, které kompilátor provádí implicitně, dokonce bez explicitního přetypování, což znamená, že přetypování není nutné a obvykle je vynecháno. Samozřejmě, pokud se nadtypování objeví někde ve zdrojovém kódu, pak je to čistě statické přetypování, které nemá dynamickou část. Přetypování s dynamickou částí jsou potenciálně nebezpečná, pokud je cílovým typem parametrizovaný typ. Runtime informace parametrizovaného typu není přesná, protože všechny instance generického typu sdílí shodou reprezentaci svého typu. Virtuální stroj neumí rozlišit mezi různými instancemi stejného generického typu. Kvůli těmto okolnostem může dynamická část přetypování uspět i v případě, kdy by k tomu dojít nemělo.

Příklad nevhodného přetypování:

```
void m1() {
  List<Date> seznam = new ArrayList<Date>();
  ...
  m2(seznam);
}
void m2(Object arg) {
  ...
  List<String> seznam = (List<String>) arg;    // unchecked warning -  varování
kompilátoru
  ...
  m3(seznam);
  ...
}
```

```
void m3(List<String> list) {
  ...
  String s = list.get(0);      // ClassCastException- došlo k výjimce
  ...
}
```

Přetypování z typu Object na List<String> v metodě m2 se jeví jako přetypování na List<String>, ovšem ve skutečnosti je to přetypování z Object na syrový typ List. Uspěje dokonce i v případě, kdyby to bylo přetypování na List<Date> místo List<String>.
Po tomto úspěšném přetypování máme referenční proměnnou typu List<String>, která odkazuje na objekt typu List<Date>. Pokud si vyžádáme elementy z tohoto seznamu, budeme očekávat String s, ovšem místo nich obdržíme Date s. a objeví se výjimka ClassCastException v místě, kde bychom ji neočekávali.

Jsme připraveni ošetřit výjimku ClassCastException, kde se objevuje výraz přetypování, ale neočekáváme výjimku ClassCastException, když získáváme prvky ze seznamu typu s. Tento druh neočekávané výjimky ClassCastException je považován porušením principů typové bezpečnosti. Aby nás na to kompilátor upozornil, zobrazí varování „unchecked warning“ při překladu tohoto nejistého výrazu přetypování.

Ve výsledku kompilátor ohlásí varování „unchecked warning“ při každém dynamickém přetypování, jehož cílový typ je parametrizovaný typ. Všimněte si, že nadtypování, jehož cílový typ je parametrizovaný typ, nevede k varování typu „unchecked warning“, protože nadtypování nemá dynamickou část.
Poznámka: Při použití parametrizovaných typů s výjimkami není dovoleno definovat generické typy odvozené jak přímo tak nepřímo od třídy Throwable. Tedy žádné parametrizované typy se nevyužívají při zpracování výjimek.

3.4 Statické typy uvnitř generických typů.

Generické typy mohou obsahovat statické členy, včetně statických polí, statických metod a statických vnořených typů typy. Každý z těchto statických objektů existuje jednou pro obalující typ, čili nezávisle na množství objektů obalujícího typu a instancí generického objektu použitého kdekoliv v kódu. Jméno statického členu se skládá – jakožto pro statické členy – z rozsahu (balíčky a obalující typ) a jeho jména. Pokud je obalující typ generický, pak typ v kvalifikaci rozsahu musí být syrový typ a nikoliv parametrizovaný typ.

3.5 Problém dědičnosti a kompatibility parametrů

Rozdílné instance stejného generického typu s různýmy typovými argumenty nemají mezi sebou jakoukoliv příbuznost, a to ani v případě, jsou-li tyto typové argumenty mezi sebou jakkoliv příbuzné.
Mohlo by se očekávat, že například List<Object> by byl rodičem pro List<String>, protože Object je rodičem pro String. Je List<Object> rodičem pro List<String>? Není, různé instance stejného generického typu s různými typovými parametry nemají společný jakýkoliv vztah.

Takové očekávání pramení z faktu, že podobné vztahy existují i pro pole: Object[] je rodičem pro String[], jelikož Object je rodičem pro String. Tento vztah mezi datovými typy se rozšiřuje i na korespondující typy polí. Tyto druhy souvztažnosti však nefungují pro instance generických typů. (parametrizované typy nejsou kovariantní-viz. následující kapitola „Parametrické typy jako součást pole“). Neexistence vztahu mezi typy pro instance generického typu má tedy velmi závažné důsledky následky. Zde je příklad:

```
void vypisVse(ArrayList<Object> c) {
  for (Object o : c)
```

```
        System.out.println(o);
    }

    ArrayList<String> list = new ArrayList<String>();
    ... naplnit seznam ...
    vypisVse(list);   // chyba
```

Objekt ArrayList<String> nelze vložit jako argument k metodě, jež požaduje typ ArrayList<Object>, protože tyto dva jsou instancemi stejného generického typu, ale jelikož jsou s rozdílnými typovými argumenty, nejsou vzájemně zaměnitelné.
Na druhou stranu, instance různých generických typů se stejným parametrem mohou být vzájemně kompatibilní, pokud generické typy mezi sebou tvoří příbuznost.

Příklad:

```
void vypisVse(Collection<Object> c) {
 for (Object o : c)

        System.out.println(o);
    }

List<Object> list = new ArrayList<Object>();
... naplnit seznam ...
vypisVse(list);   // v pořádku
```

Objekt list typu List<Object> je kompatibilním podtypem objektu Collection<Object>, protože oba typy jsou instancemi generického rodiče Collection a mají shodný typový argument Object.

Kompatibilita mezi instancemi stejného generického typu existuje pouze pro instance s žolíkovými parametry a konkrétní instance náležící množině instancí, které žolíková instance zahrnuje.

4.Typový argument

4.1 Definice typového argumentu

Typovým argumentem je referenční typ nebo žolík použitý pro vytvoření instance generického typu nebo referenčního typu použitého pro instanci generické metody. Aktuální typový argument nahrazuje formální typový parametr použitý v deklaraci generického typu nebo metody.

Generické typy a metody mají formální typové parametry, jež jsou nahrazeny aktuálními typovými parametry při instanciování parametrizované metody nebo typu.

Příklad generického typu

```
class Krabicka <T> {
  private T theObject;
  public Krabicka( T arg) { theObject = arg; }
  ...
}
class Test {
  public static void main(String[] args) {
    Krabicka <String> box = new Krabicka <String> ("Certik");
  }
}
```

V tomto příkladě můžeme vidět generickou třídu Krabicka s jedním formálním typovým parametrem T. Tento formální typový parametr je nahrazen aktuálním typovým argumentem String při vytváření objektu typu Krabicka v testovacím programu.

Pro typové argumenty existuje několik pravidel:

- aktuální typové argumenty generického typu mohou být:
  - referenční typy
  - žolíky
  - parametrizované typy (tj., instance jiného generického typu)
- generické metody nelze instanciovat s použitím žolíků coby aktuálních typových argumentů
- typové parametry je dovoleno použít jako aktuální typové argumenty
- primitivní typy nejsou povoleny jako typové argumenty
- typové parametry musí být uvnitř mezí

4.2 Povolené typy typových argumentů

Jsou povoleny všechny referenční typy včetně parametrizovaných typů kromě primitivních typů. Všechny referenční typy mohou být použity jako typové argumenty parametrizovaných typů a metod. Toto zahrnuje třídy, rozhraní, výčtové typy, vnořené a vnitřní typy a pole. Jako typové argumenty nelze použít pouze primitivní typy.

Příklady typů jako typových argumentů parametrizovaného typu:

```
List< int >              l0;     // chyba
List< String >           l1;
List< Runnable >         l2;
List< TimeUnit >         l3;
List< Comparable >       l4;
List< Thread.State >     l5;
List< int[] >            l6;
```

```
List< Object[] >                    l7;
List< Callable<String> >            l8;
List< Comparable<? super Long> >    l9;
List< Class<? extends Number> >     l10;
List< Map.Entry<?,?> >              l11;
```

Ukázka kódu dokazuje, že primitivní typy jako int nejsou dovoleny coby typové argumenty.
Typy tříd jako String, typy rozhraní Runnable, jsou povolenými typovými argumenty.
Výčtové typy jako (viz. java.util.concurrent.TimeUnit) jsou také povolenými typovými argumenty.

Syrové typy jsou též přípustnými typovými argumenty, Comparable je toho příkladem. Thread.State je příkladem vnořeného typu; Thread.State je výčtový typ vnořený do třídy Thread.

Nestatické vnitřní typy jsou též povoleny. Typy pole např. int[] nebo Object[] jsou též přípustné jako typové argumenty parametrizovaného typu nebo metody. Parametrické typy jsou povoleny též jako typové argumenty, zahrnující konkrétní parametrizované typy Callable<String>, s žolíky s mezemi typu Comparable<? super Long> a Class<? extends Number>, a s neohraničenými žolíky, např. Map.Entry<?,?>.
Výše zmíněné typy jsou povoleny jako explicitní typové argumenty generické metody.
Příklady použití typů jako typových argumentů v generické metodě:

```
List<?> list;
list = Collections.< int >emptyList();          // chyba
list = Collections.< String >emptyList();
list = Collections.< Runnable >emptyList();
list = Collections.< TimeUnit >emptyList();
list = Collections.< Comparable >emptyList();
list = Collections.< Thread.State >emptyList();
list = Collections.< int[] >emptyList();
list = Collections.< Object[] >emptyList();
list = Collections.< Callable<String> >emptyList();
list = Collections.< Comparable<? super Long> >emptyList();
list = Collections.< Class<? extends Number> >emptyList();
list = Collections.< Map.Entry<?,?> >emptyList();
```

Ukázka kódu opět říká, že primitivní typy nelze použít jako typový argument v generické metodě.

4.3 Primitivní typy jako typové argumenty

Primitivní typy jako typové argumenty nejsou povoleny, jako typové argumenty lze použít pouze referenční typy. Použití parametrizovaného typu jako List<int> nebo Set<short> je zakázáno. Pouze referenční typy lze použít pro instance generických typů a metod. Místo List<int> musíme deklarovat List<Integer>, s použitím odpovídající obalové třídy do typového argumentu.

Příklad autoboxingu – zabalování (automatické konverze mezi primitivními a jejich obalovými typy)

```
int[] pole = {1,2,3,4,5,6,7,8,9,10};
List<Integer> seznam = new LinkedList<Integer>();
for (int i : pole)
  seznam.add(i);       // autoboxing - zabalení
```

```
for (int i=0;i<seznam.size();i++)
  pole[i] = seznam.get(i);     // auto-unboxing - rozbalení
```

Zde vkládáme prvky primitivního typu int do seznamu s prvky typu Integer díky zabalení, což je automatická konverze z primitivního typu na odpovídající obalový typ. Podobně při extrakci čísel int ze seznamu, dochází k rozbalení, což je automatická konverze z odpovídajícího obalového typu na primitivní.

Všimněte si, že to , že nemožnost použití primitivních typů způsobuje omezení výkonu. Zabalování a rozbalování dělá použití obalujících instancí generických typů velmi pohodlným a zestručňuje zdrojový kód. Jenže zkracující notace zakrývá fakt, že na úrovni kódu virtuální stroj vytváří a používá spoustu obalujících objektů, z nichž každý musí být alokován do paměti a poté uklizen garbage kolektorem. Vyššího výkonu přímým použitím hodnot primitivních typů nelze dosáhnout pomocí generických typů. Pouze běžný typ může nabídnout optimální výkon s použitím hodnot primitivních typů.

Příklad:

```
class Box <T> {
  private T theObject;
  public Box( T arg) { theObject = arg; }
  public T get() { return theObject; }
  ...
}
class BoxOfLong {
  private long theObject;
  public BoxOfLong( long arg) { theObject = arg; }
  public long get() { return theObject; }
  ...
}
class Test {
  public static void main(String[] args) {
    long result;

    Box <Long> box = new Box <Long> (0L);  // zabalení
    result = box.get();           // rozbalení

    Box <long> box = new Box <long> (0L);  // chyba
    result = box.get();

    BoxOfLong box = new BoxOfLong(0L);
    result = box.get();
  }
}
```

Příklad ukazuje, že instance Box pro typ Long vede k nevyhnutelné chybě při zabalení a rozbalení. Instance přes primitivní typ long nepomáhá, jelikož primitivní typ je nepřípustný. Pouze specializovaná třída BoxOfLong eliminuje chybu při použití primitivního typu long.

<u>Obalové typy pro primitnivní typy:</u>
Pro další primitivní typy se používají následující obalové typy:
Byte –zastupuje typ byte
Double – zastupuje typ double
Float – zastupuje typ float

Integer-zastupuje typ int
Long – zastupuje typ long
Short-zastupuje typ short
Number – rodič pro třídy Byte, Double, Float, Integer, Long, Short
Boolean – pro typ boolean
Character – pro typ char

4.4 Žolíky jako typové argumenty

Pro instance generických typů jsou povoleny jako typové argumenty i žolíky, pro instance generických metod už nikoliv. Žolík je syntaktický konstrukt označující rodinu typů, k jehož vyjádření se používá znak ‚?' (více o žolíku viz. kapitola 5 - Žolíky). Všechny žolíky lze použít jako typové argumenty parametrizovaného typu. Toto zahrnuje neohraničené žolíky stejně tak jako žolíky s horní a dolní mezí.
Příklady:

```
List< ? >                l0;
List< ? extends Number >  l1;
List< ? super Long >      l2;
```

Žolíky *nelze* použít jako typové argumenty generických metod.
Příklady:

```
list = Collections.< ? >emptyList();  //chyba
list = Collections.< ? extends Number >emptyList(); //chyba
list = Collections.< ? super Long >emptyList(); //chyba
```

4.5 Typové parametry jako typové argumenty

Jako argumenty parametrizovaných typů nebo metod lze použít opět typové argumenty. Můžeme je stejně jako typové parametry použít do parametrů třídy, konstruktorů i metod. Použití ilustruje příklad instancí generického typu použitím typového parametru jako typového argumentu:

```
class someClass <T> {
  public List<T> someMethod() {
    List< T > list = Collections.< T >emptyList();
    ...
    return list;
  }
  public static <S> void anotherMethod(S arg) {
    List< s > list = Collections.< s >emptyList();
    ...
  }
}
```

Příklad výše ukazuje, jak lze použít parametry T a obalující generickou třídou jako typový argument jak pro parametrizovaný typ jménem list, tak i generickou metodu emptyList.

4.6 Omezující typové parametry pro typové argumenty

V typovém parametru pro omezení množiny typu argumentu lze použít i meze. Je nutné proto vědět, že použitý aktuální typový argument musí být potomkem všech použitých mezí uvedených pro příslušný formální typový parametr.
Příklady (s použitím typů z balíčků java.util a java.lang):

```
class Wrapper<T extends Comparable <T> > implements Comparable
<Wrapper<T>> {
  private final T theObject;
  public Wrapper(T t) { theObject = t; }
  public T getWrapper() { return theObject; }
  public int compareTo(Wrapper <T> other) { return
theObject.compareTo(other.theObject); }
}

  Wrapper <String>          wrapper1 = new Wrapper <String> ("Oystein");
  Wrapper <? extends Number> wrapper2 = new Wrapper <Long> (0L);
  Wrapper <?>               wrapper3 = new Wrapper <Date> (new Date());
  Wrapper <Number>          wrapper4 = new Wrapper <Number> (new Long(0L));  // chyba
  Wrapper <int>             wrapper5 = new Wrapper <int> (5);  // chyba
```

Příklad obsahuje obalující třídu Wrapper, od níž zkoušíme vytvořit různé instance pomocí typových argumentů.
Comparable<T> používá typový parametr jako svůj typový argument.
Comparable<Wrapper<T>> používá instanci parametrizovaného typu jako typový argument.
Wrapper<String> a Wrapper<Long> mají konkrétní referenční typy jako typové argumenty.
Wrapper<? extends Number> a Wrapper<?> používají žolíky jako typové argumenty.
Wrapper<Number> je nepřípustný protože Number není potomek Comparable<Number> a není uvnitř mezí.
Wrapper<int> je nepřípustný, jelikož primitivní typy nejsou dovoleny jako typové argumenty.

4.7 Specifikace typového argumentu při použití generického typu

Při specifikaci generického typu nejsme povinni použít typový parametr, můžeme použít jeho takzvaný syrový typ. Generický typ bez jakéhokoliv typového argumentu se též nazývá syrový typ. Příklady syrových typů jsou List, Set, Comparable, Iterable a další (příklady jsou z java.util a java.lang). Syrové typy jsou povoleny kvůli kompatibilitě mezi generickým a negenerickým rozhraním API Javy. Použití syrových typů po zavedení genericity se v Javě důrazně neporučuje. s přihlédnutím ke specifikaci jazyka Java, je možné, že v budoucích verzích Javy bude použití syrových typů zcela zakázáno.

4.8 Specifikace typového argumentu při volání generické metody

Při volání generické metody nemusíme specifikovat typový argument, tj. smí být použita i bez typových parametrů. Generická metoda může být volána stejně jako běžná metoda, čili bez specifikace typových argumentů. V takovém případě kompilátor odvodí typový argument ze statických typů argumentů metody nebo kontextu volání metody. Tento postup je znám jako odvozování typového argumentu.

5.Žolíky

5.1 Definice žolíku

Žolík je syntaktický konstrukt vyjadřující rodinu (neboli množinu) typů.
Žolík popisuje množinu typů. Existují tři různé typy žolíků:

- " ? " – neohraničený žolík. Vyjadřuje množinu *všech* typů
- " ? extends Type " – žolík s horní mezí. Vyjadřuje množinu všech typů, které jsou podtypem typu Type, zahrnující typ Type..
- " ? super Type " – žolík s dolní mezí. Zastupuje množinu všech typů, které jsou rodičem typu Type, včetně typu Type.

Žolíky se používají k deklarování žolíkových parametrizovaných typů, kde je použit žolík jako argument pro instanci generických typů. Žolíky jsou užitečné v situacích, kde je v parametrizovaném typu stačí pouze částečná nebo žádná znalost o typovém argumentu.

5.2 Neohraničený žolík

Neohraničený žolík je žolík bez zadaných mezí. Znak pro neohraničený žolík je „?“, zastupující množinu *všech* možných typů. Tento neohraničený žolík je použit jako argument pro instance generických typů. Neohraničený žolík je užitečný v situacích, kde není třeba znalost typového argumentu pro parametrizovaný typ.

Příklad:

```
void vypisKolekci( Collection<?> c){   // parametrizovaný typ s neohraničeným
žolíkem
  for (Object o : c){
    System.out.println(o);
  }
}
```

Metoda vypisKolekci nevyžaduje jakékoliv specifické vlastnosti prvků obsažených ve vypisované kolekci. Z tohoto důvodu deklaruje své argumenty s použitím neohraničeného žolíku, říkajíce tím, že je schopna přijmout kolekci s prvky jakéhokoliv typu.

5.3 Ohraničený žolík

Ohraničený žolík je žolík s horní nebo dolní mezí. Žolík s horní mezí je zapsaný jako „? extends Type“ a zastupuje Type a všechny jeho potomky.
Type se nazývá *horní mez*. Žolík s dolní mezí se zapisuje jako „? super Type“ a zastupuje rodinu všech typů, jež jsou rodiči (předky) typu Type včetně typu Type. Type se zde nazývá *dolní mez*. Ohraničené žolíky jsou použity jako argumenty pro instance generických typů. Ohraničený parametrizovaný typ je vhodný tam, kde žolíky bez mezí nabízejí příliš málo informací.

Příklad:

```
public class Collections {
  public static <T> void copy
  ( List<? super T> cil, List<? extends T> zdroj) {  // generické typy s ohraničenými
žolíky
      for (int i=0; i<zdroj.size(); i++)
```

```
      cil.set(i,zdroj.get(i));
  }
}
```

Metoda copy kopíruje prvky ze zdrojového seznamu zdroj do cílového seznamu cil. Cílový seznam musí být schopen obsahovat prvky ze zdrojového seznamu. Toto vyjádříme pomocí ohraničených žolíků: výstupní seznam musí obsahovat prvky s dolní mezí T a vstupní list musí obsahovat s horní mezí T. Prostudujme tento příklad, abychom pochopili typické použití ohraničených žolíků a abychom pochopili, proč nám neohraničené žolíky nestačí. Je to příklad výše zmíněné metody copy, která kopíruje prvky z cílového do zdrojového seznamu. Začněme se základní implementací této metody.
Metoda copy by mohla vypadat například takto:

```
public class Collections {
  public static <T> void copy( List<T> cil, List<T> zdroj) {  // nepoužívá žolíky

     for (int i=0; i<zdroj.size(); i++)
       cil.set(i,zdroj.get(i));
  }
}
```

Tato implementace metody copy je více omezující, než je nezbytně nutné, protože požaduje stejné typy pro vstupní a výstupní seznam. Například, následující volání, ačkoliv naprosto rozumná, by vedla k chybové zprávě (tato zpráva by se objevila až za běhu programu!).

IndexOutOfBoundsException:Source does not fit in dest(in Java.util.Collections)

Příklad neplatného použití metody copy:

```
List<Object> vystup = new ArrayList< Object >();
List<Long>    vstup = new ArrayList< Long >();
...
Collections.copy(vystup,vstup);  // chyba; nesouhlasné typové argumenty
```

Volání metody copy je zamítnuto, protože deklarace vyžaduje, aby oba seznamy obsahovaly prvky stejného typu. Zatímco zdrojový seznam je typu List<Long> a cílový seznam je typu List<Object>, kompilátor zamítne volání metody i přes fakt, že seznam typu List<Object> je schopen přijmout prvky typu Long. Pokud by oba seznamy byly typu List<Object> nebo List<Long>, metoda by byla přijata.

Pokusíme se uvolnit požadavky metody na typové argumenty a deklarovat žolíkem parametrizované typy jako typové parametry metody.
Deklarování parametrizovaného typu žolíkem jako typového parametru metody má výhodu dovolení širší množiny typů argumentu.

Neohraničené žolíky umožňují nejširší dosažitelnou množinu argumentů, jelikož „?“ zastupuje všechny typy bez omezení. Pokusme se tedy použít parametrizovaný typ neohraničeným žolíkem.
Metoda by pak vypadala následovně:
Příklad (rozšířené metody copy; nelze zkompilovat):

```
public class Collections {
  public static void copy( List<?> dest, List<?> src) {  // používá neohraničené žolíky
     for (int i=0; i<src.size(); i++)
       dest.set(i,src.get(i)); // chyba, neplatné typy argumentu
```

```
  }
}
```

Zjistíme, že tuto poněkud rozšířenou metodu nelze zkompilovat. Problém je, že metoda get() pro List<?> vrací odkaz na objekt neznámého typu. Odkazy ukazující na neznámý objekt jsou obvykle vyjadřovány odkazem na typ Object. Tedy metoda get() vrací objekt typu Object. Na druhou stranu, metoda set() vyžaduje cosi neznámého a „neznámý“ neznamená, že požadovaný typ bude typu Object. Tedy argument může být cokoliv odvozené od typu Object, a jelikož toto kompilátor nezná, ohlásí chybu: get() vrací Object a set() požaduje typ specifikovaný přesněji.

V podstatě signatura copy(List<?> dest, List<?> src) říká, že metoda přijímá jeden typ seznamu jako zdroj a kopíruje do zcela nesouvisejícího cílového seznamu. Principielně by to dovolovalo třeba zkopírovat seznam jablek do seznamu hrušek. Což není přesně to, co bychom chtěli.

To, co opravdu chceme, je signatura dovolující kopírování prvků ze zdrojového seznamu do cílového seznamu s určitou vlastností, tedy aby byla schopna přijmout prvky ze zdrojového seznamu. Neohraničené žolíky jsou pro tento účel příliš obecné, jak jsme viděli na příkladu výše.Ovšem ohraničené žolíky jsou vhodné právě pro tento případ.
V našem příkladě metody copy můžeme svého cíle dosáhnout takto:

```
public class Collections {
  public static <T> void copy
  ( List<? super T> dest, List<? extends T> src) {  // používá ohraničené žolíky
      for (int i=0; i<src.size(); i++)
        dest.set(i,src.get(i));
  }
}
```

V této implementaci požadujeme, aby existoval typ T , který je podtypem prvků výstupního seznamu a T by mohl být zároveň rodičem vstupního seznamu. Toto vyjádříme použitím žolíků: výstupní seznam list má mít prvky s dolní mezí T a vstupní seznam musí mít prvky s horní mezí T.
Příklad použití metody copy se žolíky:

```
List<Object> output = new ArrayList< Object >();
List<Long>    input = new ArrayList< Long >();
...
Collections.copy(output,input);  // v pořádku; T:= Number & Serializabe &
Comparable<Number>

List<String> output = new ArrayList< String >();
List<Long>    input = new ArrayList< Long >();
...
Collections.copy(output,input);  // chyba
```

V prvním volání metody T by měl být rodičem typu Long a podtypem Object a dokonce existuje celá množina typů, která spadá do této kategorie: například Number, Serializable a Comparable<Number>. Proto může kompilátor použít jakýkoliv z těchto typů a zpracování metody je povoleno. Druhá nesmyslná metoda je kompilátorem zamítnuta, jelikož si kompilátor uvědomuje, že není typ, který by byl podtypem String a zároveň rodičem typu Long.

Závěr::
Ohraničené žolíky obsahují více informací než neohraničené žolíky. Zatímco neohraničený žolík zastupuje všechny typy, ohraničený žolík reprezentuje množinu rodiče nebo potomků. Tudíž ohraničený žolík obsahuje více typových informací než neohraničený žolík. Nadtyp ohraničeného žolíku se nazývá horní mez, zatímco potomek se nazývá dolní mez.

5.4 Víceúrovňové žolíky

Víceúrovňový žolík je žolík jevící se jako typový parametr uvnitř typového parametru. Příklady objektů typu Collection<E> Collection<Dvojice<String,?>> a Collection<? extends Dvojice<String,?>> jsou příklady využití víceúrovňových žolíků. Víceúrovňové žolíky parametrizovaných typů může být obtížné interpretovat kvůli dědičnosti, protože žolíky se mohou objevit na různých úrovních. pro ilustraci. Zkusme probrat rozdíl mezi Collection<Dvojice<String,?>> a Collection<? extends Dvojice<String,?>> .
Pro zjednodušení ze začátku předpokládejme, že třída Dvojice je typu final.

Typ Collection<Dvojice<String,?>> je konkrétní instancí generického rozhraní Collection. Je to heterogenní kolekce dvojic různých typů. Může obsahovat prvky typu Dvojice<String,Long>, Dvojice<String,Date>, Dvojice<String,Object>, Dvojice<String,String> a mnoho dalších. Jinak řečeno, Collection<Dvojice<String,?>> obsahuje směs dvojic splňující formu Dvojice<String,?>.

Typ Collection<? extends Dvojice<String,?>> je parametrizovaný typ s žolíkem, nepředstavuje konkrétní parametrizovaný typ. Je reprezentativním typem pro rodinu typů kolekcí, jež jsou instancemi rozhraní Collection, kde typový argument je typu Dvojice<String,?>.
Kompatibilní instance jsou Collection<Dvojice<String,Long>>, Collection<Dvojice<String,String>>, Collection<Dvojice<String,Object>> , nebo Collection<Dvojice<String,?>> . Jinak řečeno, nevíme jaký typ instance Collection zastupuje.
Je třeba vědět, že víceúrovňové žolíky je třeba dle pravidla palce číst shora dolů

Type<DruhyArg<NejvyssiArg>>

Pokud je typový argument první úrovně konkrétní typ, pak instance je konkrétního typu, nebo je smíšený typ něčeho, pokud obsahuje žolík hlouběji na nižší úrovni. Takto typ Collection<Dvojice<String,?>> je kolekcí dvojic určité formy se spoustou potomků. Je podobná ke Collection<Object>, která je kolekcí určitého typu mající spoustu potomků a je směsí čehokoliv co je podtypem typu Object.
Pokud je na nejvyšší úrovni žolíkový argument, pak typ není konkrétní. Je zástupce určité rodiny prvků. Tedy Collection<? extends Dvojice<String,?>> je zástupce pro kolekci instancovanou pro částečně neznámý typ určité formy. Je podobný ke Collection<?>, což je zástupce pro specifickou instanci rozhraní Collection, ale není konkrétním typem.

Zde je příklad ukazující rozdíl mezi Collection<Dvojice<String,?>> a Collection<? extends Dvojice<String,?>> .

Příklad

```
Collection< Dvojice<String,Long> >       c1 = new
ArrayList<Dvojice<String,Long>>();

Collection< Dvojice<String,Long> >       c2 = c1;  // ok
Collection< Dvojice<String,?> >          c3 = c1;  // chyba
Collection< ? extends Dvojice<String,?> > c4 = c1; // ok
```

Samozřejmě můžeme přiřadit Collection<Dvojice<String,Long>> ke Collection<Dvojice<String,Long>>, na tom není nic překvapivého.

Ale nemůžeme přiřadit Collection<Dvojice<String,Long>> ke Collection<Dvojice<String,?>> . Parametrický typ Collection<Dvojice<String,Long>> je homogenní kolekce String a Long ; parametrizovaný typ Collection<Dvojice<String,?>> je heterogenní kolekcí dvojic typu String a neznámého typu. Heterogenní kolekce Collection<Dvojice<String,?>> může například obsahovat Dvojice<String, Date> a zcela jistě nenáleží ke Collection<Dvojice<String,Long>>. Z tohoto důvodu nelze přiřazení povolit.

Ale můžeme přiřadit Collection<Dvojice<String,Long>> ke Collection<? extends Dvojice<String,?>> protože typ Dvojice<String,Long> náleží do množiny označené žolíkem ? extends Dvojice<String,?>. Protože typová množina označená ? extends Dvojice<String,?> zahrnuje všechny potomky Dvojice<String,?>.

Protože předpokládáme, že Dvojice je typu final (nelze z něj dědit další třídu), tato typová množina obsahuje všechny instance generického typu Dvojice, kde je prvním argumentem String a druhý je zcela libovolný typ nebo žolík. Množina typů zahrnuje členy jako Dvojice<String,Long> , Dvojice<String,Object> , Dvojice<String,? extends Number> , a samotný Dvojice<String,?>.

Pokud se vzdáme zjednodušení, že Dvojice je finální třídou, pak musíme uvažovat i potomky typu Dvojice .

Collection<Dvojice<String,?>> je heterogenní kolekcí dvojic různých typů formy Dvojice<String,?> anebo jejich potomků. Obsahuje prvky typu Dvojice<String,Long>, Dvojice<String,Date> , ale také prvky typu PotomekDvojice<String,Date>, PotomekDvojice<String,Object> , a podobné další.

Collection<? extends Dvojice<String,?>> zastupuje rodinu kolekcí instancí rozhraní Collection, kde jsou typové argumenty Dvojice<String,?> , nebo jeho potomky. Kompatibilní parametrizované typy jsou Collection<Dvojice<String,Long>> , Collection<Dvojice<String,Object>>, ale také Collection<PotomekDvojice<String,Object>>, nebo Collection<PotomekDvojice<String,?>> .

Zde je příklad ukazující rozdíl mezi konkrétním parametrizovaným typem Collection<Dvojice<String,?>> a parametrizovaným typem s žolíkem Collection<? extends Dvojice<String,?>> .

Příklad:

```
Collection< PotomekDvojice<String,Long> > c1 = new
ArrayList<PotomekDvojice<String,Long>>();

Collection< Dvojice<String,Long> >        c2 = c1;  // chyba
Collection< PotomekDvojice<String,Long> > c3 = c1; // ok
Collection< Dvojice<String,?> >           c4 = c1;  // chyba
Collection< ? extends Dvojice<String,?> >  c5 = c1;  // ok
```

V tomto případě *nemůžeme* přiřadit Collection<PotomekDvojice<String,Long>>
ke Collection<Dvojice<String,Long>> , protože tyto dvě instance generické Collection jsou nepříbuzné a neslučitelné typy.
Collection<PotomekDvojice<String,Long>> obsahuje PotomekDvojice<String,Long> objekty, zatímco Collection<Dvojice<String,Long>> obsahuje mix objektů typu PotomekDvojice<String,Long>. Toto zahrnuje, ale neomezuje se pouze na objekty PotomekDvojice<String,Long>. Proto Collection<PotomekDvojice<String,Long>> nemůže být přiřazena ke Collection<Dvojice<String,Long>> .
Také nemůžeme přiřadit Collection<PotomekDvojice<String,Long>> ke Collection<Dvojice<String,?>>, protože parametrizovaný typ Collection<PotomekDvojice<String,Long>> je homogenní kolekcí objektů PotomekDvojice<String,Long> , zatímco parametrizovaný typ Collection<Dvojice<String,?>> je heterogenní kolekcí.Heterogenní Collection<Dvojice<String,?>> může obsahovat Dvojice<String,Date> a proto zcela jistě nenáleží ke Collection<PotomekDvojice<String,Long>> .

Přiřazení Collection<PotomekDvojice<String,Long>> ke Collection<? extends Dvojice<String,?>> je v pořádku, protože typ PotomekDvojice<String,Long> náleží do množiny typů označené žolíkem ? extends Dvojice<String,?> zahrnuje všechny potomky pro Dvojice<String,?> . Protože již nepředpokládáme, že Dvojice je finálním typem, tato množina typů je String a druhý typový argument je libovolný nebo žolík. Množina typů zahrnuje členy jako Dvojice<String,Long> , Dvojice<String,Object> , Dvojice<String,? extends Number> a Dvojice<String,?> samotný, stejně jako PotomekDvojice<String,Long> , PotomekDvojice<String,Object> , PotomekDvojice<String,? extends Number> , a PotomekDvojice<String,?> .

5.5 Vícenásobný žolík

Je pravda, že zastupuje vícenásobný žolík v programu pokaždé ten samý typ? Každý žolík nezastupuje ve stejném typu stejný konkrétní typ. Každý žolík naopak může zastupovat cokoliv jiného.

Pokud je více žolíků v sekci typového argumentu, každé jeho použití vždy odkazuje na potenciálně rozdílný typ. Je to podobné žolíkům v regulárním výrazu: v „s??““ žolík nezastupuje stejný znak. Může to být „sob“, „sok“ „suk“, nebo „srp“, všechna vyhovují tomuto regulárnímu výrazu. Každý symbol otazníku zastupuje potenciálně jiný znak, a to samé i pro žolíky v generické Javě.
Příklady opakovaného použití žolíku:

```
Dvojice< ? , ? > dvojice = new Dvojice< String , String >("Orpheus","Eurydike");
Dvojice< ? , ? > vanoce   = new Dvojice< String , Date >("Vánoce", new
Date(104,11,24));
```

V příkladu výše, žolík „?“ může zastupovat stejný typ jako String, ovšem vůbec nemusí. Každý žolík může označovat úplně jiný typ, třeba String a Date o řádku dole, což je samozřejmě syntakticky také správně.
Naopak, různé žolíky nemusí zastupovat různé typy. Pokud se typové množiny označené dvěma žolíky překrývají, pak mohou oba žolíky označovat ten samý typ.
Příklad s použitím různých žolíků:

```
Dvojice< ? extends Appendable , ? extends CharSequence > textPlusSuffix
  = new Dvojice< StringBuilder , String >(new StringBuilder("log"), ".txt");
```

```
Dvojice< ? extends Appendable , ? extends CharSequence > textPlusText
  = new Dvojice< StringBuilder , StringBuilder >(new StringBuilder("log"), new
StringBuilder(".txt"));
```

V příkladu výše můžeme vidět, že různé žolíky "? extends Appendable" a "? extends CharSequence" mohou zastupovat různé typy StringBuilder a String, ovšem stejně tak mohou být stejným typem StringBuilder, jelikož nám to dovoluje dědičnost a obojí rozhraní jsou jeho nadtřídou.

Dále je několik příkladů s opakujícím se žolíkem, kde nám kompilátor zobrazí chybovou zprávu. V prvním příkladě se objevuje v parametru 2x, konkrétně v třídě Dvojice<?,?> .

Příklad č.1 zobrazující nekompatibilitu žolíků:

```
class Dvojice<S,T> {
  private s prvnit;
  private T druhy;
  public Dvojice(S s,T t) { prvni = s; druhy = t; }
  ...
  public static void flip( Dvojice< ? , ? > pair) {
  Object tmp = pair.prvni;
  pair.prvni = pair.druhy;  // chyba: neslučitelné typy
  pair.druhy = tmp;   // chyba: neslučitelné typy
  }
}
class Test {
  public static void test() {
    Dvojice<?,?> vanoce = new Dvojice< String , Date >("Vanoce",new Date(104,11,24));
    Dvojice.flip(vanoce);
   Dvojice <?,?> jmeno = new Dvojice< String , String >("Pavel","Riha");
    Dvojice.flip(jmeno);
  }
}
```

Parametry pro Dvojice <?,?> mohou být rozdílnými typy. V případě, je - li zavolána metoda flip s argumentem typu Dvojice<String,Date> , první by byl typu String a druhý je typu Date, a obojí nelze k sobě přiřadit. Dokonce i když je metoda flip zavolána s argumentem Dvojice<String,String>, kompilátor nezná implementaci metody flip. Proto kompilátor zobrazí chybovou zprávu v implementaci metody, protože oba neznámé, potenciálně rozdílné, mohou být k sobě navzájem přiřazeni. ve druhém případě se žolíky objevují v instancích stejného generického typu, konkrétně Collection<?>.

Příklad č.2 ukazující vzájemnou nekompatibilitu žolíku:

```
class Utilities {
   public static void add( Collection< ? > list1, Collection< ? > list2) {
     for (Object o : list1)
       list2.add(o);    // chyba: neslučitelné typy
   }
}
class Test {
  public static void test() {
   Collection<?> dates   = new LinkedList< Date >();
   Collection<?> strings = new LinkedList< String >();
   add(dates,strings);
   add(strings,strings);
  }
}
```

Obě kolekce obsahují prvky dvou potenciálně rozdílných typů. Kompilátor nepovolí vložení prvků z jedné kolekce do druhé, protože se může jednat potenciálně o dva různé typy.
Ve třetím příkladě se objevují v Collection<? extends CharSequence>
a Class<? extends CharSequence> .

Příklad č.3 ukazující vzájemnou nekompatibilitu žolíků:

```
class Utilities {
  public static void method( Collection< ? extends CharSequence > coll,
                             Class     < ? extends CharSequence > type) {
    ...
    coll.add(type.newInstance()); // chyba: neslučitelné typy
    ...
  }
}
class Test {
  public static void test() {
    List< StringBuilder > stringList = new LinkedList<StringBuilder>();
    method(stringList, String .class);
  }
}
```

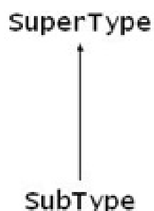
Metoda newInstance třídy Class<? extends CharSequence> vytváří objekt neznámého potomka CharSequence , zatímco kolekce Collection<? extends CharSequence> obsahuje prvky potenciálně různých potomků CharSequence. Kompilátor ohlásí chybu, pokud vložíme do kolekce nový objekt, který je neslučitelným typem s parametrem kolekce.

6. Meze žolíků

6.1 Mez žolíků

Mez žolíků je referenční typ použitý k bližšímu popsání množiny typů vyznačené žolíkem.

Žolík může být neohraničený, přičemž je pak označen znakem ‚?‘. Připomeňme si, že neohraničený žolík zastupuje množinu *všech* typů. Případně žolík může mít i svou mez. Existují dva druhy mezí: horní a dolní mez. Syntax pro ohraničený žolík je „? extends SuperType“ (žolík s horní mezí) a „? super SubType“ (pro žolík s dolní mezí). Termíny horní a dolní mez vycházejí ze způsobu, jakým se vyjadřují dědičné vztahy mezi typy.



Mez zužuje a upřesňuje rodinu typů, kterou zastupuje žolík. Například žolík „? extends Number“ zastupuje množinu typů, které jsou podtypy typu Number včetně typu Number. Žolík „? super Long“ zastupuje typ Long a veškeré jeho předky. Je třeba vědět, že každý žolík může mít pouze *nejvýše jedn*u mez. Nemůže mít tedy jak horní, tak i dolní mez a konstrukce typu „? super Long extends Number“ nebo " ? extends Comparable<String> & Cloneable " nejsou povoleny.

6.2 Povolené meze žolíků

Povolené jsou všechny referenční typy včetně typů parametrizovaných, s výjimkou primitivních typů. Jako meze můžeme použít jakékoliv třídy, rozhraní, výčtové typy, vnořené a vnitřní typy a typy polí. Jediné, co nesmíme použít, jsou primitivní typy a místo nich musíme použít zapouzdřující typy (Integer, Long, Number, Boolean…).

Příklady výčtových typů:

```
//List<? extends int>  list0;        // primitivní datový typ - chyba
List<? extends Integer> list1;       //zapouzdřující primitivní datový typ
List<? extends String> list2;      // třída String
List<? extends Runnable> l3;    // rozhraní Runnable
List<? extends TimeUnit>  l4;    // výčtový typ
List<? extends Comparable> list5;        // rozhraní syrový typ
List<? extends Thread.State> list6;        // vnořený typ
List<? extends int[]>  list7;        // typ pole primitivního typu
List<? extends Object[]> list8;    //typ pole lib. objektu
List<? extends Callable<String>> list9;           //parametrizovaný typ
List<? extends Comparable<? super Long>> list10;  // parametrické rozhraní s žolíkem
List<? extends Class<? extends Number>>  l11;  // parametrizovaná třída s žolíkem
List<? extends Map.Entry<?,?>> list12;  // vnořený generický typ
List<? extends Enum<?>> list13;    // parametrizovaný výčet
```

Příklad ukazuje pouze několik referenčních typů jako horní meze žolíku, ovšem tyto typy lze stejně snadno použít i jako dolní mez. Můžeme také vidět, že primitivní typy nejsou povolené jako mez žolíku. (toto již bylo řečeno v kapitole 4.6 „Povolené typy typových argumentů“). Použití třídy typu např. String stejně jako rozhraní Runnable jsou přípustné jako meze žolíků. Výčtové typy, např. TimeUnit jsou též povolenými. Dokonce i třídy bez potomků jako final třídy lze použít jako mez. Výsledná množina je pak samozřejmě jen jeden prvek. Například mez „? extends String“ se skládá pouze ze samotného prvku String. Dle stejné logiky můžeme dokonce napsat i „? super Object“, ačkoliv Object již žádnou nadřazenou třídu nemá, a toto bychom využili jen ve speciálních případech. Výsledný typ je pak již jen samotný Object. Můžeme použít i syrové typy; viz. seznam list5 používá syrový typ Comparable. Thread.State je příklad vnořeného typu; Thread.State je statický výčtový typ vnořený do třídy Thread. Nestatické vnitřní typy jsou také dovolené typy.

Typy polí viz. int[] a Object[] jsou povolenými mezemi žolíků. Žolíky s typem pole jako mezí označují množinu pod- nebo nadtypů žolíku. Například " ? extends Object[] " je množina všech typů polí, jehož typ prvku je referenčním typem. Typ int[] nenáleží do této množiny, ovšem Integer[] ano. Podobně „? super Number[]“ je množina předků tohoto typu, což je nejen Object[], ale i Object, Cloneable a Serializable. Parametrické typy jsou povolené meze, zahrnující konkrétní parametrizované typy jako Callable<String>, parametrizované typy s ohraničeným žolíkem jako Comparable<? super Long> a Class<? extends Number> , a parametrizovaným typem s neohraničeným žolíkem jako Map.Entry<?,?> . Dokonce i prvotní nadtyp všech výčtových typů, třídu Enum, lze použít jako mez žolíku.

6.3 Rozdíly mezi mezí žolíku a mezí typového parametru

Žolík může mít pouze jednou mez, zatímco typový parametr jich může mít více. Žolík může mít dolní nebo horní mez, zatímco pro typový parametr dolní mez neexistuje.
Meze žolíků a typových parametrů jsou často zaměňovány, jelikož obojí jsou nazývány mezemi a mají částečně stejný syntax.
Příklad meze typového parametru a meze žolíku:

```
class Box< T extends Comparable<T> & Cloneable >
  implements Comparable<Box<T>>, Cloneable {

  private T theObject;
  public Box(T arg)              { theObject = arg; }
  public Box(Box< ? extends T > box) { theObject = box.theObject; }
  ...
  public int compareTo(Box<T> other) { ... }
  public Box<T> clone()           { ... }
}
```

Ukázka kódu výše ukazuje typový parametr T s dvěma mezemi, konkrétně Comparable<T> a Cloneable , a žolík s horní mezí T.

Meze typových parametrů dávají přístup k využití nestatických metod. V příkladu výše, mez Comparable <T> umožňuje zavolání metody compareTo s parametrem typu T. Jinak řečeno, kompilátor je schopen přijmout výraz typu theObject.compareTo(other.theObject).
Mez žolíku popisuje rodinu typů, jež zastupuje. Jak lze vidět v příkladu, žolík "? extends T " popisuje množinu všech podtypů typu T. Je použit v typovém argumentu konstruktoru a dovoluje vložení objektů typu Box a prvků z množiny Box<? extends T> do parametru konstruktoru. Dovoluje například vytvoření Box<Number> z objektu Box<Long>.
Syntax je podobný, ale existují zde drobné rozdíly:

	Syntax
mez typového parametru	TypeParameter extends Class & Interface $_1$ & ... & Interface $_N$
meze žolíků:	
horní mez **dolní mez**	? extends SuperType ? super SubType

Žolík může mít pouze jednu mez, buď horní nebo dolní. Seznam mezí žolíka není dovolen. Typový parametr naopak může mít několik mezí, ovšem neexistuje zde nic jako dolní mez.

7.Generické metody

7.1 Co je to generická metoda

Generická metoda je metoda s typovými parametry. Generické mohou být tedy nejen typy, ale i metody. Statické i nestatické metody stejně jako konstruktory mohou mít typové parametry. Syntax deklarace generické metody je podobný deklaraci generického typu. Sekce typového parametru je ohraničena špičatými závorkami a je před návratovým typem metody. Jeho syntax a význam je identický jako typový parametr objektu generického typu.
Zde je příklad generické metody max, schopné najít nejvyšší hodnotu v kolekci prvků neznámého typu A.

Příklad generické metody:

```
class Collections {
  public static <A extends Comparable<A>> A max(Collection<A> xs) {
    Iterator<A> xi = xs.iterator();
    A w = xi.next();
    while (xi.hasNext()) {
      A x = xi.next();
      if (w.compareTo(x) < 0) w = x;
    }
    return w;
  }
}
```

Metoda max má jeden typový parametr se jménem A. Tento parametr je zástupcem pro prvky kolekce, s nimiž pracuje tato metoda. Typový parametr má horní mez Comparable<A>; musí být typem, který je zároveň potomkem třídy Comparable<A> .

7.2 Jak použít generickou metodu

Generickou metodu použijeme stejně jako negenerickou jejím zavoláním. Potřebné typové argumenty nejsou nutné zadávat explicitně, jsou odvozeny automaticky při kompilaci. Typové parametry jsou odvozeny z kontextu použití metody.

Příklad použití generické metody; viz. předchozí příklad.

```
class Collections {
public static <A extends Comparable<A>> A max (Collection<A> xs) {
    Iterator<A> xi = xs.iterator();
    A w = xi. next();
    while (xi.hasNext()) {
      A x = xi.next();
      if (w.compareTo(x) < 0) w = x;
    }
    return w;
  }
}
final class Test {
  public static void main (String[ ] args) {
    LinkedList<Long> list = new LinkedList<Long>();
    list.add(0L);
    list.add(1L);
    Long y = Collections.max(list) ;
```

```
  }
}
```

V našem případě kompilátor automaticky zavolá instanci metody max s typovým argumentem Long, což znamená, že formální typový parametr je nahrazen typem Long. Všimněte si, že nemusíme explicitně zadat typový argument. Kompilátor automaticky odvodí vlastní typový argument od typových argumentů zadaných při volání metody. Kompilátor zjistí, že Collection<A> je žádán a LinkedList<Long> je k dispozici. Od této informace kompilátor usoudí, že A má být nahrazena typem Long.

7.3 Volání generických metod

V doprovodném příkladu si všimněte, že na rozdíl od deklarací tříd a volání konstruktorů jsme při vyvolávání metody nezadávali žádné typy typových parametrů. Z typu parametru metody si totiž překladač většinou odvodí dostatek informací pro to, aby správný typový parametr přiřadil sám.

Jsou ale situace, kdy to překladač odhadnout nedokáže, nebo když máme nějaké speciální požadavky a chceme, aby metoda pracovala s jiným datovým typem typového parametru. Pak musíme typový parametr zadat.

Typy typových parametrů se při volání metod zadávají do špičatých závorek před identifikátor volané metody, avšak za tečku následující za její kvalifikací. Metodám volaným bez kvalifikace proto není možno typové parametry zadat. Vše ukazuje následující příklad:

```
public class Metody
{
        public static<T> List<T> dvojice( T prvek ){
                List<T> ret = new ArrayList<T>();
                ret.add( prvek);
                ret.add(prvek);
                return ret;
}

public void volání() {
                Metody e = new Metody();
                List<String> ls = dvojice( „Text“ );
                        List<Object> lo;
                //          lo = dvojice( “Text);  //Nesouhlasí typy - nelze
                //          lo = <Object>dvojice(„objekt“); //Nekvalifikované volání - nelze
                        lo = Metody.<Object> dvojice (“Kvalifikace třídou“);
                        lo=m.<Object>dvojice( “Kvalifikace instancí“);
                        lo = dvojice( (Object)“Řešení přetypováním“ );
}
}
```

V předchozích případech byl parametrem metody dvojice vždy textový řetězec. Pokud jsme proto typ parametru nezadali explicitně, překladač dosadil typovému parametru hodnotu String a vracel návratovou hodnotu typu List<String>.

Jak si vysvětlíme za chvíli, parametrizované typy s různými datovými typy svých typových parametrů jsou neslučitelné. Do proměnné typu List<Object> proto není možné přiřadit odkaz na instanci typu List<String>. Proto je zakomentovaný první příkaz, který se o to pokouší, jinak jej překladač označí za syntakticky chybný.

Musíme proto zařídit, aby metoda dvojice vracela potřebný typ — toho dosáhneme explicitním zadáním typových parametrů. Jak jsme si ale řekli, můžeme je zadat pouze při kvalifikovaném volání. Proto je příkaz po zakomentovaném příkazu s nekvalifikovaným voláním také zakomentován, jelikož i on je syntakticky chybný.

V pořádku jsou až následující dva příkazy na řádcích 18 a 19. (po druhém zakomentovaném příkazu) První kvalifikuje volání metody dvojice třídou, druhý instancí této třídy (jak víme, to je u metod třídy možné, i když to není z hlediska čistoty programu doporučované). Poslední příkaz ukazuje alternativní možnost řešení daného problému: vhodné přetypování parametru, z jehož typu překladač odvozuje datový typ typového parametru.
Lze ale spíše doporučit explicitní uvádění typových parametrů, protože takto sdělíte své požadavky mnohem průhledněji.
Samozřejmě při volání a použití parametrizovaných metod můžeme narazit na další problémy.

7.4 Přetěžování generických metod

Proč nefunguje překrývání metod vždy tak, jak bychom očekávali?
Důvodem je jen jednobytové vyjádření každého generického typu a nebo metody - pokud zavoláte přetíženou metodu a vložíte argument, jehož typ je typová proměnná nebo zahrnuje typovou proměnnu, můžete obdržet překvapivé výsledky. Podívejme se na následující příklad:

Příklad (volání přetížené metody):

```
static void pretizenaMetoda( Object o) {
      System.out.println("pretizenaMetoda (Object) zavolána");
}
static void pretizenaMetoda ( String s) {
      System.out.println("pretizenaMetoda (String)  zavolána ");
}
static void pretizenaMetoda ( Integer i) {
      System.out.println("pretizenaMetoda (Integer)  zavolána ");
}
static <T> void generickaMetoda(T t) {
     pretizenaMetoda (t) ;  // která metoda bude zavolána?
}
public static void main(String[] args) {
      generickaMetoda( "abc" );
}
```

Máme zde několik verzí metody. Přetížená metoda je zavolána generickou metodou, která obsahuje argument T pro přetíženou metodu. Případně generická metoda je zavolána a řetězec je vložen jako argument do generické metody. Někdo by mohl očekávat, že uvnitř generické metody bude zavolána metoda s argumentem typu String. Což ale není správně.
Program vypíše:

```
pretizenaMetoda( Object ) zavolána
```

Jak k tomu ale došlo? Vždyť jsme vložili argument typu String do přetížené metody a místo ní je zavolána metoda s parametrem Object. Důvod je ten, že kompilátor vytváří pouze jednobytovou reprezentaci na generickou metodu nebo třídu a mapuje všechny iniciace generického typu nebo metody pouze na jednu tuto reprezentaci.
V našem příkladě je metoda převedena na následující reprezentaci:

```
void generickaMetoda( Object t) {
      pretizenaMetoda (t) ;
}
```

S přihlédnutím k překladu by mělo být jasné, proč je zavolána verze metody s parametrem Object. Je zcela nepodstatné, jaký typ objektu je zadán do generické metody a poté předán přetížené metodě. Voláním jakékoliv metody obdržíme z přetížených metod tu s parametrem Object, a to v důsledku typového čištění.

Obecněji řečeno: řešení přetížení metody se děje v okamžiku kompilace, čili kompilátor zde rozhoduje, která z metod bude zavolána. Kompilátor tak činí v okamžiku, kdy je generická metoda překládána do reprezentace unikátního bajtového kódu. Během překladu je provedeno typové čištění, což znamená, že typové parametry jsou nahrazeny nejbližší horní mezí nebo typem Object, pokud není žádná horní mez zadána. Následně, horní mez nebo Object určují, která verze přetížené metody bude zavolána. Jaký typ objektu je vložen metodě, je zcela nepodstatné pro řešení přetížení.

Zde je ještě jeden příklad, o něco složitější hádanka:

Příklad (volání přetížené metody):

```
public final class GenericClass<T> {
   private void pretizenaMetoda( Collection<?> o) {
      System.out.println(" pretizenaMetoda (Collection<?>)");
   }
   private void pretizenaMetoda( List<Number> s) {
      System.out.println("pretizenaMetoda (List<Number>)");
   }
   private void pretizenaMetoda( ArrayList<Integer> i) {
      System.out.println("pretizenaMetoda (ArrayList<Integer>)");
   }

   private void method(List<T> t) {
pretizenaMetoda(t);  // která metoda bude zavolána?
   }

   public static void main(String[] args) {
GenericClass <Integer> test = new  GenericClass < Integer >();
      test.method( new  ArrayList< Integer > () );
   }
}
```

program vypíše:

```
pretizenaMetoda (Collection<?>)
```

Očekávali bychom, že bude zavolána verze metody s ArrayList<Integer>, ovšem i toto je chybné očekávání. Podívejme se nyní, do jakého tvaru kompilátor přeloží tuto generickou třídu.

Příklad (po typovém očištění):

```
public final class GenericClass {
   private void pretizenaMetoda ( Collection o) {
      System.out.println("pretizenaMetoda (Collection<?>)");
```

```
    }
    private void pretizenaMetoda ( List s) {
        System.out.println("pretizenaMetoda (List<Number>)");
    }
    private void pretizenaMetoda ( ArrayList i) {
        System.out.println("pretizenaMetoda (ArrayList<Integer>)");
    }

    private void method(List t) {
pretizenaMetoda (t);
    }

    public static void main(String[] args) {
GenericClass test = new  GenericClass ();
        test.method( new  ArrayList () );
    }
}
```

Mohli bychom předpokládat rozhodnutí kompilátoru, že verze List přetížené metody bude představovat nejlepší shodu. Což by ale bylo opět špatně. Verze List přetížené metody byla původně verzí, jež má List<Number> jako argument, ovšem při volání, kde je vložen jako argument List<T>, kdy T může být jakýmkoliv typem, nejen Number. Protože T může být jakýmkoliv typem, jediná přípustná verze metody je verze s argumentem Collection<?> .

Závěr:
Vyhněte se typovým proměnným v přetížených metodách. Anebo, přesněji, buďte opatrní, pokud vkládáte argument do přetížené metody, jejíž typ je typová proměnná nebo zahrnuje typovou proměnnou, a proveďte pro jistotu i test dle výše zmíněného postupu.

8.Parametrizace a očišťování

8.1 Proces překladu

Většinou není nutné znát, jak překladač převádí zdrojový kód do bajt kódu. Ovšem v případě generického programování je pochopení základního principu nutné. Může se totiž stát, že budete muset pracovat i se zpětně přeloženým kódem.
Než se pustíme do vysvětlování, připomeňme si ještě jednou terminologii: termínem parametrizovaný typ (někdy také nazývaný „ozdobený datový typ“) rozumíme datové typy doplněné o typové argumenty. Například se může jednat o rozhraní List<String> nebo třída LinkedList<String>.

Proces očišťování parametrizovaných typů známe též pod názvem „typové čištění“, stejně tak jako pod anglickým názvem „type erasure“. Ačkoliv jsme se s tímto termínem již několikrát setkali, bude dobré jej zde popsat konečně vcelku a podrobněji. Typovým čištěním rozumíme proces, kdy kompilátor během překladu nahrazuje generické typy a metody syrovými typy a metody , tj. odstraňuje informace o parametrizaci objektů a metod. Kompilátor při tomto doplňuje potřebná přetypování a provádí typovou kontrolu, a nahrazuje parametry vhodnými datovými typy. Nahrazování ozdobených typů jejich syrovými typy (angl. „raw types“) bylo zavedeno z důvodů dosažení zpětné kompatibility. Termín „raw“ lze překládat více způsoby a znamená **syrové**, **neozobené** nebo **očištěné** (je to obráceně než u cukru, zde – syrový cukr je nečištěný). Takovými syrovými typy jsou například rozhraní List nebo třída LinkedList.

8.2 Typové očišťování (type erasure):

Kompilátor vytváří jednobajtové reprezentace generických typů nebo metod a mapuje všechny instance generického typu či metody do bajtové reprezentace. Toto mapování je prováděno typovým čištěním. Účel typového čištění je odstranění všech informací vztahujících se k typovým parametrům a typovým argumentům. Navíc, kompilátor přidává typové kontroly a přetypování, kde je to nutné a vkládá doplňující můstkové metody, kdykoliv je to nutné. Je důležité porozumět typovému čištění, jelikož jistým efektům vztahujícím se ke genericitě Javy je obtížné porozumět bez chápání procesu překladu.
Typové čištění si lze představit jako překlad ze zdrojového generického kódu do běžného kódu Javy. V podstatě kompilátor je vhodnější na čištění a překládá Javu přímo do bajtového kódu. Jenže vytvořený kód je ekvivalentní negenerickému kódu, který uvidíte v následujících příkladech.

Kroky provedené při typovém čištění zahrnují:

Vyřazení typových parametrů:

Když kompilátor najde definici generického typu nebo metody, odstraní všechny výskyty typového parametru a nahradí je nejbližším rodičem, nebo typem Objekt, když není zadána žádná mez.

Vyřazení typových argument:.

Pokud kompilátor nalezne parametrizovaný typ, neboli instanci generického typu, odstraní typové argumenty. Například, typy List<String> , Set<Long> , a Map<String,?> jsou přeloženy do typu List, Set a Map respektive.
Příklad (před typovým čištěním):

```
interface Comparable <A> {
  public int compareTo( a that);
}
final class NumericValue implements Comparable <NumericValue> {
  private byte value;
  public  NumericValue (byte value) { this.value = value; }
  public  byte getValue() { return value; }
  public  int compareTo( NumericValue t hat) { return this.value - that.value; }
}
class Collections {
  public static <A extends Comparable<A>>A max(Collection <A> xs) {
    Iterator <A> xi = xs.iterator();
    a w = xi.next();
    while (xi.hasNext()) {
      a x = xi.next();
      if (w.compareTo(x) < 0) w = x;
    }
    return w;
  }
}
final class Test {
  public static void main (String[ ] args) {
    LinkedList <NumericValue> numberList = new LinkedList <NumericValue> ();
    numberList .add(new NumericValue((byte)0));
    numberList .add(new NumericValue((byte)1));
    NumericValue y = Collections.max( numberList );
  }
}
```

Typové parametry jsou zelené a typové argumenty jsou modré. Během typového čištění jsou typové argumenty zrušeny a typové parametry jsou nahrazeny jejich levou mezí.
Příklad (po typovém čištění):

```
interface Comparable {
  public int compareTo( Object that);
}
final class NumericValue implements Comparable {
  private byte value;
  public  NumericValue (byte value) { this.value = value; }
  public  byte getValue() { return value; }
  public  int compareTo( NumericValue t hat)   { return this.value - that.value; }
  public  int compareTo(Object that) { return this.compareTo((NumericValue)that);  }
}
class Collections {
  public static Comparable max(Collection xs) {
    Iterator xi = xs.iterator();
    Comparable w = (Comparable) xi.next();
    while (xi.hasNext()) {
      Comparable x = (Comparable) xi.next();
      if (w.compareTo(x) < 0) w = x;
    }
    return w;
  }
}
final class Test {
  public static void main (String[ ] args) {
    LinkedList numberList = new LinkedList();
    numberList .add(new NumericValue((byte)0));
```

```
        numberList .add(new NumericValue((byte)1));
        NumericValue y = (NumericValue) Collections.max( numberList );
    }
}
```

Generické rozhraní Comparable je přeloženo do negenerického rozhraní a typ bez meze je nahrazen typem Object.

Třída NumericValue implementuje negenerické rozhraní Comparable a kompilátor přidal tzv. přemosťovací metodu. Přemosťovací metoda je nutná, aby třída NumericValue zůstala třídou implementující rozhraní Comparable i po typovém čištění.

Generická metoda max je přeložena do negenerické metody a ohraničený parametr a je nahrazen jeho horní mezí, jmenovitě Comparable. Parametrické rozhraní Iterator<A> je přeloženo do syrového typu Iterator a kompilátor přidá přetypování, kdykoliv je obdržen prvek ze syrového typu Iterator.

Použití parametrizovaného typu LinkedList<NumericValue> a generická metoda max v metodě main jsou přeloženy do použití negenerických metod a opět musí přidat přetypování.

8.3 Typové čištění parametrizovaného typu

Výsledkem typového čištění je typ bez typových argumentů.

Čištění parametrizovaného typu je parametrizovaný typ bez jakýchkoliv typových argumentů (čili syrový typ). Toto platí i o polích a vnořených typech.

Příklady:

parametrizovaný type	*typové čištění*
List<String>	List
Map.Entry<String,Long>	Map.Entry
Dvojice<Long,Long>[]	Dvojice[]
Comparable<? super Number>	Comparable

Typové čištění parametrizovaného je typ samotný.

8.4 Typové čištění typového parametru

Typové čištění je odstranění je převedení parametru na jeho horní (levou) mez nebo typ Object, není-li horní mez zadána. Typové čištění typového parametru je zrušení jeho horní meze. Typové čištění typového parametru bez meze je typ Object .

Příklady:

typové parametry	*typové čištění*
<T>	Object
<T extends Number>	Number
<T extends Comparable<T>>	Comparable
<T extends Cloneable & Comparable<T>>	Cloneable
<T extends Object & Comparable<T>>	Object
<S, T extends S>	Object,Object

8.5 Typové čištění generické metody

Výsledkem typového čištění generické metody je metoda se stejným jménem a její typové parametry jsou nahrazeny jejich možným typovým čištěním. Čištění signatury metody je signatura skládající se ze stejného jména a čištěním všech formálních parametrizovaných typů metody.

Příklady:

generická metoda	*typové čištění*
Iterator<E> iterator()	Iterator iterator()
<T> T[] toArray(T[] a)	Object[] toArray(Object[] a)
<U> AtomicLongFieldUpdater<U> newUpdater(Class<U> tclass, String fieldName)	AtomicLongFieldUpdater newUpdater(Class tclass,String fieldName)

8.6 Můstkové metody

Můstková metoda je metoda uměle vytvořená kompilátorem během typového čištění. Někdy je potřebná, pokud typ rozšiřuje nebo implementuje generickou metodu nebo rozhraní. Kompilátor vkládá můstkovou metodu do potomků generických rodičů pro zajištění, že podtypování bude fungovat.
Příklad před typovým čištěním:

```
interface Comparable <A> {
  public int compareTo( a that);
}
final class NumericValue implements Comparable <NumericValue> {
  private byte value;
  public  NumericValue (byte value) { this.value = value; }
  public  byte getValue() { return value; }
  public  int compareTo( NumericValue that) { return this.value - that.value; }
}
```

V příkladě výše třída NumericValue implementuje rozhraní Comparable<NumericValue> a musí tudíž překrýt nadřazané rozhraní pro metodu compareTo. Metoda přijímá třídu NumericValue Comparable<A> do syrového typu Comparable . Typové čištění mění signaturu metody `compareTo` tohoto rozhraní. Po typovém čištění přijímá jako argument typ Object.

Příklad po typovém čištění:

```
interface Comparable {
  public int compareTo( Object that);
}
final class NumericValue implements Comparable {
  private byte value;
  public  NumericValue (byte value) { this.value = value; }
  public  byte getValue() { return value; }
  public  int compareTo( NumericValue t hat)   { return this.value - that.value; }
  public  int compareTo(Object that) { return this.compareTo((NumericValue)that);  }
}
```

Po tomto překladu metoda NumericValue.compareTo(NumericValue) již není implementací metody compareTo. Typově čisté rozhraní Comparable vyžaduje metodu compareTo s argumentem typu Object , nikoliv NumericValue . Toto je vedlejší efekt typového čištění: dvě metody (v rozhraní a implementující třídě) měly identickou signaturu před typovým čištěním a po něm mají signaturu metody rozdílnou.

Aby třída NumericValue zůstala třídou správně implementující rozhraní Comparable, kompilátor vloží do třídy můstkovou metodu. Můstková metoda má stejnou signaturu jako metoda rozhraní před typovým čištěním, protože tato metoda má být implementována. Můstková metoda odkazuje na původní metodu v implementující třídě.
Existence můstkové metody neznamená, že lze vložit do metody compareTo libovolné objekty. Můstková metoda je implementační detail a kompilátor zajišťuje, aby nemohla být normálně zavolána.

Příklad neplatného pokusu zavolání metody:

```
NumericValue value = new NumericValue((byte)0);
value.compareTo(value);  // v pořádku
value.compareTo("abc");  // chyba
```

Kompilátor nevolá můstkovou metodu v případě, byl-li vložen objekt jiného typu než NumericValue do metody compareTo. Místo toho zamítne volání s chybovou zprávou oznamující, že metoda compareTo očekává typ NumericValue a jiné typy nejsou povoleny.

Můžete také zkusit zavolat můstkovou metodou pomocí reflexe. ale pokud nabídnete argument jiného typu než NumericValue, metoda vyhodí výjimku ClassCastException díky přetypování v můstkové metodě.

Příklad (pokus o volání můstkové metody pomocí reflexe, která neuspěje):

```
int reflectiveCompareTo(NumericValue value, Object other)
  throws NoSuchMethodException, IllegalAccessException, InvocationTargetException
{
  Method meth = NumericValue.class.getMethod("compareTo", new Class[]{Object.class});
  return (Integer)meth.invoke(value, new Object[]{other});
}
NumericValue value = new NumericValue((byte)0);
reflectiveCompareTo(value, value);  // v pořádku
reflectiveCompareTo(value,"abc");   // chyba (ClassCastException)
```

Přetypování na typ NumericValue v můstkové metodě selže při vložení argumentu jiného než NumericValue, s vyvoláním výjimky ClassCastException. Takto je zaručeno, že můstková metoda při zavolání neuspěje s jiným než vlastním argumentem.

8.6.1 Za jakých okolností dochází k vygenerování můstkové metody?

Můstkové metody jsou nutné, když typ rozšiřuje nebo implementuje generickou třídu nebo rozhraní a typové čištění mění signaturu jakékoliv ze zděděných metod. Níže je příklad třídy rozšiřující generickou nadtřídu.
Příklad (před typovým čistěním):

```
class NadTrida <T extends Mez> {
  public void m1( T arg) { ... }
  public T m2() { ... }
}
class PodTrida extends NadTrida <PodtypMeze> {
   public void m1(PodtypMeze arg) { ... }
   public PodtypMeze m 2() { ... }
}
```

Příklad (po typovém čištění):

```
class NadTrida {
  void m1( Mez arg) { ... }
Mez m2() { ... }
}
```

```
class PodTrida extends NadTrida {
  public void m1(PodtypMeze arg) { ... }
  public void m1(Mez arg) { m1((PodtypMeze)arg); }
  public PodtypMeze m2() { ... }
  public Mez        m2() { return m2(); }
}
```

Typové čištění mění signaturu metod nadtřídy. Po typovém čištění již metody podtřídy tedy nepřekrývají metody nadtřídy. Pro zajištění překrytí metod kompilátor tedy přidá můstkové metody. Kompilátor musí přidat můstkové metody dokonce, i když podtřída nepřekrývá zděděné metody.

Příklad (před typovým čištěním):

```
class NadTrida <T extends Mez> {
  public void m1( T arg) { ... }
  public T m2() { ... }
}
class JinaPodTrida extends NadTrida <PodtypMeze> {
}
```

Po typovém čištění:

```
class NadTrida {
  void m1( Mez arg) { ... }
  Mez m2() { ... }
}
class JinaPodTrida extends NadTrida {
  public void  m1(Mez arg) { super.m1((PodtypMeze)arg); }
  public Mez m2() { return super.m2(); }
}
```

Podtřída je odvozena od částečné instance nadtřídy a tedy dědí metody s částečnou signaturou. Po typovém čištění signatura metody nadtřídy je rozdílná od signatury metody zděděné podtřídy. Kompilátor tedy přidá můstkové metody, aby i podtřída měla očekávané zděděné metody. Není třeba můstkové metody, když typové čištění nemění signaturu jakékoliv z metod generického rodiče. Také není nutná, pokud se signatury nadtřídy a podtřídy mění stejným způsobem. Toto platí, pokud je potomek sám o sobě generickým.

Příklad (před typovým čištěním):

```
interface Callable <V> {
  public V call();
}
class Task <T> implements Callable <T> {
  public T call() { ... }
}
```

Příklad (po typovém čištění):

```
interface Callable {
  public Object call();
}
class Task implements Callable {
  public Object call() { ... }
}
```

Návratový typ metody call se mění jak v rozhraní tak i v implementující třídě.Po typovém čištění mají obě metody stejnou signaturu, čili metoda podtřídy implementuje metody rozhraní bez můstkové metody. Samozřejmě ale nestačí jen, aby podtřída byla generická. Hlavní je, aby se signatura metody změnila během typového čištění. Jen tehdy potřebujeme můstkovou metodu.

Příklad (před typovým čištěním):

```
interface Copyable <V> extends Cloneable {
  public V copy();
}
class Triple <T extends Copyable<T>> implements Copyable <Triple<T>> {
  public Triple<T> copy() { ... }
}
```

Příklad (po typovém čištění):

```
interface Copyable extends Cloneable {
  public Object copy();
}
class Triple implements Copyable {

    public Copyable copy() { ... }
    public Triple copy() { return copy(); }
    }
```

Signatura metody se změní na Object copy() v rozhraní a Copyable copy() v podtřídě (kvůli mezi parametru). ve výsledku přidá kompilátor můstkovou metodu.

8.7 Přetypování při překladu

Proč přidává kompilátor přetypování při překladu generických typů?

Protože návratový typ metody generického typu se může změnit jakožto vedlejší efekt při typovém čištění.

Během typového čištění kompilátor nahradí typové parametry jejich levou (horní) mezí, nebo typem Object, není li žádná specifikována. Což znamená, že metody, jejichž návratový typ je typový parametr, může vrátit odkaz na horní mez nebo na typ Object, místo konkrétního typu specifikovaného v generickém typu a jaký očekává volající objekt. Je nutné přetypování z horní meze nebo typu Object na specifický typ.

Příklad (před typovým čištěním):

```
public class Dvojice<X,Y> {
  private X prvni;
  private Y druhy;
  public Dvojice(X x, Y y) {
    prvni = x;
    druhy = y;
  }
  public X getPrvni() { return prvni; }
  public Y getDruhy() { return druhy; }
  public void setPrvni(X x) { prvni = x; }
  public void setDruhy(Y y) { druhy = y; }
}
final class Test {
  public static void main(String[] args) {
```

```
        Dvojice<String,Long> pair = new Dvojice<String,Long>("limit", 10000L);
        String s = pair.getPrvni();
        Long   l = pair.getDruhy();
        Object o = pair.getDruhy();
      }
    }
```

Příklad (po typovém čištění):

```
public class Dvojice {
  private Object prvni;
  private Object druhy;
  public Dvojice( Object x, Object y) {
    prvni = x;
    druhy = y;
  }
  public Object getPrvni() { return prvni; }
  public Object getDruhy() { return druhy; }
  public void setPrvni( Object x) { prvni = x; }
  public void setDruhy( Object y) { druhy = y; }
}

final class Test {
  public static void main(String[] args) {
    Dvojice pair = new Dvojice("limit", 10000L);
    String s = (String) pair.getPrvni();
    Long   l = (Long)    pair.getSeond();
    Object o =          pair.getDruhy();
  }
}
```

Po typovém čištění metody getPrvni a getDruhy typu Dvojice mají obě návratový typ Object. Zatímco deklarovaný statický typ dvojice v našem testu je Dvojice<String,Long>, volání metod getPrvni a getDruhy očekává String a Long jako návratové typy. Bez přetypování by toto nefungovalo a k funkčnosti musí tedy přidat kompilátor nutná přetypování z Object na String a Long.

Vložená přetypování nemohou vyvolat při běhu výjimku typu ClassCastException, protože kompilátor je si již jist, že při kompilaci obě pole odkazují na objekty očekávaného typu.
Kompilátor vydá chybovou metodu, pokud budou argumenty typů jiných než String a Long vloženy do konstruktoru metody set. Čímž je zaručen úspěch přetypování.
Obecně, diskrétně kompilátorem přidaná přetypování jsou garancí nevyvolání výjimky ClassCastException, pokud došlo k překladu bez varovných zpráv.

Při volání metod jsou vkládána implicitní přetypování, pokud se návratový typ změnil během typového čištění. Volání metod, jejichž *typový argument* se změnil během typového čištění nepotřebují vložit přetypování. Například, po typovém čištění metody setPrvni a setDruhy třídy Dvojice přijímají argument typu Object. Jejich volání s argumentem specifičtějšího typu je možné bez vložení přetypování.

8.7.1 Typové čištění pro parametr s více mezemi

Při typovém čištění pro parametr s více mezemi přidá kompilátor nutná přetypování - během procesu typového čištění kompilátor nahradí typové parametry jejich horní mezí nebo typem Object, není-li mez uvedena. Jak to ale funguje, když má parametr uvedených více mezí?

Příklad (před typovým čištěním):

```
interface Runnable {
  void run();
}
interface Callable<V> {
  V call();
}
class X<T extends Callable<Long> & Runnable> {
  private T task1, task2;
  ...
  public void do() {
    task1.run();
    Long result = task2.call();
  }
}
```

Příklad (po typovém čištění):

```
interface Runnable {
  void run();
}
interface Callable {
  Object call();
}
class X {
  private Callable task1, task2;
  ...
  public void do() {
    ( (Runnable) task1).run();
    Long result = (Long) task2.call();
  }
}
```

Typový parametr T je nahrazen mezí Callable, což znamená, že obě položky jsou považovány zároveň jako proměnné typu Callable. Metody s horní mezí (což je v našem příkladě Callable) lze volat přímo. pro volání metod jiných mezí, (v našem příkladě Runnable) kompilátor přidá přetypování k nejbližší mezi, pro zpřístupnění metody. Vložené přetypování nemůže vyvolat výjimku ClassCastException, protože kompilátor již zajistil, že obě pole jsou referencemi k typu objektu, jež je mezi oběma mezemi.
Obecně, kompilátorem přidaná přetypování jsou garancí nevyvolání výjimky ClassCastException, pokud došlo k překladu bez varovných zpráv. Tím je zaručena typová bezpečnost.

9. Nejednoznačnosti, zakázané operace a kolize

9.1 Parametrizované typy jako součást pole

Lze vytvořit pole, jehož komponent je konkrétní parametrizovaný typ?

Nelze, jelikož to není typově bezpečné. Pole jsou kovariantní, což znamená, že pole rodičů je rodičem pro pole všech svých potomků . Například, Object[] je rodičem pro String[], a k poli řetězců lze přistupovat skrz odkazovou proměnnou typu Object[].

Příklad kovariantních polí:

```
Object[] objArr = new String[10];  // lze
objArr[0] = new String();
```

Dále pole obsahují informace o runtime typu svých prvků. Informace o runtime typu prvku jsou použity při vkládání prvků do pole za účelem kontroly, že nebude vložen žádný prvek neslučitelného typu.

Ukázka typové kontroly:

```
Object[] objArr = new String[10];
objArr[0] = new Long(0L); // zkompiluje se, při spuštění ohlásí ArrayStoreException
```

Proměnná typu Object[] odkazuje na String[], což znamená možnost použití pouze řetězců typu String. Při vložení prvku do pole se použije informace o prvku pole ke kontrole – dojde ke kontrole vložení. V našem příkladě kontrola vložení do pole neuspěje kvůli faktu, že se snažíme vložit Long do pole typu String s. Neúspěch kontroly vložení do pole je ohlášen vyvoláním výjimky ArrayStoreException.

Problém nastane, pokud pole obsahuje prvky, jejichž typem je konkrétní parametrizovaný typ. Kvůli typovému čištění pak parametrizované typy nemají přesné informace o běhovém typu. Následkem toho nemůže fungovat kontrola vkládání do pole, jelikož používá dynamickou typovou informaci náležící poli s neupřesněným typem prvku pro kontrolu vložení do něj.

Příklad kontroly vložení do pole s parametrizovaným typem prvku:

```
Dvojice<Integer,Integer>[] intDvojiceArr = new Dvojice<Integer,Integer>[10]; //
zakázaná operace
Object[] objArr = intDvojiceArr;
objArr[0] = new Dvojice<String,String>("",""); // mělo by neuspět, ale je zkompilováno
```

Pokud by byly dovoleny konkrétní parametrizované typy, pak referenční proměnné typu Object[], by mohly odkazovat na Dvojice<Integer,Integer>[], viz. příklad výše.

Při běhu se provádí kontrola vkládání prvku do pole. Pokud se snažíme vložit Dvojice<String,String> do Dvojice<Integer,Integer>[], očekáváme, že to typová kontrola nedovolí. Samozřejmě, Java Virtual Machine neumí zde najít neshodu, při běhu po čištění typů objArr by měl dynamický typ Dvojice[] a prvek k uložení by měl odpovídající typ Dvojice. Proto typová kontrola uspěje, i v případě, kdy by neměla.

Pokud by bylo dovoleno deklarovat pole s prvky, jehož prvky by byly parametrizované typy, mohli bychom se ocitnout v nepřípustné situaci. Pole v našem příkladě by obsahovalo různé typy Dvojice místo dvojic stejného typu. Toto je v rozporu oproti očekávání, že pole budou obsahovat prvky stejného typu (nebo potomků). Tato nechtěná situace by mohla vést k chybě v programu v okamžiku pokusu o čtení prvku pole, ať už voláním metody nebo přímo.

Příklad chyby při běhu:

```
Dvojice<Integer,Integer>[] intDvojiceArr = new Dvojice<Integer,Integer>[10]; //
nepřípustné
Object[] objArr = intDvojiceArr;
objArr[0] = new Dvojice<String,String>("",""); // uspěje ačkoliv by nemělo.

Integer i = intDvojiceArr[0].getPrvni(); // ohlásí výjimku při běhu ClassCastException
```

Na první prvek pole je použita metoda getPrvni a vrací String místo Integer, protože první prvek v poli intDvojice je dvojice String, a nikoliv dvojice slov jak by se očekávalo. Nevinně vypadající přiřazení proměnné Integer vyvolá výjimku ClassCastException, i přesto, že se v kódu nenachází žádné přetypování.
Tato neočekávaná výjimka ClassCastException je považována za důkaz porušení typové bezpečnosti.

Je zakázáno programům vytvářet pole obsahující prvky parametrizovaného typu, jelikož nejsou typově bezpečné. Ze stejného důvodu jsou zakázána i pole obsahující prvky, jejichž typ je parametrizovaný s žolíkem. Jsou dovolena pouze pole s prvky parametrizovanými neohraničeným žolíkem. Obecněji jsou povoleny pouze typy s neohraničenými prvky, zatímco prvky s nevolitelnými prvky jsou zakázány.

9.2 Hodnoty polí parametrizovaného typu

Mohu deklarovat pole proměnných, jehož typ prvku je konkrétní parametrizovaný typ?

Ano, můžete, ale není to ani praktické ani typově bezpečné. Můžeme deklarovat referenční proměnnou typu pole, jehož typ prvku je konkrétní parametrizovaný typ. Pole tohoto by neměla být vůbec vytvářena. Proto takto referenční proměnná nemůže odkazovat na pole svého typu. Jediné, na co může odkazovat je typ null, pole, jehož typ prvků je neparametrizovaný potomek konkrétního parametrizovaného typu, nebo pole, jehož typ prvků je odpovídající neparametrizovaný typ. Žádný z těchto případů není užitečný i přesto, že jsou povoleny.

Příklad pole referenčních proměnných s parametrizovaným typem prvku:

```
Dvojice<String,String>[] arr = null;  // v pořádku
arr = new Dvojice<String,String>[2] ; // chyba: pokus o vytvoření generického
pole
```

Ukázka kódu dokazuje, že lze vytvořit referenční proměnnou typu Dvojice<String,String>[].
Ovšem deklarace takového pole bude již zamítnuta. Ale můžeme nechat referenční proměnnou typu Dvojice<String,String>[] odkazovat na pole neparametrizovaného typu.
Příklad dalšího pole referenční proměnné s parametrizovaným typem prvku:

```
 Class Jmeno extends Dvojice<String,String> { ... }
Dvojice<String,String>[] arr = new Jmeno[2] ;    // v pořádku
```

Ukázka kódu ukazuje, že lze deklarovat referenční proměnnou typu Dvojice<String,String>[], ale vytvoření takového pole je zakázáno.
Příklad jiného pole s referenční proměnnou odkazující na pole potomků, nedoporučuje se:

```
void vypisPoleDvojicRetezcu( Dvojice<String,String>[] pa) {
  for (Dvojice<String,String> p : pa)
    if (p != null)
      System.out.println(p.getPrvni()+" "+p.getDruha());
```

```
}
Dvojice<String,String>[] vytvorPoleRetezcuDvojic() {
  Dvojice<String,String>[] arr = new Jmeno[2];
  arr[0] = new Jmeno("Angelika","Langer");   // v pořádku
  arr[1] = new Dvojice<String,String>("a","b");  // kompilace bez chyby (způsobí
ArrayStoreException)
  return arr;
}
void extrahujDvojiceStringZPole( Dvojice<String,String>[] arr) {
  Jmeno name = (Jmeno) arr[0];   // ok
  Dvojice<String,String> p1 = arr[1];    // ok
}
void test() {
  Dvojice<String,String>[] arr = vytvorPoleRetezcuDvojic ();
  vypisPoleDvojicRetezcu(arr);
 extrahujDvojiceStringZPole(arr);
}
```

Příklad ukazuje, že referenční proměnná Dvojice<String,String>[] může odkazovat na pole typu Jmeno[], kde Jmeno je neparametrizovaný potomek typu Dvojice<String,String>[] . Samozřejmě použití referenční proměnné typu Dvojice<String,String>[] nenabízí žádnou výhodu při využití proměnné typu Jmeno[]. Spíše naopak je příležitostí pro vznik chyb.

Například, v metodě vytvorPoleRetezcuDvojic kompilátor by dovolil vložit kódu vložení elementu typu Dvojice<String,String>, do pole, ačkoliv referenční proměnná je Dvojice<String,String>[]. Až při běhu samozřejmě vložení selže s výjimkou ArrayStoreException, jelikož se snažíme vložit typ Dvojice do typu Jmeno[]. To samé se stane, pokud se pokusíme vložit do pole syrový typ Dvojice. při kompilaci bude ohlášeno „unchecked warning“ varování a při běhu skončí výjimkou ArrayStoreException. Stejně tak, pokud bychom použili Jmeno[] místo Dvojice<String,String>.

Také si pamatujte, že proměnná typu Dvojice<String,String>[] nikdy nemůže odkazovat na pole obsahující prvky typu Dvojice<String,String>. Pokud chceme získat původní typ prvků pole, což je v našem příkladě Jmeno, musíme přejmenovat z Dvojice<String,String> na Jmeno, jak je ukázáno v metodě extrahujDvojiceStringZPole. Zde opět s použitím proměnné Jmeno[], to bude názornější.

Příklad (vylepšená verze):

```
void vypisPoleDvojicRetezcu ( Dvojice<String,String>[] pa) {
  for (Dvojice<String,String> p : pa)
    if (p != null)
      System.out.println(p.getPrvni()+" "+p.getDruhy());
}
Jmeno[] vytvorPoleRetezcuDvojic () {
  Dvojice<String,String>[] arr = new Jmeno[2] ;
  arr[0] = new Jmeno("Angelika","Langer");   // ok
  arr[1] = new  Dvojice<String,String>("a","b");  // chyba
  return arr;
}
void extrahujDvojiceStringZPole( Jmeno[] arr) {
  Jmeno name = arr[0];         // ok (potřebuje přetypování)
  Dvojice<String,String> p1 = arr[1];    // ok
}
void test() {
  Jmeno[] arr = vytvorPoleRetezcuDvojic();
vypisPoleDvojicRetezcu(arr);
 extrahujDvojiceStringZPole (arr);
}
```

Potom položka v poli proměnných , jehož typ prvku je parametrizovaný typ, nemůže odkazovat na pole svého typu, a tato referenční proměnná nemá smysl (typ prvku se neshoduje s typem pole).

Ještě k větším problémům může dojít, pokud se pokusíme proměnnou přiřadit k poli syrového typu místo jeho potomku. Zaprvé to vede k mnoha varovným hlášením „unchecked warning" protože spojujeme parametrizované a syrové typy. Za druhé, což je důležitější, tento přístup není typově bezpečný a trpí mnoha nedostatky vedoucí k zákazům polí konkrétních instancí.

Nezáleží na přístupu, je vždy lepší se vyhnout užívání polí referencích proměnných, jehož typy prvků jsou konkrétní parametrizované typy. Všimněte si, že toto platí i pro pole, jehož prvky jsou parametrizované pomocí žolíků. Mají smysl pouze pole s prvky neparametrizovanými žolíky. Pouze parametrizovaný typ s neohraničeným žolíkem je tzv. reifiable typ (úplný, určitý typ). Je možné vytvořit pouze pole s úplnými prvky; pole referenčních proměnných může odkazovat na pole svého typu a nedostatky diskutované výše neexistují pro pole s neohraničenými žolíky.

9.3 Použití žolíka při deklaraci objektu

Můžeme použít žolíka při deklaraci objektu? Nemůžeme, objekt sice lze vytvořit, ale nelze s ním dále pracovat, jelikož generický typ nebude definován. Tedy vytvořit lze, ale takový objekt má téměř nulové praktické využití. Důvod je ten, že generický argument je odvozován od parametru statického typu, nikoliv dynamického.
Uvažujme následující příklad:

```
List<? extends Number> seznamInteger=new ArrayList<Integer>();
    // seznamInteger.add(15); //chyba!
```

Když se pokusíme přidat integer 15 do objektu seznamItneger, obdržíme následující chybové hlášení: *cannot find symbol: method add(int).* Proč tomu tak je? Jak jsme si řekli výše, kompilátor odvozuje parametr od parametru statického typu, a tudíž od „? extends Number" nelze odvodit konkrétní typový parametr. Proto taky metoda add(int) pro tento objekt neexistuje. Jediné, na co můžeme takový objekt použít, je jako pomocná proměnná.

```
List<? extends Number> cisla;
List<Integer> cisla1=new ArrayList<Integer>()
cisla1.add(1);
cisla1.add(3);
cisla1.add(6);
List<Integer> cisla2=new ArrayList<Integer>()
cisla2.add(2);
cisla2.add(5);
cisla2.add(0);
List<Double> cisilka1=new ArrayList<Double>()
cisilka1.add(0.5);
cisilka1.add(1.2);
cisilka1.add(3.1);
List<Integer> cisla1=new ArrayList<Integer>()
cisilka1.add(3.1);
cisilka1.add(1.7);
cisilka1.add(2.5);
cisla=cisla1;
cisla1=cisla2;
cisla2=cisla;
cisla=cisilka1;
```

```
cisilka1=cisilka2;
cisilka2=cisla;
```

Vzhledem ale k tomu, že na pomocné proměnné jsou preferovány generické metody, s podobnou konstrukcí se zřejmě nesetkáme. Přesto je důležité vědět, že žolíky při deklaraci generických objektů nemá valný smysl využívat.
Závěr: pokud chceme vytvořit seznam schopný pojmout jakýkoliv objekt typu Number, nezbývá než použít konstrukci:

```
List< Number> seznamInteger=new ArrayList<Number>();
```

Samozřejmě, k takovému seznamu nelze přiřadit seznam typu Integer, Double a další, ale opět jen jiný seznam s typem Number.

10. Příklady vlastních generických typů

10.1 Definice vlastního generického typu

Protože ne všechny generické třídy nabízí dostatečné možnosti v programu a navíc je třeba někdy výhodné parametrizovat více částí, nabízí Java i možnost parametrizace celé vlastní třídy, tedy objektu typu Class.

Samozřejmě v generické třídě není pravidlem používat jakékoliv seznamy či další třídy z balíčku java.util, takže si můžeme definovat vlastní třídu, kde si můžeme definovat vlastní typ seznamu. Jako příklad si definujme třídu, která bude používat ArrayList, bude mít velikost omezenou na určitý počet prvků, přičemž si bude pamatovat posledních maximálně x prvků:

```
import java.util.ArrayList;
import java.util.ArrayList;
/**
 * Fronta umožňující vkládat pouze omezené množství prvků x,;
 * pamatuje si posledních x prvků
 */
public class OmezenaFronta<E>
{
   // vlastní seznam a jeho max délka – ta se počítá nebo zadává až v konstruktorech
   private ArrayList<E> ol;
   public final int maxdelka;

   /**
    * Výchozí kapacita je 10 prvků
    */
   public OmezenaFronta()
   {
      maxdelka=10;
      ol=new ArrayList<E>(maxdelka);
   }
   /**
    * zde volíme max délku
    */
   public OmezenaFronta(int maxdelka)
   {
      ol=new ArrayList<E>(maxdelka);
      this.maxdelka=maxdelka;
   }
   /**
    * kapacita se nastaví dle počtu vložených prvků
    */
   public OmezenaFronta(E... ee)
   {
      ol=new ArrayList<E>(ee.length);
       for( E e : ee )
       ol.add(e);
      this.maxdelka=ol.size();
   }
   /**
    * byla-li by překročena kapacita, smaže první a vkládá prvek na konec
    */
   public void zarad(E e)
   {
```

```
    if (ol.size()==maxdelka)
    ol.remove(0);
    ol.add(e);
}
/**
 * vloží pole prvků
 */
public void zarad( E... ee)
{
    for( E e : ee )
    zarad(e);
}
/**
 * pokusí se vymazat zadaný prvek a vrátí true, pokud existoval v seznamu
 */
public boolean vymaz(E e)
{
    return ol.remove(e);
}
/**
 * vymaže celý seznam
 */
public void vymazVse()
{
    ol.clear();
}
/**
 * vrátí poslední prvek beze změny seznamu
 */
public E posledniPrvek()
{
    return ol.get(ol.size()-1);
}
/**
 * vrátí maximální počet prvků, které si pamatuje fronta
 */
public int getMaxPrvku()
{
    return maxdelka;
}
/**
 * vrátí true, je-li seznam neprázdný
 */
public boolean isDalsi()
{
    return (ol.size() > 0);
}
/**
 * vrátí první prvek a smaže jej z fronty
 */
public E dalsi()
{
    //Při získávání odkazů z kontejneru již nemusíme vracené odkazy
    //přetypovávat , to za nás udělá překladač.
    E ret = ol.get(0);
    ol.remove(0);
    return ret;
}
/**
 * vrátí seznam prvků obsažených ve frontě
```

```
   */
  public ArrayList<E> prvky()
  {
    return ol;
  }
  public static void test()
  {
    OmezenaFronta<String> fString = new OmezenaFronta<String>(3);
    for(String s : new String []{"raz","dva","tři","čtyři","pět"})
    fString.zarad(s);
    System.out.println("\nFronta String po naplnění:"   + fString.prvky());
    while( fString.isDalsi() ){
      System.out.print(" " +fString.dalsi() );
    }
    System.out.println("\nFronta po vyprázdnění:" + fString.prvky());
    OmezenaFronta<Double> fDouble=new OmezenaFronta<Double>(3);
    for( Double d : new Double []{1d,2d,3d,4d,5d})
    fDouble.zarad(d);
    System.out.println("\nFronta Double po naplnění:"  + fDouble.prvky());
    while( fDouble.isDalsi() ) {
      System.out.print(" " + fDouble.dalsi() );
    }
    System.out.println("\nFronta po vyprázdnění:" + fDouble.prvky());
    // Následující příkaz by způsobil syntaktickou chybu,  protože
    // konstruktor očekává parametry Double nebo alespoň  double.
    //    fDouble = new FrontaP<Double> ( 1, 2, 3 );
    //    fDouble = new FrontaP<Integer>( 1, 2, 3 );
      OmezenaFronta<Integer> fInteger = new OmezenaFronta<Integer>( 1,2, 3 );
    // Následující příkaz by způsobil syntaktickou chybu,  protože
      //Typovým parametrem nesmí být primitivní typ
        //FrontaP<int> fInt = new FrontaP<int>( 1, 2, 3 );
  }
}
```

Projdeme si nyní tuto definici krok za krokem a podívejme se, jaké konstrukce jsme zde použili.

Prvním zajímavým příkazem je samotná hlavička třídy, v níž jsme deklarovali použití formálního typového parametru E. Jak vidíte, typový parametr se zde deklaruje ve špičatých závorkách.

Typový seznam jsme také použili v definici pro ArrayList ol, kde jsme definovali ol jako omezený seznam. Samotná inicializace seznamu se provádí až v konstruktorech.

```
private ArrayList<E> ol=new ArrayList<E>(maxdelka);
```

V bezparametrickém konstruktoru pouze vytvoříme vnitřní seznam o velikosti 10, a podíváme se dále na konstruktory s parametry deklarovanými jako instance třídy E.

Typové parametry mohou být použity nejenom k definici konkrétního použitého generického typu, ale také k definici typu parametrů a typu návratové hodnoty definovaných metod. Jako příklad může sloužit metoda zarad(E), v níž je pomocí typového parametru specifikován typ parametru metody, a metoda dalsi(), v jejíž definici je prostřednictvím typového parametru specifikován typ její návratové hodnoty.

V přiloženém testu jsou vytvořeny dvě fronty: fronta fString, do níž jsou zařazovány textové řetězce, a fronta fDouble, do niž jsou zařazovány instance třídy Double. Můžete si ověřit, že takto definované fronty nedovolí zařadit do fronty instance jiného než deklarovaného typu.

Na konci testu je několik zakomentovaných příkazů, které obsahují syntaktickou chybu vyvolanou pokusem o přiřazení hodnoty nesprávného typu.

První příkaz inicializuje frontu s prvky typu Double hodnotami typu int. Je ale nutné zopakovat, že při používání automatického převodu z primitivních typů na příslušné obalové typy je třeba zadat převáděnou hodnotu správného typu. Požaduje-li proto metoda hodnoty typu Double, můžeme místo nich zadat hodnoty typu double, ale nemůžeme zadávat hodnoty typu int.

Druhý příkaz se pokouší tento problém napravit tím, že definuje frontu typu OmezenaFronta<Integer>. To by sice bylo přijatelné řešení, ale nesměl by se pokoušet uložit odkaz na tuto frontu do proměnné fDouble, která je typu OmezenaFronta<Double>. i když si za chvíli vysvětlíme, že se při překladu všechny typy očistí, takže pro virtuální stroj budou obě instance stejného typu, překladač podobná přiřazení neakceptuje.

Třetí příkaz ukazuje správné řešení: neexistuje-li proměnná potřebného typu, je třeba ji vytvořit a odkaz do ní uložit.

Poslední příkaz upozorňuje na to, že typovými parametry nesmí být primitivní datové typy, ale musí jimi být vždy třídy či rozhraní, aby jejich instance bylo možno převést na společného rodiče.

10.2 Typová bezpečnost a dědičnost

Pokud použijeme parametrizovaný seznam, můžeme pracovat nejen s datovými typy, které jsou shodné typovým argumentem, ale i jeho potomky. Ale i jeho potomky. Vytvoříme si nyní třídu Osoba, a odvodíme si od něj třídy Student a Zamestnanec. Třídy Student a Zamastnanec budou uchovávat informace o jménech a povolání.

Zde je ukázka:

```
public abstract class Osoba
{
        public abstract String jmeno();
        public abstract String cinnost();
}
-------------------------------------------------
public class Student extends Osoba
{
   private String jmeno;
   private String prijmeni;
   private String obor;
   public Student(String jmeno, String prijmeni,String obor)
   {
      this.jmeno=jmeno;
      this.prijmeni=prijmeni;
      this.obor=obor;
   }
   public String jmeno()
   {
      return jmeno+" "+prijmeni;
```

```
    }
    public String cinnost()
    {
        return obor;
    }
}
-------------------------------------------------
public class Zamestnanec extends Osoba
{
    private String jmeno;
    private String prijmeni;
    public String povolani;
    public Zamestnanec(String jmeno, String prijmeni,String povolani)
    {
        this.jmeno=jmeno;
        this.prijmeni=prijmeni;
        this.povolani=povolani;
    }
    public String jmeno()
    {
        return jmeno+" "+prijmeni;
    }
    public String cinnost()
    {
        return povolani;
    }
}
--------------------------------------------------------
public class Dvojice
{
    private String x;
    private String y;
    public Dvojice(String x, String y)
    {
        this.x=x;
        this.y=y;
    }

}
--------------------------------------------------
import java.util.ArrayList;

public class Seznam
{
    private ArrayList<Osoba> osoby=new ArrayList<Osoba>();

    public Seznam()
    {
        osoby.add(new Student("Pavel","Říha","VTI"));
        osoby.add(new Student("Radek","Bartůšek","VTI"));
        osoby.add(new Student("Petr","Novák","VTI"));
        osoby.add(new Zamestnanec("Jaroslav","Icha","VTI učitel"));
        osoby.add(new Zamestnanec("Soňa","Jurásková","sektretářka VTI"));
        //osoby.add(new Dvojice("Nebe","Dudy")); // chyba
    }
    public void vypisStudenty()
    {
        for(Osoba o: osoby)
        {
            if (o instanceof Student)
```

```
                {
                    System.out.println(o.jmeno()+","+o.cinnost());
                }
            }
        }
        public static void main(String[] args)
        {
            Seznam s=new Seznam();
            s.vypisStudenty();
        }
    }
```

Jak můžeme vidět, díky parametrizaci ArrayListu osoby jsme se nejen vyhnuli přetypování v metodě add, ale také zároveň není možné vložit do seznamu objekt typu Dvojice, který není v žádné potřebné příbuznosti se třídou Osoba. Pokud bychom nepoužili parametrizaci, vložený objekt Dvojice by mohl vyvolat výjimku programu nejpozději v metodě vypisStudenty(), pokud by došlo k zavolání jedné z metod jmeno() nebo cinnost(). Zde jsme se této chybě vyhnuli elegantním použitím operátoru instanceof. I přesto bez využití typových parametrů nelze zabránit vložení nežádoucího objektu, zde objekt Dvojice.

Pro představu, zde je kód třídy Seznam, pokud bychom vynechali parametrizaci seznamu osoby:

```
    import java.util.ArrayList;

    public class NonGenSeznam
    {
        private ArrayList osoby=new ArrayList();

        public NonGenSeznam()
        {
            osoby.add((Object) new Student("Pavel","Říha","VTI"));
            osoby.add((Object) new Student("Radek","Bartůšek","VTI"));
            osoby.add((Object) new Student("Petr","Novák","VTI"));
            osoby.add((Object) new Zamestnanec("Jaroslav","Icha","VTI učitel"));
            osoby.add((Object) new Zamestnanec("Soňa","Jurásková","sektretářka VTI"));
            osoby.add((Object) new Dvojice("Nebe","Dudy")); // chyba
        }
        public void vypisStudenty()
        {
            for(Object o: osoby)
            {
                if (o instanceof Student)
                {
                    System.out.println(((Student) o).jmeno()+","+((Student) o).cinnost());
                }
            }
        }
        public static void main(String[] args)
        {
            NonGenSeznam s=new NonGenSeznam();
            s.vypisStudenty();
        }
}
```

Nevýhody jsou zjevné: při každém vložením do seznamu jsme nejprve nuceni provést přetypování, a v iteraci cyklem for opět vyvstává nutnost zpětného přetypování.

10.3 Ohraničení typových hodnot parametrizovaných typů

Jak jsme si už ukázali, generické typy nabízejí typovou kontrolu a tvorbu čistě homogenních kolekcí. Ovšem i u použití těchto typů můžeme narazit na neočekávané problémy, nejčastěji ne-bezpečné iniciace dat. typů. Podívejme se na následující příklad:

```
public class Koule<E extends Number&Comparable<E>>
{
   private E r;
   public Koule(E e){  r=e;  }

   public Koule(){  r=e;  }

   //vypočte povrch pro zadaný poloměr E
   public double povrch()    {   return 4*Math.PI*Math.pow(r.doubleValue(),2d);  }
   //vypočte objem pro zadaný poloměr E
   public double objem()    { return 4/3*Math.pow(r.doubleValue(),3d); }

   public E polomer()    { return r; }

   public static <E extends Number&Comparable<E>> double povrch(E polomer)
   {    return 4*Math.PI*Math.pow(polomer.doubleValue(),2d); }

   public static <E extends Number&Comparable<E>> double objem(E polomer)
   {  return 4/3*Math.pow(polomer.doubleValue(),3d); }

   public static void test() {
      int i=5;
      short s=4;
      float f=4.3f;
      double d=3.1d;
      //u této konstrukce dostaneme varování
      Koule<Float> k1=new Koule(i);
      //správná parametrizace
      Koule<Integer> k1=new Koule<Integer>(i);
      System.out.println("Povrch koule s polomerem "+i+" je :"+kl.povrch()); }
}
```

Na řádku 47 vidíme příkaz

```
Koule<Float> k1=new Koule(i);
```

Při překladu pro `Koule<Float> k1=new Koule(i);` obdržíme varování a je zcela zřejmě, že překladač musel provést přetypování. Může se to zdát praktické, ale správný programátor by toto jistě považoval za chybu, neboť taková konstrukce by neměla být použita. Přesto, pro Float i Integer platí, že jsou potomky Number, tudíž lze toto provést. O řádku níže nalezneme již správnou parametrizaci objektu Koule.

11. Vybrané generické třídy v Java Development Kit

11.1 Třída java.lang.Class

Instance třídy Class reprezentují v Java třídy a rozhraní. Enum je typ třídy a Annotation je druh rozhraní. Každé pole také náleží ke třídě, která je reprezentována objektem Class, jenž je shodný pro všechna pole stejného typu všech velikostí a shodné dimenze. Primitivní typy Javy (boolean, byte, char, short, int, long, float, and double), a typ void jsou také reprezentovány jako jako objekty typu Class. Class nemá žádný veřejný konstruktor. Místo toho jsou objekty typu Class automaticky vytvářeny pomocí Java Virtual Machine jako třídy voláním metody defineClass() v class loaderu.
Následující příklad používá objekt typu Class k získání jména třídy objektu:

```
void printClassName(Object obj) {
   System.out.println("The class of " + obj +
                  " is " + obj.getClass().getName());
}
```

Je také možné získat třídu objektu i pro pojmenovaný typ používající literal třídy (JLS Section 15.8.2). Například:

```
System.out.println("Jméno třídy String je: "+ String.class.getName());
```

Balíček java.lang obsahuje generická rozhraní Comparable a Iterable , a generickou třídou Enum.

11.2 Rozhraní java.lang.Comparable a java.util.Comparator

public interface Comparable<E>{
public int compareTo(E object);
}

Rozhraní Comparable:

Rozhraní Comparable nabízí možnost uspořádání objektů pro každou třídu, ve které je implementováno. Tato schopnost uspořádání je nazývána jako *přirozená schopnost uspořádání*, a metoda třídy compareTo je proto označována jako přirozená porovnávací metoda, jelikož kolekce, jež implementují rozhraní Comparable, lze řadit pomocí metody Collections.sort (a Arrays.sort).

Objekty implementující toto rozhraní lze využít jako klíče v tříděné mapě (SortedMap) nebo prvky v tříděné kolekci, bez nutnosti specifikovat komparátor (implementující rozhraní java.util.Comparator).

Přirozené uspořádání pro třídu C je považováno za shodné s metodou equals pouze tehdy, když e1.compareTo((Object) e2) má stejnou logickou hodnotu jako e2.equals((Object) e1) pro všechny objekty e1 a e2 ve třídě C. Pro null platí, že není instancí žádné třídy a metoda e.compareTo(null) u některých implemenetací kolekcí vyhazuje výjimku NullPointerException, zatímco pro e.equals(null) vždy vrací false. Pravidlem je ale, metoda compare pro null by měla vždy vracet výjimku, zatímco equals výsledek false.

Generické rozhraní Comparable můžeme využívat několika způsoby: při deklaraci typového argumentu požadujeme, aby objekt typového parametru implementoval rozhraní Comparable, tj nabízel možnost vzájemného porovnání těchto prvků.
Příklad použití rozhraní Compaparable jako typového parametru:

```
LinkedList< Comparable> porovnatelny=new LinkedList<Comparable>();
```

Do seznamu s typovým argumentem Comparable potom můžeme vložit instanci jakékoliv třídy, která implementuje rozhraní Comparable.

Příklad implemetace metody compareTo pro třídu Koule porovnávající jednotlivé instance koule dle poloměru:

```
import java.util.*;
import java.lang.*;

public class Koule implements Comparable<Koule>
{
private double polomer;
public Koule(double p)
{
this.polomer=p;
}

public double getPolomer()
{
return polomer;
}

public double getObsah()
{
return 4/3d*Math.PI*Math.pow(polomer,3d);
}
public double getPovrch()
{
return 4d*Math.PI*Math.pow(polomer,2d);
}
public int compareTo(Koule k)
{
  return (int) Math.signum(k.getPolomer() - this.getPolomer());
}

}
```

Příklad generické metody s rozhraním Comparable najde maximum v kolekci s prvky implementujícími rozhraní Comparable.

```
import java.util.*;
import java.lang.*;

class Collections {
  public static <A extends Comparable<A>> A max(Collection< A> xs) {
    Iterator< A > xi = xs.iterator();
    A w = xi.next();
    while (xi.hasNext()) {
      A x = xi.next();
      if (w.compareTo(x) < 0) w = x;
```

```
    }
    return w;
  }
}
```

Je třeba podotknout, že ve třídě Collections je metoda max již implementována, a to s následující signaturami:

```
static  <T extends Object & Comparable<? super T>> T  max(Collection<? extends T> coll)
static <T> T  max(Collection<? extends T> coll, Comparator<? super T> comp)
```

U první signatury požadujeme, aby prvek v kolekci pro T nebo jeho podtyp, byl potomkem typu Object a měl předka implementujícího rozhraní Comparable (tj. měl k dispozici metodu compareTo).
U druhé signatury požadujeme, aby prvek v kolekci pro typ T a podtypy, a jeho nadtyp implementoval přiřazený Comparator.

Rozhraní **java.util.Comparator:**

```
public interface Comparator<E>
{
public boolean equals(E object);
}
```

Rozhraní s porovnávající funkcí equals(E), jež umožňuje uspořádané řazení objektů v implementující kolekci. Komparátory (porovnávací funkce) se využívají v metodě sort pro Collections.sort a její podtřídy pro seřazení prvků. Komparátory lze také použít na řazení určitých datových struktur (např. TreeSet nebo TreeMap. V tomto rozhraní jsou dvě metody: int compareTo(E obj1, E obj2) a equals(E obj), pro zjišťování vzájemného uspořádání objektů a zjišťování vzájemné identity.

V následujícím příkladu porovnáváme dle proměnné cas pro objekt Novinka, který obsahuje určitý text a k němu přiřazený čas. Tyto objekty chceme řadit podle času a za identitu považujeme situaci, kdy ukrývají shodný text. Všimněte si, že za parametr pro Comparable je dosazené jméno implementované třídy.

```
import java.util.GregorianCalendar;

public class Novinka implements Comparable<Novinka>
{
   // instance variables - replace the example below with your own
   private StringBuffer text;
   private GregorianCalendar cas;
   /**
    * Constructor for objects of class Novinka
    */
   public Novinka(StringBuffer text,int den, int mesic, int rok)
   {
      this.text=text;
      cas=new GregorianCalendar(rok,mesic,den);
   }
   public GregorianCalendar cas()
   {
      return cas;
   }

   /**
    * metoda porovnání času
```

```
 */
public int compareTo(Novinka os)
{
   final int BEFORE = -1;
   final int EQUAL = 0;
   final int AFTER = 1;
   if (this.cas().before(os.cas())) return BEFORE;
   else{
   if (this.cas()==os.cas()) return BEFORE;
   else return AFTER;}
}
boolean equals(Novinka obj)
{
  return this.text().equals(obj.text);
}

}
```

Tento objekt je připraven k vložení do libovolného seznamu implementujícího Comparator, aby s ním mohly být prováděny operace sort, tj setřídění vložených prvků.

11.3 Třída java.lang.Enum<E extends Enum<E>>

Jak rozumět výrazu "Enum<E extends Enum<E>>"?
Enum je abstraktní třída, což znamená, že nelze vytvářet její instance. Kompilátorem jsou vytvářeny pouze třídy jejího podtypu, přičemž tyto objekty získají jeho metody, z nichž některé používají jako argument svůj typ, příp. v závislosti na něm. Tvar deklarace třídy ve třídě Enum vypadá takto:

```
public abstract class Enum<E extends Enum<E>> {
  ...
}
```

Typ Enum je základní výchozí třída pro všechny výčtové typy V javě výčet typu s názvem Color je přeložen do tvaru „class Color extends Enum<Color>“. Účel nadtřídy Enum je nabízet funkce pro všechny výčtové typy.
Ukázka kódu třídy Enum:

```
public abstract class Enum< E extends Enum<E>> implements Comparable< E >,
Serializable {
  private final String name;
  public  final String name() { ... }

  private final int ordinal;
  public  final int ordinal() { ... }

  protected Enum(String name, int ordinal) { ... }

public String           toString() { ... }
  public final boolean    equals(Object other) { ... }
  public final int        hashCode() { ... }
  protected final Object  clone() throws CloneNotSupportedException { ... }
  public final int        compareTo( E o) { ... }

  public final Class< E > getDeclaringClass() { ... }
  public static <T extends Enum<T>> T valueOf(Class<T> enumType, String name) {
```

```
    ... }
}
```

Zarážející vlastností v deklaraci „Enum<E extends Enum<E>>" je fakt, že nově definovaná třída Enum a její nový parametr jsou ve mezi stejného typového parametru. Což značí, že Enum musí být instanciován pro jeden ze svých podtypů. Pokud chceme porozumět jeho účelu, uvažujme, že každý výčtový typ je přeložen do podtypu Enum.
Zde je ukázkový enum typ Color :

```
enum Color {RED, BLUE, GREEN}
```

Kompilátor toto přeloží do následujícího tvaru:

```
public final class Color extends Enum<Color> {
  public static final Color[] values() { return (Color[])$VALUES.clone(); }
  public static Color valueOf(String name) { ... }

  private Color(String s, int i) { super(s, i); }

  public static final Color RED;
  public static final Color BLUE;
  public static final Color GREEN;

  private static final Color $VALUES[];

  static {
    RED = new Color("RED", 0);
    BLUE = new Color("BLUE", 1);
    GREEN = new Color("GREEN", 2);
    $VALUES = (new Color[] { RED, BLUE, GREEN });
  }
}
```

Dědičnost způsobí, že typ Color zdědí veškeré metody implementované v Enum<Color>. Mezi nimi je i metoda compareTo. Metoda Color.compareTo by měla jako svůj argument přijmout objekt Color. Aby tohoto bylo dosaženo, třída Enum je definována jako generická a E je původní typový paranetr pro compareTo. Následně tedy typ Color odvozený od Enum<Color> zdědí metodu compareTo s objektem Color jako svůj argument. Pokud se podíváme blíže na deklaraci " Enum<E extends Enum<E>> ", můžeme vidět, že tento návrh má několik aspektů.

Zaprvé, je tu fakt, že mez typového parametru je generický typ Enum<E extends Enum<E>>. Což říká, že pouze podtypy typu Enum jsou povoleny jako typový argument. (Teoreticky lze typ Enum intancovat na sebe, ve smyslu Enum<Enum>, jenže takto to zcela jistě nebylo zamýšleno a je obtížné si představit situaci, ve které by takový objekt měl smysl.)

Zadruhé, je tu fakt, že mez typového parametru je generický typ Enum<E>, který používá typový parametr E jako typový parametr meze. Tato deklarace říká, že vztah dědičnosti mezi podtypem a instancí Enum je ve formě " X extends Enum<X> ". Podtyp " X extends Enum<Y> " nelze deklarovat, jelikož typový argument Y by nebyl mezi oběma mezemi, pouze podtypy Enum<X> jsou uvnitř meze.

Zatřetí, je tu fakt, že Enum je především generický. To znamená, že některé z metod třídy Enum mají argument nebo návratovou hodnotu neznámého typu (neboli závislou na neznámém typu). Jak již víme, tento neznámý typ bude později podtypem X pro Enum<X>. Proto v generickém typu Enum<X> tyto metody zahrnují podtyp X a jsou zděděny s podtypem

X. Například metoda compareTo, je zděděna ze své nadtřídy do každé podtřídy a má specifickou signaturu podtřídy pro každý typ.

Shrnuto, deklaraci "Enum<E extends Enum<E>>" lze chápat takto: Enum je generický typ, který lze instanciovat pouze na svých podtypech, a tyto podtypy zdědí jisté užitečné metody, z nichž některé používají podtyp jako specifický argument (nebo v závislosti na typu).

11.4 Rozhraní **java.lang.Iterable<E>,** java.util**.Iterator<E>**

Rozhraní Iterable má jedinou metodu Iterator(), vracející iterátor Iterator<E>.
Rozhraní Iterator implementuje dále následující tři metody:

```
Interface Iterator<E>
{
public booleand hasNext();
public E next();
public void remove();
}
```

Metoda hasNext() vrací true v případě, obsahuje-li implementující třída další prvek. Metoda next() vrací přímo tento prvek, a metoda remove() odstraní poslední prvek v kolekci. Implementace rozhraní Iterable umožňuje procházet prvky v implementující třídě pomocí metody hasNext() a cyklu.
Od verze 5 je i rozhraní Iterable generické, takže nový zápis iterace, pomocí konstruktoru Iterator<E> se zapisuje takto:

```
ArrayList<String> zvirata=new ArrayList<String>();
zvirata.add("osel");
zvirata.add("kráva");
zvirata.add("vepř");
Iterator<String> it=zvirata.iterator ();
String zvire;
while (it.hasNext())
{
        zvire=it.next();
        System.out.println(zvire);
}
```

bez použití parametrizace by vypadalo použití iterátoru takto:

```
Iterator it=zvirata.iterator();
String zvire;
while (it.hasNext}])
{
        zvire=(String) it.next();
        System.out.println(zvire);
}
```

Všimněte si, že při získání iterátoru it je konstruktor nově parametrizován typem String, díky čemuž odpadá přetypování u proměnné zvire.
Od verze 1.5 nabízí Iterátor procházet kolekci nejen pomocí tohoto cyklu, ale také i pomocí nového cyklu for-each.

```
ArrayList<String> zvirata=new ArrayList<String>;
zvirata.add(„osel")
zvirata.add(„kráva")
zvirata.add(„vepř");
for(String z: zvirata)
{
System.out.println(z);
```

```
}
```

Ovšem je nutné podotknout, že i při použití tohoto cyklu for je použito metody iterator(). Její volání je již však jen záležitostí překladače, a pokud provedeme debugging cyklu for, zjistíme, že překladač vytvořil pro cyklus for pomocnou proměnnou typu Iterator, se jménem i$. Prvky se pak čtou z této proměnné, nikoliv přímo z objektu zvirata, jak by se mohlo na první pohled zdát.

Představme si, že chceme procházet seznam LinkedList obsahující instance typu Data.
Toto můžeme provést pomocí parametru metody jako Iterable<Data>:

```
public Collection<Data> processData(Iterable<Data> data) {
List<Data> result = new ArrayList<Data>();
for (Data d:data)
{
            prozkoumejAzpracuj(data);

}
  return result;
}
```

Tímto říkáme, že budeme typ Data v seznamu data pouze iterovat a že jej nebudeme nijak měnit.
(použito z http://blog.krecan.net/2008/03/14/list-nebo-iterable/ 29.11. 2008)

Lze deklarovat i vlastní třídu, implementující rozhraní Iterable:

```
import java.util.*;

class StrIterable implements Iterable<Character>,
                             Iterator<Character> {
  private String str;
  private int count = 0;

  StrIterable(String s) {
    str = s;
  }

  // Následující tři metody implementují iterátor .
  public boolean hasNext() {
    if(count < str.length()) return true;
    return false;
  }

  public Character next() {
    if(count == str.length())
      throw new NoSuchElementException();

    count++;
    return str.charAt(count-1);
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }

  // Tato metoda implementuje Iterable.
  public Iterator<Character> iterator() {
```

```
    return this;
  }
}

public class MainClass {
  public static void main(String args[]) {
    StrIterable x = new StrIterable("Toto je test.");

    for(char ch : x)
      System.out.print(ch);

    System.out.println();
  }
}
```

Ukázka z http://www.java2s.com/Code/Java/Language-Basics/UsingaforeachforeachforlooponanIterableobject.htm (29.11.2008)

Třída StrIterable je jednoduchá třída, která obsahuje jeden textový řetězec String a provádí iteraci po jednotlivých písmenech. Dále si můžete všimnout, že u metody next je místo neurčeného generického typu Character.

Rozhraní **java.util.Iterator<E>**

Novější rozhraní určené na procházení (iteraci) kolekce. Iterator nahrazuje a rozšiřuje rozhraní Enumeration v kolekci Java knihoven. Liší se několika věcmi: jak už bylo řečeno, Iterator umožňuje odstranit prvek z kolekce během iterace, a byla zkrácena jména stejných metod (viz. Enumeration).

Příklad využití rozhraní Iterator:

```
import java.util.*;

public class Iterated
{
   public static void main(String args[])
   {
      ArrayList<String> ar=new ArrayList<String>();
      ar.add("Hynek");
      ar.add("Jarmila");
      ar.add("Vilém");
      String slovo;
        //toto je, krom parametrizace Iteratoru, původní způsob iterace kolekce
        Iterator<String> it = ar.iterator();
        while ( it.hasNext() )
        {
                slovo = it.next();  // Není nutné přetypování, provedeno díky parametrizaci.
                System.out.println(slovo);
        }
        //Nově lze ve verzi Java 1.5 procházet kolekci i cyklem for:
          for (String mojeSlovo : ar )
        {
                  System.out.println(mojeSlovo);
        }

    }
}
```

Zde je využit i nový cyklus for.

11.5 Rozhraní java.util.Collection<E>, rozšiřující rozhraní a implementující třídy

Rozhraní Collection rozšiřuje rozhraní Iterable. Jedná se o nejvyšší rozhraní v hierarchii kolekcí, od nějž dědí všechny kolekce zde (v rozhraní Collection). Již zde jsou hlavičky všech metod pro vkládání, čtení a mazání prvků, v závislosti na typu kolekce. Pokud nějaká metoda není u nějaké implementující třídy podporována, po zavolání dojde k vyvolání výjimky UnsupportedOpperationException.

Od verze Java 1.5 jsou zde metody pro práci s prvky parametrizovány, díky čemuž odpadá nutnost přetypování prvku při výběru z kolekce.

Rozhraní **java.util.Enumeration<E>**

Toto rozhraní sice nespadá přímo pod rozhraní Collection, ale jeho metody jsou využívány také především při práci s kolekcemi. Metody tohoto rozhraní fungují obdobně jako rozhraní Iterator, vývojově je ovšem o něco starší. Enumeration byla v Javě od verze 1.0, zatímco rozhraní Iterator se objevilo až ve verzi 1.2. Navíc Iterator přidává operaci remove(), a používá kratší názvy metod.(Enumeration: hasMoreElements(), nextElement(); Iterator: hasNext() , next()). Obecně se doporučuje použití rozhraní Iterator před Enumeration.

Třída implementující rozhraní Enumeration umožňuje postupné vyjmenovávání jednotlivých prvků. Volání metody nextElement() vrací postupně jednotlivé prvky množiny (při prvním volání vrátí první prvek, při druhém druhý až nakonec vyvolá výjimku NoSuchElementException, není-li žádný další prvek k dispozici. Tomu můžeme předejít používáním metody hasMoreElements().

Například, k vypsání prvků objektu vector *v*:

```
for (Enumeration<String> e = v.elements() ; e.hasMoreElements() ;)
{
      System.out.println(e.nextElement());
}
```

Rozhraní Enumeration implementuje třída StringTokenizer, třídy Vector<E>, HashTable<K,V>, HasMap<K,V>, používají metodu elements() vracející referenci na rozhraní Enumeration.

Metoda elements() umožňují výčet prvků vektoru, klíčů a hodnot v HashTable, v hodnotách HashMap, a v jakékoliv další kolekci používající rozhraní Enumeration. Enumeration je také použito v konstruktoru k specifikaci vstupního proudu pro SequenceInputStream jako výčtu vstupních proudů InputStream. Rozhraní Enumeration bylo zachováno v knihovnách JDK pouze z důvodu zpětné kompatibility.

Rozhraní **java.util.List<E>**

List je rozhraní rozšiřující rozhraní Collection a Iterable. Třída implementující rozhraní List ja uspořádaná kolekce (také známá pod názvem sekvence). Uživatel tohoto rozhraní má plnou kontrolu nad tím, kam bude vložen každý prvek. Uživatel může přistupovat k prvkům pomocí čísla jejich pozice a hledat prvky v seznamu. Jako všechny kolekce ve verzi 1.5, všechny kolekce odvozené od tohoto rozhraní jsou generické.

Rozhraní List má speciální iterator, nazývaný ListIterator, který umožňuje vkládání a nahrazování prvků a obousměrný přístup v porovnání s operacemi, které nabízí rozhraní Iterator. Nabízí i metodu získání iteratoru na určité pozici.
Příklad deklarace List:

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

Rozhraní List implementují např. třídy AbstractList, ArrayList a LinkedList.

Rozhraní **java.util.ListIterator<E>**

Iterátor pro seznamy umožňující programátoru-uživateli procházet seznamem typu List oběma směry, měnit obsah během iterace a získat aktuální pozici iteratoru. ListIterator nemá aktuální vlastní prvek, je to ukazatel mezi prvky, který je posouván metodami previous() na předchozí a next() na následující prvek. V seznamu délky n je zde n+1 počet ukazatelů od 0 do n včetně.

```
           Provek(0)    prvek(1)   prvek(2)   ... prvek(n)
          ^            ^           ^           ^            ^
 Index: 0              1           2           3            n+1
```

Všimněte si, že metody remove() a set(E et) nepoužívají pozici ukazatele, ale pracují s prvkem, který byl naposledy vrácen pomocí metod next(), nextElement() nebo previous() či previousElement().

Rozhraní java.util.**Queue<E>**

I toto je rozhraní implementující generickou kolekci java.util.PriorityQueue. Kolekce navržená na prvky pro průběžné zpracování. Kromě základních operacích pro Collection, fronty nabízejí dodatečné vložení, extrakci, a prohlížení.
Fronty obvykle řadí svoje prvky systémem FIFO (první dovnitř, první ven) . Mimo výjimek existují prioritní řady, které řadí prvky podle určeného komparátoru, nebo přirozeného řazení prvků (třída java.util.PriorityQueue a **java.util.concurrent.PriorityBlockingQueue)**, příp. systémem LIFO (poslední dovnitř, první ven) . Při jakémkoliv způsobu řazení je vždy hlava, čili první člen fronty odstraněn metodou remove() nebo poll(). Ve frontě FIFO jsou všechny nové prvky vkládány na konec fronty. Jiné druhy fronty mohou mít odlišná pravidla, například LIFO je vkládá na začátek. Každá implementace Queue musí specifikovat svůj způsob řazení.

Implementace Queue nedovoluje implicitně vkládat prvky typu null, ačkoli něketré jiné, jako třeba LinkedList, toto nezakazují. Dokonce ani v implementaci, která to dovoluje, bychom neměli vkládat prvky typu null, jelikož ta je zpravidla používána jako speciální návratová hodnota indikující, že fronta je prázdná.

Rozhraní **java.util.Set<E>**

Speciální typ kolekce odvozené od rozhraní Collection. Od tohoto rozhraní jsou odvozeny podrozhraní NavigableSet<E> a SortedSet<E>. Kolekce tohoto typu neobsahuje duplicitní prvky, a může mít nejvýše jeden prvek null. Jak naznačuje její jméno, jeho rozhraní modeluje matematickou abstrakci množiny.

Některé implementace pro Set mají omezení na typ prvků, které mohou obsahovat. Některé implementace zakazují použití prvku null, pokus o vložení nežádoucího prvku může tedy skončit vyvoláním výjimky, typicky NullPointerException nebo ClassCastException při pokusu o vložení zakázaného typu. Pokus o dotaz na nežádoucí prvek může vyvolat jak výjimku tak i vrátit hodnotu false - v závislosti na typu prvku, liší se dle implementace. Výjimky tohoto typu se nazývají v dokumentaci 'optional'.

Rozhraní **java.util.SortedSet<E>**

Rozhraní zaručující, že jeho iterátor bude procházet prvky ve vzestupném pořadí, které jsou uspořádány v přirozeném pořadí svých prvků (viz. Comparable), nebo dle komparátoru přiřazeném při jeho vytvoření. Je také k dispozici několik dodatečných metod pro využití tříděného setu (analogicky s rozhraním SortedMap.)

Všechny vložené prvky musítedy implementovat rozhraní Comparable<E> (nebo spolupracovat s přiřazeným komparátorem). Dále, všechny prvky musí být vzájemně porovnatelné: e1.compareTo(e2) (nebo comparator.compare(e1, e2)) nesmí vyvolat výjimku ClassCastException pro jakýkoliv prvek e1 a e2 ve tříděné množině. Pokus o obejití tohoto omezení vyvolá u příslušné metody ClassCastException.

Třídy **AbstractCollection**, **AbstractList**, **AbstractMap**, **AbstractQueue**, **AbstractSequentialList**, **AbstractSet**

Abstraktní třídy implementující všechny základní metody pro Collection, List, Map, Queue, SequentialList a Set.

Třída **java.util.ArrayList<E>**

Implementace pole s měnitelnou velikostí implementující rozhraní List. Implementuje všechny volitelné operace a dovoluje vkládat všechny prvky včetně null. Oproti zděděným metodám z rozhraní List má schopnost navíc měnit svoji velikost - pomocí metody trimToSize() zmenší kapacitu na aktuální počet prvků a EnsureCapacity(int size) – na velikost zadanou parametrem size (je podobná třídě Vector, krom toho že není synchronizovaná).

Každý ArrayList má svou kapacitu. Kapacita je velikost pole použitelná k ukládání prvků. Kapacita je vždy rovna nebo větší počtu prvků. při každém přidání prvku se automaticky zvýší kapacita, rychlost zvětšování má konstantní hodnotu. Aplikace může zvýšit kapacitu ArrayListu pomocí EnsureCapacity před přidáním velkého množství prvků, což sníží realokaci prvků a zvýší rychlost jejich přidávání.

Tato implementace není synchronizována. Pro zabezpečení synchronizace můžeme použít metodu Collections.synchronizedList, k zabránění nesychronizovanému přístupu.

```
List<Parametr> list = Collections.synchronizedList(new ArrayList<Parametr>([...]));
```

Třída **java.util.Dictionary<K,V>**

Třída Dictionary je abstraktním rodičem pro jakoukoliv třídu, např. HashTable, jež mapuje klíče a hodnoty. Každý klíč a hodnota je objekt v jakémkoliv objektu Dictionary, každý klíč je asociován s jednou hodnotou. pro jakýkoliv objekt Dictionary a klíč lze hledat hodnotu, a klíč ani hodnota nesmí být typu null.
Dle dokumentace se jedná o zastaralou abstraktní třídu, doporučuje se používat rozhraní Map.

Třída **java.util.EnumSet<E extends Enum<E>>**

Specializovaná implementace Set pro použití s výčtovými typy. Veškeré prvky ve výčtové množině musí být stejného typu, jenž je explicitně nebo implicitně určen při vytvoření objektu. Výčtové množiny jsou interně reprezentovány jako bitové vektory.

Při iteraci vrací prvky seřazené v jejich přirozeném pořadí.

Jako mnoho kolekcí, EnumSet není synchronizován. Pro zabezpečení synchronizace můžeme použít zapouzdřující metodu Collections.synchronizedSet(java.util.set) pro zabránění nesychronizovaného přístupu.

```
Set<MyEnum> s = Collections.synchronizedSet(EnumSet.noneOf(Foo.class));
```

Třída **java.util.HashSet<E>**

Tato třída implementuje rozhraní Set, využívající hash tabulku (konkrétně instanci třídy HashMap). Nezaruručuje zachování pořadí prvků během iterace v kolekci, a nezaručuje ani, že tabulka zůstane během čtení konstantní. Tato třída povoluje prvky typu null.

Pokud je důležitý výkon, je doporučno nastavit výchozí kapacitu kolekce ne příliš vysokou, nebo faktor zaplnění moc nízký.

Tato implementace není synchronizovaná. Pro zabezpečení synchronizace můžeme použít metodu Collections.synchronizedSet, k zabránění nesychronizovanému přístupu.

```
Set<Parametr> m = Collections.synchronizedSet(new HashSet([...]));
```

Třída **java.util.LinkedHashSet<E>**

Třída LinkedHashSet je implementace rozhraní Set, s předvídatelním pořadí iterace.

Tato implementace se liší od HashSet tím, že udržuje obousměrný spojový seznam pro všechny své prvky. spojový seznam definuje pořadí iterace dle toho, v jakém pořadí byly položky vloženy. Pořadí vkládání neovlivní znovuvložení stejného prvku (Prvek je opětovně vložen, pokud je zavoláno s.add(prvek), pokud s.contains(prvek) vrátí true před provedením příkazu.)

Tato třída nabízí veškeré volitelné operace pro Set a povoluje prvky typu null. Stejně jako HashSet nabízí tato třída provedení základních operací (add, contains a remove) v konstantním čase za předpokladu že hash funkce správně rozmístí prvky do slotů.

Výkon je poněkud nižší než u HashSet, kvůli údržbě spojového seznamu, s jednou výjimkou: Iterace kolekcí potřebuje čas v závislosti na počtu prvků v HashSet, nezávisle na jeho konečné kapacitě.

Konstruktor třídy LinkedHashSet má dva parametry, jenž ovlivňují její výkon: výchozí kapacita a loadfactor. Jsou definovány shodně jako pro HashSet. Poznámka: postih za zvolení příliš velké kapacity je poněkud nižší než u HashSet, jelikož iteraci neovlivňuje kapacita.

Tato implementace není synchronizovaná. pro obalující třídu můžeme použít metodu Collections.synchronizedSet , k zabránění nesychronizovanému přístupu.

```
Set<Parametr> s = Collections.synchronizedSet(new LinkedHashSet([...]));
```

Třída java.util.**LinkedList<E>**

Třída LinkedList je implementací rozhraní List, Deque, Cloneable a java.io.Serializable. Nabízí veškeré volitelné operace pro seznam, a dovoluje všechny prvky včetně null. Navíc k metodám implementovaným z rozhraní List, nabízí i metody umožňující vložení a odstranění prvku na konec a začátku seznamu. Tyto operace dovolují spojovému seznamu využití i jako zásobníku (stack), fronty (queue) nebo obousměrné fronty (double-ended queue). Třída implementuje i rozhraní Deque, nabízející operaci add první-dovnitř-první-ven (FIFO), dále poll, a další. (Další funkce pro stack a deque lze sestavit ze standardních funkcí seznamu, nejsou-ji liž v této třídě implementovány.)
Veškeré tyto operace fungují tak, jak lze očekávat pro obousměrný seznamu. Operace, jež pracují se seznamem, vstupují buď od začátku nebo od konce, podle toho, je-li index prvku blíž konci nebo začátku.
Tato implementace není synchronizovaná. Pro obalující třídu můžeme použít metodu Collections.synchronizedList, k zabránění nesychronizovanému přístupu.

```
List<Parametr> list = Collections.synchronizedList(new LinkedList([...]));
```

Třída **java.util.Stack<E>**

Třída Stack reprezentuje zásobník LIFO (Last-In-First-Out). Rozšiřuje třídu Vector pěti operacemi, které dovolují zacházet s vektorem jako zásobníkem. Umožňuje obvyklé operace push a pop, dále peek pro nahlédnutí na vrchol zásobníku, metoda isEmpty() ke zjištění, je-li zásobník prázdný a metoda search(Object e) k prohledání zásobníku pro zjištění přítomnosti určitého prvku, kde v případě nalezení tato metoda vrátí index prvku. Není-li prvek přítomen, metoda vrací –1. Pro lepší práci se zásobníkem typu LIFO lze doporučit spíše třídu Deque, která nabízí daleko více funkcí pro práci s prvky.

Třída java.util.**TreeSet<E>**

Tato třída implementuje rozhraní NavigableSet, založené na třídě TreeMap. Tato třída zaručuje, že tříděná množina bude ve vzestupném pořadí, řazeném dle přirozeného řazení prvků (dle Comparable), nebo dle komparátoru přiřazeného při vytvoření dle použitého konstrukrotu.
Tato implementace nabízí časovou náročnost log(n) pro n prvků se základními operacemi (add, remove a contains).
Dále řazení v setu musí být konzistentní s metodou equals (nezávisle na tom, zda-li byl přidělen příslušný komparátor) pro správnou implementaci rozhraní.
Tato implementace není synchronizovaná. Pro obalující třídu můžeme použít metodu Collections.synchronizedSet , k zabránění nesychronizovanému přístupu.

```
SortedSet<Parametr> s = Collections.synchronizedSortedSet(new TreeSet([...]));
```

Třída java.util.**Vector<E>**

Třída Vector implementuje dynamické pole objektů. Podobně jako pole obsahuje prvky, k nimž lze přistupovat pomocí číselného indexu. Samozřejmě, ale velikost objektu Vector se může zmenšovat nebo zvětšovat v závislosti na tom, jak jsou prvky přidávány nebo mazány.
Každý Vector se snaží přizpůsobit řízení paměti udržováním proměnné kapacity a capacityIncrement. Kapacita je nejméně tak velká jako je velikost Vectoru, obvykle je větší,

protože jak jsou prvky přidávány, prostor pro Vector se zvětšuje po násobcích dle capacityIncrement. Aplikace může zvětšit kapacitu vektoru před vložením velkého množství prvků, což zmenšuje množství realokace paměti.
Od verze Java platformy 1.2, tato třída byla přepsána na implemenetaci pro List, aby se stala součástí Java Collection Framework. Oproti novým implementacím Collection, tato třída je synchronizovaná.

11.6 Rozhraní java.util.Map<E>, rozšiřující rozhraní a implementující třídy

Rozhraní **java.util.Map<K,V>**

Množina dvojic klíčů a hodnot. Kolekce Map nemůže obsahovat duplicitní klíče; každý klíč může odkazovat nejvýše na jednu hodnotu. Rozhraní Map nabízí tři pohledy na kolekci, které umožňují vidět kolekci jako množinu klíčů, kolekci hodnot, nebo množinu dvojic klíč-hodnota. Pořadí v mapě je shodné jako pořadí, ve kterém vrací iterátor jejich prvky.

Generický návrh rozhraní Map umožňuje vymezit předem typy klíčů i hodnot, které bude používat. Některé implementace mapy, např. třída TreeMap, specifikují určité pořadí, jiné, např. HashMap, toto nespecifikují.

Příklad implementace rozhraní Map:
Následující příklad ukazuje implementaci rozhraní Map ve třídě MyMap, kde množina klíčů je uložena v seznamu typu ArrayList s názvem keys, a kolekce hodnot v jiném ArrayListu s názvem values. Propojení těchto seznamů je v metodách getKey a getValue, hodnoty a klíče jsou získávány pomocí shodného pořadí v obou seznamech. Vnořená třída MyMapEntry ukazuje implementaci metod getKey, getValue, setValue a dalších.

```
/*
keys: [Learning Tree, IBM, Google, Adobe]
You asked about Google.
They are located in: null

Key Adobe; Value Mountain View, CA
Key IBM; Value White Plains, NY
Key Learning Tree; Value Los Angeles, CA
entrySet() returns 3 Map.Entry's*/

import java.util.AbstractSet;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.TreeSet;

public class MainClass {

  public static void main(String[] argv) {
    MyMap<String,String> map = new MyMap<String,String>();

    map.put("Adobe", "Mountain View, CA");
    map.put("Learning Tree", "Los Angeles, CA");
    map.put("IBM", "White Plains, NY");
    map.put("Google", "It's a secret, N/A");
```

```
    System.out.println("keys: " +   map.getListKeys());

    String queryString = "Google";
    System.out.println("You asked about " + queryString + ".");
    String resultString = map.get(queryString);
    System.out.println("They are located in: " + resultString);
    System.out.println();

    Iterator<String> k = map.keySet().iterator();
    while (k.hasNext()) {
      String key = k.next();
      System.out.println("Key: " + key + "; Value " +  map.get(key));
    }

    Set es = map.entrySet();
    System.out.println("entrySet() returns " + es.size() + " Map.Entry's" );
  }
}

class MyMap<K extends Comparable<K>,V> implements Map<K,V> {

  private ArrayList<K> keys;

  private ArrayList<V> values;

  public MyMap() {
    keys = new ArrayList<K>();
    values = new ArrayList<V>();
  }

  public ArrayList<K> getListKeys()
  {
      return keys;
    }

    public ArrayList<V> getListValues()
  {
      return values;
    }

  /** Vrátí počet dvojic v Map. */
  public int size() {
    return keys.size();
  }

  /** Vrátí true je-li mapa prázdná. */
  public boolean isEmpty() {
    return size() == 0;
  }

  /** Vrátí true pokud o je klíčem v této Mapě. */
  public boolean containsKey(Object o) {
    return keys.contains(o);
  }

   /** Vrátí true pokud o je hodnotou v této Mapě. */
  public boolean containsValue(Object o) {
    return values.contains(o);
  }
```

```
  /**Získá hodnotu ke klíči k. */
  public V get(Object k) {
    int i = keys.indexOf(k);
    if (i == -1)
      return null;
    return values.get(i);
  }

  /**
   * Vloží zadanou dvojici (k, v) do této mapy udržováním klíči v řazeném pořadí.
   */
  public V put(K k, V v) {
    for (int i = 0; i < keys.size(); i++) {
      K old = keys.get(i);

      /* Existuje už klíč? */
      if (k.compareTo(old) == 0) {
          V oldValue = values.get(i);
//        keys.set(i, k);
        values.set(i,v);
        return oldValue;
      }

      /**
*Došli jsme k místu, kam to vložit? t. j. , udržuj klíčem v sestupném pořadí .
       */
      if (k.compareTo(old) > 0) {
        int where = i;
        keys.add(where, k);
        values.add(where, v);
        return null;
      }
    }
    // Jinak to dej nakonec.
    keys.add(k);
    values.add(v);
    return null;
  }
  /**
* Vložit všechny dvojice z oldMap do této mapy
   */
  public void putAll(java.util.Map<? extends K,? extends V> oldMap) {
    Iterator<? extends K> keysIter = oldMap.keySet().iterator();
    while (keysIter.hasNext()) {
      K k = keysIter.next();
      V v = oldMap.get(k);
      put(k, v);
    }
  }

  public V remove(Object k) {
    int i = keys.indexOf(k);
    if (i == -1)
      return null;
    V old = values.get(i);
    keys.remove(i);
    values.remove(i);
    return old;
  }
```

```
public void clear() {
  keys.clear();
  values.clear();
}

public Set<K> keySet() {
  return new TreeSet<K>(keys);
}

public Collection<V> values() {
  return values;
}

/**
 * Objekty Map.Entry obsažené v Setu vráceném  z metody entrySet().
 */
private class MyMapEntry<K extends Comparable<K>,V> implements Map.Entry<K,V>,
Comparable <Map.Entry<K,V> > {
  private K key;
  private V value;

  MyMapEntry(K k, V v) {
    key = k;
    value = v;
  }

  public K getKey() {
    return key;
  }

  public V getValue() {
    return value;
  }

  public V setValue(V nv) {
    throw new UnsupportedOperationException("setValue");
  }
  public int compareTo(Map.Entry<K,V> o2) {
     return key.compareTo(o2.getKey());
  }
}
/** Množina  Map.Entry objektů z metody entrySet(). */
private class MyMapSet extends AbstractSet<Map.Entry<K,V>> {
  List<Map.Entry<K,V>> list;

  MyMapSet(ArrayList<Map.Entry<K,V>> al) {
    list = al;
  }

  public Iterator<Map.Entry<K,V>> iterator() {
    return list.iterator();
  }

  public int size() {
    return list.size();
  }
}

/**
 *Vrací pohledvou množinu pro seznam Map, každý prvek v této množině je MyMap.Entry.
```

```
 *Není zcela zaručena shodná implementovace s entrySet() v java.utilMap
 */
public Set<Map.Entry<K,V>> entrySet() {
  if (keys.size() != values.size())
    throw new IllegalStateException("InternalError: keys and values out of sync");
  ArrayList<Map.Entry<K,V>> al = new ArrayList<Map.Entry<K,V>>();
  for (int i = 0; i < keys.size(); i++) {
    al.add(new MyMapEntry<K,V>(keys.get(i), values.get(i)));
  }
  return new MyMapSet(al);
 }
}
```

Příklad byl převzat ze stránek Java Sun a rozšířen o generické prvky.

(http://www.java2s.com/Code/JavaAPI/java.util/implementsMap.htm)

Rozhraní **java.util.Map.Entry<K,V>**

Záznamy mapy (dvojice klíč-hodnota). Metoda Map.entrySet() vrací kolekci záznamů jako pohled kolekce. Jediný způsob získání referenci k objektu typu Map.Entry je během procesu iterace této kolekce. Tyto objekty jsou platné pouze během procesu iterace, a dokud se pokud původní mapa nezmění během iterace. Jediná výjimka je metoda setValue() pracující se záznamem Value, kterou se platnost objektu Map.Entry nezmění. Příklad implementace je uveden výše, objekty tohoto typu lze také získat pomocí typového parametru.

Třída **java.util.EnumMap<K extends Enum<K>,V>**

Speciální implementace Map pro práci s klíči typu Enum. Veškeré klíče v mapě musí být stejného enum typu, který je specifikován explicitně nebo implicitně při vytvoření mapy. Enum mapy ukládají v přirozeném pořadí ejich klíčů. To je patrné z iterátorů získaných pomocí keySet(), entrySet() a values().
Jako mnoho implementací Collection, i tato není synchronizovaná. Pro obalující třídu můžeme použít metodu Collections.synchronizedMap(), k zabránění nesychronizovanému přístupu.

```
Map<EnumKey, V> m = Collections.synchronizedMap(new EnumMap([...]));
```

Rozhraní java.util.**SortedMap<K,V>**

Mapa, v níž jsou klíče uspořádány ve vzestupném pořadí, dle způsobu přirozeného řazení prvků (viz. rozhraní Comparable), nebo dle komparátoru (viz rozhraní Comparator) při inicializaci mapy. Toto pořadí je zohledněno při iteraci kolekce tříděné mapy (získané pomocí metod entrySet, keySet a values), navíc nabízí několik přídavných metod pro využití tříděnosti. (Toto rozhraní je analogické pro rozhraní SortedSet)

Všechny vložené klíče musí implementovat rozhraní Comparable (nebo přijímat specifikovaný komparátor). Dále, všechny klíče musí být vzájemně porovnatelné, čili k1.compareTo(k2) (nebo comparator.compare(k1, k2)) nesmí vyvolat ClassCastException pro žádný z klíčů k1 ani k2. Jakýkoliv pokus ignorovat toto omezení vyvolá ClassCastException.

Třída **java.util.HashMap<K,V>**

HashMap je tabulka založená na implementaci rozhraní Map. Tato implementace nabízí veškeré volitelné operace pro Map, a povoluje null hodnoty a klíče (Třída HashMap je podobná Hashtable, krom toho, že je nesynchronizovaná a povoluje null). Tato třída neprovádí řazení pořadí prvků a ani nezaručuje, že se nezmění jejich pořadí během vkládání.

Tato implementace nabízí časově konstantní výkon pro základní operace (get a put) za předpokladu, že hash funkce správně rozčlení prvky do slotů. Iterace kolekcí vyžaduje čas úměrný ke „kapacitě" instance HashMap (počtu slotů) plus počtu dvojic klíč-hodnota. Tedy je důležité nenastavit výchozí kapacitu HashMap příliš vysokou (nebo loadfactor příliš vysoký) pokud je důležitý výkon během iterace.

Instance HashMap má dva parametry ovlivňující její výkon: výchozí kapacita a loadfactor. Kapacitou je počet slotů v HashMap a výchozí kapacita je kapacita při vytvoření tabulky. Loadfactor je míra, při které je tabulka považována za plnou, a má být automaticky zvýšena její kapacita. Pokud počet vstupů převýší loadfactor a je vyčerpána kapacita tabulky, je její kapacita zhruba zdvojnásobena zavoláním metody rehash().

Jako výchozí pravidlo, výchozí loadfactor (0.75) nabízí dobrý kompromis mezi časem a kapacitou. Vyšší hodnoty snižují režii místa, ale zvyšují doby vyhledávání. Při vytváření tabulky by se mělo vzít na vědomí nastavení výchozí kapacity pro snížení počtu volání metody rehash(). Pokud je výchozí kapacita větší než počet vstupů dělená loadfactorem , metoda rehash() nebude vůbec zavolána.

Pokud bude v tabulce uloženo velké množství prvků, vytvořit ji předem dostatečně velkou je lepší, než neustálé zvyšování její kapacity.
Tato implementace není synchronizovaná. Pro zabezpečení synchronizace konkurenčního přístupu můžeme použít metodu Collections.synchronizedMap:

```
Map<Key,Value> m = Collections.synchronizedMap(new HashMap([...]));
```

Třída **java.util.IdentityHashMap<K,V>**

IdentityHashMap je implementace Map využívající HashMap. Vyznačuje se tím, že testuje rovnost pomocí reference místo hodnoty. Klíče a hodnoty jsou porovnávány na shodu pomocí referencí místo volání funkce equals() pro klíče a hodnoty. IdentityHashMap používá otevřené adresování (v podstatě lineární průchod) pro řešení kolizí. Tímto se liší od HashMap, která používá zřetězení. Podobně jako HashMap, IdentityHashMap není vláknově bezpečná, čili přístup pro více vláken musí být řízen externím mechanismem - nejčastěji pomocí Collections.sychronizedMap().

Tato třída je užitečná k uchovávání topologie objektových transformací grafů, například k serializaci nebo hloubkovému kopírování. K provádění těchto transformací potřebujeme udržovat uzlovou tabulku, jež uchovává informace o referencích na objektech, kterými již prošla, a nesmí zaměňovat shodné prvky na různých místech grafu. Mapy založené na identitě jsou také použity k provádění mapování meta-informací na objekty v debuggerech a podobných systémech. Dále mapy tohoto typu jsou užitečné k odhalení maskovaných útoků „spoof attacks" (vstup do systému s falešnou identitou za účelem získání neoprávněných výhod), které jsou výsledkem úmyslně nesprávně navržených implementací metody equals(), jelikož IdentityHashMap nikdy nevolá metodu equals() na své klíče (místo nich používá na klíče ==).I tato implementace HashMap byla přepsána do nové verze jako generická.

Třída java.util.**LinkedHashMap<K,V>**

Implementace rozhraní Map využívající hash tabulky a spojového seznamu s předvídatelným pořadím iterace. Tato implementace se liší od HashMap tím, že udržuje dvojitě spojový seznam skrz veškeré jeho vstupy. Tento spojový seznam definuje pořadí iterace, což je normální pořadí, ve kterém byly vkládány klíče do mapy. Opakované vložení klíče nemá vliv na jeho pořadí (Klíč je opětovně vložen do mapy m pokud m.put(k, v) a m.containsKey(k) vrací true těsně před voláním).

Je k dispozici i speciální konstruktor k vytvoření mapy, jejíž průběh iterace začíná od nejvíce navštěvovaných prvků. Tento druh mapy se výborně hodí ke stavbě LRU cache. (Least Recently Used – pamět s omezenou kapacitou, která při zaplnění vyřazuje nejméně používané prvky. Využívá se např. v databázových systémech nebo vyrovnávacích pamětích). Tato implementace není synchronizovaná: pro obalující třídu můžeme použít metodu Collections.synchronizedMap, která zabrání nesychronizovanému přístupu.

```
Map<Parametr> m = Collections.synchronizedMap(new LinkedHashMap<K,V>([...]));
```

<u>Třída **java.util.TreeMap<K,V>**</u>

Strom typu R/B – implementace rozhraní NavigableMap. Tato třída ukládá prvky dle přirozeného uspořádání klíčů (pomocí rozhraní Comparable) nebo dle přiřazeného komparátoru, v závislosti na použitém konstruktoru.

Časová náročnost pro tuto implementaci je log(n) pro operace containsKey, get, put a remove. Použité algoritmy jsou adaptace těch, které byly použity ve spisu „*Introduction to Algorithms*“ od autorů Cormen, Leiserson a Rivest.

Pro správnou funkčnost této třídy musí mít jeho prvky správně implementovanou metodu equals a compareTo, pokud má tato třída správně fungovat na rozhraní Map. Tato implementace není synchronizovaná. pro obalující třídu můžeme použít metodu Collections.synchronizedMap , k zabránění nesychronizovanému přístupu.

```
Map<Parametr> m = Collections.synchronizedMap(new TreeMap<K,V>([...]));
```

11.7 Balíčky java.util.concurrent, java.util.concurrent.atomic

Další parametrizované třídy a rozhraní obsahují i balíčky **java.util.concurrent** a **java.util.concurrent.atomic**, které obsahují speciální kolekce uzpůsobené pro konkurenční přístup při multithreadingu(ConcurrentHashMap, ConcurrentLinkedQueue, ConcurrentSkipListMap, ConcurrentSkipListSet), práci s vlákny (ThreadPoolExecutor a podtřídy) a řad typu Queue, konkrétně ConcurentLinkedQueue. Jejich popis se však už neliší od principu jejich předchozích, negenerických verzí tříd a popis jejich problematiky není účelem této práce.

12. Závěr

12.1 Jaké jsou výhody parametrizace generických typů?

Použití parametrizovaných typů místo neparametrizovaných má mnohé výhody a je doporučeno, pokud nemáte pádný důvod upřednostnit neparametrizovaný typ, nejčastěji kvůli kompatibilitě s nižší verzí než Java 5.

Je stále dovoleno použít generické typy bez typových argumentů, čili v jejich syrové formě. V podstatě můžete ignorovat Java genericitu a použít syrové typy v celém programu. Zde je ale doporučeno použít typové argumenty, kdykoliv je použit generický typ, dokud ovšem nemáte pádný důvod se jim vyhnout.

Využití typových argumentů s generickými typy a metodami místo použití syrových typů má několik výhod:

- **Vylepšená čitelnost:** instance s typovým argumentem je více informativní a zlepšuje čitelnost zdrojového kódu. Sděluje především, s jakým datovým typem bude generický typ nebo metoda operovat.
- **Lepší podpora nástrojů:** použití typových argumentů umožňuje vývojářským nástrojům vás efektivněji podpořit: IDEs (=integrated development environments) mohou nabídnout přesnější informace k obsahu dat, kompilátory mohou označit chyby v momentě, kdy vložíte chybný kód. Bez použití typových argumentů mohou zůstat chyby v programu skryté až do okamžiku spuštění a testování programu

- **Méně chyb při přetypování (ClassCastExceptions).** Typové argumenty umožňují kompilátoru provést typovou kontrolu pro zajištění typové bezpečnosti již při kompilaci, oproti dynamické kontrole typů prováděné virtuálním strojem při běhu. ve výsledku je zde méně příležitostí v programu k vyvolání chyby nepřípustného typu ClassCastException.

- **Méně přetypovávání**. při použití typových parametrů jsou dostupné specifičtější informace o typu, takže je zde daleko méně nutného přetypování oproti použití syrového typu, například při získávání objektu z kolekce.

- **Žádné hlášky typové kontroly.** Syrové typy vedou k varování typové kontroly „unchecked warning“ , čemuž lze předejít použitím typových argumentů.

- **Žádné odmítnuní v budoucnosti.** Specifikace jazyka Java prohlašuje, že syrové typy by mohly být v budoucích verzích Javy zamítnuty, a mohly by být stáhnuty jako vlastnost jazyka. Což znamená, že by použití parametrizovaných typů v případě zrušení možnosti používat syrové typy se stalo nutností.

12.2 Nevýhody parametrizovaných a generických typů:

- **žádný nárůst výkonu.** Obzvláště programátoři C++ mohou očekávat, že generické programy jsou výkonnější než negenerické programy, jelikož C++ šablony mohou zvýšit rychlost běhu programu. Ovšem pokud se podíváte pod pokličku Java kompilátoru a pochopíte, jak kompilátor překládá generický kód do bytového kódu, uvědomíte si, že Java kód při použití parametrizovaných typů neběží jakkoliv rychleji než negenerické programy.

•**typové parametry nelze přetypovat a přiřadit k nim hodnoty.** Jelikož typové parametry nemají předem známý typ, je nemožné jim v konstruktoru přiřadit předem hodnotu.

•**nutnost přesné parametrizace při implementaci rozhraní a generických tříd**.
Obzvláště pokud používáme složené generické parametry u rozhraní a metod, zjednodušenní může vést k nejrůznějším neočekávaným chybovým hlášením a varováním. Kompilátor nám nemusí dávat přesná chybová hlášení při nedostatečné parametrizaci kódu. Obzvláště, provádíme-li generifikaci negenerické verze kódu.
Například implementace metody compareTo:

```
 private class MyMapEntry<K extends Comparable<K>,V> implements Map.Entry<K,V>,
Comparable <Map.Entry<K,V> > {
.
.
.
  public int compareTo(Map.Entry<K,V> o2) {
    return key.compareTo(o2.getKey());
  }
```

pokud uděláme hlavičku metody bez parametrizovaného parametru metody, bude to muset asi takto:

```
public int compareTo(Map.Entry o2) {
     return key.compareTo((K) o2.getKey());
  }
```

Bez přetypování (K) o2.getKey dostaneme chybové hlášení "compareTo(K) in java.lang.Comparable(K) cannot be applied to (java.lang.Object)" a s přetypováním, obdržíme varování, kde ani s příkazem javac –Xlient:unchecked mainClass.java neobdržíme zprávu, že metoda int compareTo(Map.Entry<K,V>) není implementována, resp, že je nesprávně implementována jako int compareTo(Map.Entry) včetně svého obsahu.

12.3 Výhody použití syrových typů:

•**Nulová náročnost na učení.** Pokud ignorujete genericitu Java a použijete syrové typy v celém programu, nepotřebujete se seznámit s novými vlastnostmi jazyka, či se učit, jakékoliv složité chybové zprávy. Ovšem jak už bylo řečeno, stále existuje nebezpečí, že používání syrových typů bude z bezpečnostních důvodů zcela zakázáno, takže v budoucnu Vám to může poněkud zkomplikovat život ☺.

• **možnost použití i v nižších verzí Javy.** Ačkoliv program Retroweaver (http://retroweaver.sourceforge.net/) nabízí konverzi .class souborů do kódu použitelného i v nižších verzích než je Java 1.5, vynechání genericity a použití prostých syrových typů nabízí možnost kompilace zdrojového kódu v nižších verzích Javy i bez použití výše zmíněného programu.

12.4 Shrnutí:

Generické programování je velmi silný programový rys jazyka Java, který může být velmi dobrým pomocníkem nejen pro zajištění typové bezpečnosti programů, ale v některých případech dokáže upozornit i na něketré logické chyby v programu. Přesto jeho zvládnutí není tak triviální, jak by se mohlo zdát a skrývá nejedna úskalí.

Použitá literatura (References):

1. http://angelikalanger.com/genericsFAQ (1. 9. 2008-28. 4. 2009)

2. **Pecinovský Rudolf: Novinky jazyka Java 5.0 a upgrade aplikací (CP Brno 2005)**

3. **Generics Tutorial:** http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

4. http://www.java2s.com/Code/JavaAPI/java.util/implementsMap.htm (4. 2. 2009)