9. Polymorfismus a rozhraní

Tato kapitola navazuje na základní informace o objektech v kapitole 2, zde se budeme zabývat přetěžováním metod, polymorfismem a rozhraními.

9.1. Přetěžování metod a polymorfismus

Pojem polymorfismus pochází z řečtiny a znamená mnohotvarost. V objektovém programování vyjadřuje situaci, kdy se při stejném volání provádí různý kód. Která konkrétní metoda se provede, závisí na:

- předávaných parametrech,
- objektu, kterému je zpráva předána.

První varianta polymorfismu[15] je v Javě realizována přes **přetěžování metod**. Druhá varianta polymorfismu je v Javě závislá na dědičnosti a pozdní vazbě, realizuje se přes **překrývání metod**. Nyní si vysvětlíme přetěžování metod, překrývání metod bude popsáno v kapitole 11 věnované dědičnosti.

9.1.1. Přetěžování metod

Přetěžování metod označuje situaci, kdy nějaký objekt má více metod stejného jména. Při volání metody se konkrétní varianta metody vybere na základě typu a počtu předávaných parametrů.
V Javě může existovat ve třídě více metod stejného jména, musí se však lišit typem a počtem parametrů. Správná metoda se přiřadí při překladu – vybere se na základě počtu a typu parametrů. Příkladem přetížení metod lišící se počtem parametrů mohou být dvě metody *substring()* pro vybrání podřetězce ze třídy *String*. Jejich hlavičky následují:

```
String substring (int beginIndex)
String substring (int beginIndex, int endIndex)
```

Přetížení na základě typu parametru si ukážeme na statických metodách *valueOf()* opět ze třídy *String*, které převádějí hodnoty v primitivních typech na řetězce. Následují hlavičky některých variant:

```
static String valueOf(double d)
static String valueOf(int i)
static String valueOf(boolean b)
static String valueOf(char c)
```

V následující ukázce použití překladač přiřadí správnou metodu *valueOf()* na základě typu parametru:

```
     String text1 = "reálné číslo " + String.valueOf(2.52);
     String text2 = "celé číslo " + String.valueOf(178);
     String text3 = "logická hodnota " + String.valueOf(true);
     String text4 = "znak tabulátor " + String.valueOf('\t');
```

Teprve při použití překrytých metod se hovoří o polymorfismu – při použití metody *valueOf()* programátorem není hned vidět, že se pod ní schovávají čtyři různé metody. Tj. při stejném volání se provádí různý kód.
V situaci, kdy při vytváření třídy potřebujeme dvě významově podobné metody, je vhodné obě metody pojmenovat stejně a rozlišit je počtem či typem parametrů – vede to ke snadnějšímu použití třídy. Pokud metody nelze rozlišit počtem či typem parametrů, musí se napsat dvě metody s různým jménem. Příkladem takové situace ve třídě *String* jsou metody *replaceAll()* a *replaceFirst()*, které nahrazují části řetězce jiným. První metoda nahradí všechny výskyty, které odpovídají vzoru, druhá metoda nahradí pouze první výskyt.

---

[15] Někteří teoretici objektového programování do polymorfismu zahrnují pouze překrývání.

Pokud se metody liší typem parametru a mezi typy je vztah dědičnosti, tak se při překladu vybírá metoda s konkrétnějším typem.
V Javě je možné též přetěžovat konstruktory, tj. jedna třída může mít více konstruktorů – ukázku můžete vidět ve třídě `Ucet`, která je popsána v kapitole 2.6.

9.1.2. Příklad polymorfismu s přetěžováním metod

Polymorfismus s přetížením metody si ukážeme na příkladu louky s květinami, nad kterou létají včely (instance třídy `Vcela`) a motýli (instance třídy `Motyl`). Naším úkolem je ve třídě `Louka` dopsat metody pro vkládání motýlů a včel na louku[16]. Pro včely i pro motýly si vytvoříme samostatné seznamy (seznamy jsou popsány v kapitole 10.3):

```
private List<Vcela> vcely;
private List<Motyl> motyli;
```

Nyní si ukážeme tři různé varianty řešení tohoto úkolu. Čtvrtá varianta bude uvedena v části věnované rozhraním.

Varianta se dvěmi metodami různých názvů

```
public void pridejMotyla (Motyl motyl) {
     motyli.add(motyl);
}

public void pridejVcelu (Vcela vcela) {
     vcely.add(vcela);
}
```

Zmiňovaná varianta má tyto nevýhody:

- duplikuje se informace o tom, zda se předává motýl či včela – je to napsáno nejen v názvu metody, ale i jako typ parametru metody,
- vytváří se tím velké množství metod s různými názvy, které komplikují použití. Ve třídě `Louka` bude počet metod stejný, ale ten, kdo bude chtít používat tuto třídu, si bude muset pamatovat větší počet metod (místo jedné metody *`pridej()`* dvě metody *`pridejVcelu()`* a *`pridejMotyla()`*),

Varianta s jednou metodou s rozskokem dle typu parametru

```
public void pridej (Object o) {
     if (o instanceof Vcela) {
          Vcela vcela = (Vcela)o;
          vcely.add(vcela);
     }
     else {
          if (o instanceof Motyl) {
               Motyl motyl = (Motyl)o;
               motyli.add(motyl);
          }
          else {
               throw new InvallidArgumentException(
                                   "lze vkládat pouze motýly a včely");
          }
     }
}
```

[16] Příznivcům bojových her doporučujeme si představovat, že programují zbrojnici, do které se vkládají meče, luky a další zbraně.

Tato varianta má následující nevýhody:

- Metoda je poměrně složitá, je to více řádků kódu, než varianta s přetížením metod, struktura metody je složitější (přibývá rozhodování ohledně typu).
- Kontrola přípustných typů se přesouvá z doby překladu do běhu aplikace. Překladač povolí, aby parametrem byla instance libovolné třídy (např. `String`), při běhu však v tomto případě vznikne výjimka.
- Při použití (tj. při vkládání objektů na louku) by se měl ošetřovat vznik výjimky, tj. vzniká složitější kód i při použití této metody.

Varianta s přetížením metody

Varianta přidávání včel a motýlů na louku s přetížením metody bude vypadat následovně:

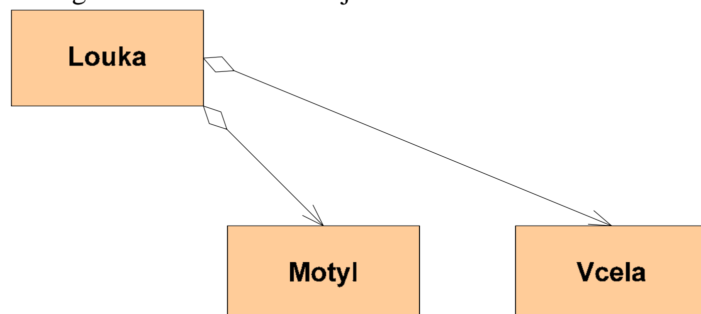
```
public void pridej (Vcela vcela) {
     vcely.add(vcela);
}

public void pridej (Motyl motyl) {
     motyli.add(motyl);
}
```

Všimněte si, že jsou zde dvě metody stejného jména s různým typem parametrů. Volba správné metody se provádí při překladu na základě typu parametru. Kód pro vkládání včel a motýlů by mohl vypadat následovně (nejsou uvedeny parametry konstruktoru instancí tříd `Vcela` a `Motyl`):

```
louka.pridej(new Vcela( ...... ));
louka.pridej(new Vcela( ...... ));
louka.pridej(new Motyl( ...... ));
louka.pridej(new Motyl( ...... ));
```

Pokud si porovnáte tuto variantu s předchozími, zjistíte, že přetěžování metod zjednodušuje psaní kódu na straně používání metod i na straně třídy, která metodu poskytuje (přijímá zprávy).
V diagramu na obrázku 9.1 je zobrazen vztah mezi třídou `Louka` a třídami `Motyl` a `Vcela`.



Obrázek 9.1 Diagram tříd zobrazující motýly a včely na louce

Přetěžování metod též umožňuje omezit počet selekcí (příkazů `if` a `switch`) v programu – je to zřetelné při porovnání s předchozí variantou.

9.2. Rozhraní

Pojem rozhraní se v programování používá v různých významech, nejčastější jsou následující:

- **uživatelské rozhraní** (user interface), které definuje způsob komunikace mezi uživatelem a programem. Tato problematika je velmi důležitá, ale mimo zaměření těchto skript.
- **aplikační programové rozhraní** (application programming interface, API), což je množina definicí, která určuje, jak program (část programu) může komunikovat s jiným. Je to určitá abstraktní vrstva, která je k dispozici programátorům při psaní zdrojového kódu dalších programů. Příkladem API může být rozhraní pro programátory ve Windows (Microsoft Win32

API, viz příslušné části http://msdn.microsoft.com/) či rozhraní pro využívání služeb Google (Google API, viz http://www.google.com/apis/). Java má také svá API – programátor aplikace může využívat jak standardní knihovny (např. pro práci s datovými strukturami, se soubory, pro vytváření grafického rozhraní), tak velké množství knihoven a tříd, která dávají za určitých podmínek k dispozici různí programátoři a různé firmy. V Javě jsou tato aplikační rozhraní obvykle popsána v dokumentaci vygenerované programem **javadoc**.

- **rozhraní** (interface) jako jazyková konstrukce, která podporuje vytváření abstraktní vrstvy mezi konkrétními implementacemi a programy, které je používají. To je náplň této části kapitoly.
- „**vzdálené rozhraní**" (remote interface) – rozhraní z programovacího jazyka na jiné systémy. V tomto smyslu se zavedla rozhraní např. do Pascalu jako rozhraní pro přístup ke komponentní technologii CORBA. V Javě se toto označení příliš nepoužívá, byť zde existují rozhraní pro vzdálené volání procedur (RMI) či rozhraní do komponentní technologie CORBA.

9.2.1. Deklarace rozhraní

Deklarace rozhraní je podobná deklaraci rozhraní třídy s tím, že v rozhraní se definují pouze hlavičky veřejných metod a případně veřejné konstanty. Deklarace rozhraní se obvykle zapisuje do samostatného souboru s koncovkou `.java`, který se jmenuje stejně jako uvnitř uvedené rozhraní (stejné pravidlo jako pro třídy či pro výčtový typ).
Obecná **deklarace rozhraní** vypadá takto:

```
[public] interface Identifikátor [extends rozhraní1 [, rozhraní2 ...]] {
}
```

Dovnitř rozhraní se zapisují pouze veřejné metody – může a nemusí být u nich uveden modifikátor `public`. Pokud není uveden, doplní tento modifikátor překladač, tj. nelze uvést jiné modifikátory přístupu. Následuje příklad deklarace rozhraní *ObyvatelLouky*, které by mělo být zapsáno v souboru *ObyvatelLouky.java*. Rozhraní obsahuje pouze jednu metodu.

```
public interface ObyvatelLouky {
     public void jednaAkce();
}
```

Třída, která **implementuje rozhraní**, musí tuto skutečnost vyjádřit v hlavičce třídy pomocí klíčového slova **implements**. Za tímto klíčovým slovem se uvádějí jednotlivá rozhraní, která třída implementuje (může jich implementovat více). Třída (pokud není abstraktní, viz kapitola 11.4) musí deklarovat všechny metody předepsané v rozhraní – musí odpovídat modifikátory (tj. musí být `public`), musí odpovídat jméno a hlavička metody.

> **Pokud implementujete nějaké rozhraní a neuvedete některou požadovanou metodu (např. metodu *xxx()*), tak překladač uvede méně srozumitelné chybové hlášení. Nevypíše, že chybí nějaká metoda požadovaná rozhraním, ale oznámí, že máte svoji třídu označit jako abstraktní, neboť není implementována metoda *xxx()*.**

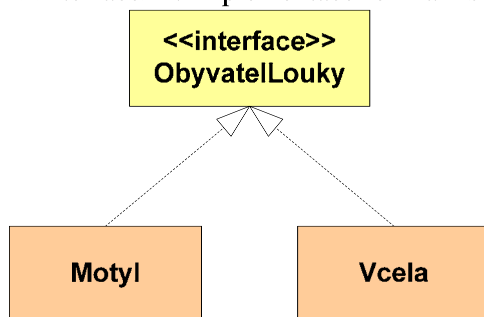
Třída *Motyl* může schématicky vypadat následovně:

```
public class Motyl implements ObyvatelLouky {
     public void jednaAkce () {
          if (naKvetineSNektarem()) {
               // sbirej nektar
          }
          else {
               preletni();
          }
     }
}
```

Třída `Vcela` by vypadala schématicky takto:

```
public class Vcela implements ObyvatelLouky {
     public void jednaAkce () {
          if (vUlu() && maNektar()) {
               // odevzdat nektar
          }
          else {
               if (plnyKosicek()) {
                    // let k úlu
               }
               else {
                    if (naKvetineSNektarem()) {
                         // sbirej nektar
                    }
                    else {
                         preletni();
                    }
               }
          }
     }
}
```

V diagramu tříd se obvykle rozhraní zobrazuje jako třída s tím, že se před jméno třídy uvede **stereotyp** <<interface>>. Implementace rozhraní se zobrazuje pomocí přerušované čáry s plnou šipkou na konci.



Obrázek 9.2 Zobrazení rozhraní a implementace rozhraní v diagramu tříd

V rozhraní je možné vedle metod definovat i pojmenované konstanty. Tato možnost má pouze jednu problematickou výhodu – pokud nějaká třída implementuje rozhraní s konstantami, nemusí se v ní uvádět názvy konstant společně s názvem třídy, ve které jsou konstanty definovány. Vhodnější je definovat konstanty v samostatné třídě či ještě lépe za pomocí výčtového typu.

9.2.2. Rozhraní umožňuje volbu implementace

Tuto možnost využití rozhraní si ukážeme na seznamech pro ukládání prvků z balíčku `java.util` (blíže jsou popsány v kapitole 10.1). Pro seznamy jsou zde dvě základní implementace *`ArrayList`* a *`LinkedList`*. Třída *`ArrayList`* je rychlejší pro přístup k prvkům, třída *`LinkedList`* je rychlejší, pokud je časté vkládání prvků dovnitř seznamu či přesun/rušení prvků uvnitř seznamu. Obě dvě třídy implementují rozhraní `List`.
Při deklaraci seznamu v programu je vhodné jako typ použít příslušné rozhraní:

```
     private List <Motyl> motyli;
```

a v konstruktoru potom inicializovat seznam příslušným typem:

```
     motyli = new ArrayList<Motyl>();
```

Použití rozhraní v typu datového atributu omezuje programátora na používání pouze těch metod, které jsou deklarovány v rozhraní. Výhodou však je, že změnou pouze na jednom řádku můžeme změnit

implementaci. Pokud bychom chtěli použít `LinkedList`, tak stačí upravit řádek při vytváření instance seznamu:

```
    motyli = new LinkedList<Motyl>();
```

Tento výběr implementace samozřejmě nemusí být pouze statický v době překladu, na základě podmínky lze vybrat různé implementace za běhu aplikace:

```
    if (podmínka) {
        motyli = new ArrayList<Motyl>();
    }
    else {
        motyli = new LinkedList<Motyl>();
```

9.2.3. Rozhraní a polymorfismus

Pokud rozhraní implementuje více tříd, je možné ho použít jako zástupce těchto tříd. Ukážeme si to opět na příkladu s motýly a včelami. Třídy `Motyl` a `Vcela` nyní implementují rozhraní `Obyvatel` a je tudíž možné ve třídě popisující louku vytvořit společný seznam pro instance obou tříd:

```
    private List<Obyvatel> obyvateleLouky;
```

Vytvoření instance bude vypadat takto:

```
    obyvateleLouky = new ArrayList<Obyvatel>();
```

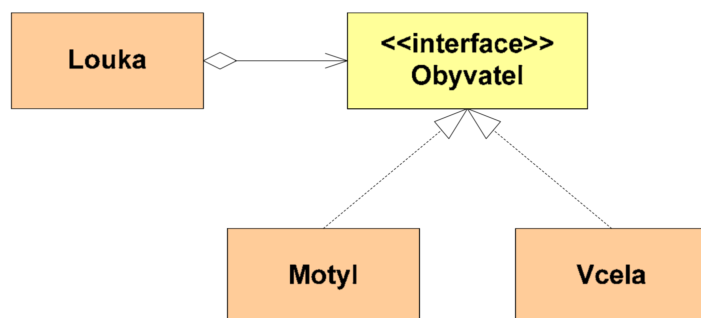
Nyní v této třídě stačí jedna metoda pro vkládání motýlů, včel:

```
    public void pridej (Obyvatel obyvatel) {
        obyvateleLouky.add(obyvatel);
    }
```

Při porovnání této metody s variantami uvedenými na začátku kapitoly je zřejmé, že toto je nejkratší a typově bezpečné řešení. Je zde ještě jedna výhoda – pokud se přidá další obyvatel louky, není potřeba upravovat zdrojový kód třídy popisující louku.
Použití této varianty má také jednu nevýhodu – ve třídě popisující louku můžeme používat pro obyvatele louky pouze ty metody, které jsou definovány v rozhraní `Obyvatel`, tj. v našem případě volat metodu `jednaAkce()`. Následuje ukázka metody, která simulaci louky posune o jeden krok:

```
    public void jedenKrok() {
        for (Obyvatel obyvatel : obyvateleLouky) {
            obyvatel.jednaAkce();
        }
        pocetKroku++;
    }
```

Tato metoda je ukázkou **polymorfismu** – při volání metody `jednaAkce()` se volá různý kód v závislosti na tom, zda konkrétní obyvatel je instance třídy `Vcela` či instance třídy `Motyl`. Využívá se zde toho, že třídy implementují stejné rozhraní, které jim předepisuje existenci metod. Pokud v našem příkladu třída implementuje rozhraní `Obyvatel`, musí mít metodu `jednaAkce()`. Pokud nějaká třída toto rozhraní neimplementuje, tak překladač zabrání umístění instance této třídy do seznamu obyvatel louky (do datového atributu `obyvateleLouky`).
Následuje diagram tříd zobrazující vztahy mezi třídami `Louka`, `Motyl`, `Vcela` a rozhraním `Obyvatel`. Porovnejte ho s diagramem na obrázku 9.1, popisujícím obdobnou situaci bez polymorfismu. Na tomto diagramu není vztah mezi třídou `Louka` a konkrétními třídami `Motyl` a `Vcela`, ale kreslí se asociace od `Louky` k rozhraní `Obyvatel`.



Obrázek 9.3 Diagram tříd zobrazující motýly a včely na louce s využitím polymorfismu

Stejný typ polymorfismu zajišťuje i dědičná hierarchie – to si vysvětlíme v kapitole 11.5.

9.2.4. Rozhraní umožňuje předávat metody jako parametr

Existence rozhraní též umožňuje předávat metodu (funkci) jako parametr či ji přiřazovat do datového atributu. Příkladem může být třídění – existuje obecný algoritmus pro třídění seznamu prvků, ale ten potřebuje vědět, jak se porovnají dva prvky (např. zda se mají dvě instance třídy `Ucet` porovnat dle čísla účtu či dle stavu na účtu). Tj. algoritmus pro třídění potřebuje metodu pro porovnání dvou instancí třídy `Ucet`. V Javě je to řešeno pomocí rozhraní – existuje rozhraní `Comparator`, které definuje metodu *`int compare(E prvni, E druhy)`*. Programátor vytvoří pomocnou třídu, která implementuje rozhraní `Comparable`, ve které napíše metodu *`compare()`* na porovnání dvou účtů. Při třídění se předají statické metodě na třídění (`Collections.sort()`) dva parametry: vlastní seznam a instance pomocné třídy. Tím algoritmus pro třídění získá metodu, pomocí které bude porovnávat vždy dvě instance třídy `Ucet`.
Podrobnější popis třídění a včetně příkladů je v kapitole 10.3.

9.2.5. Rozhraní a vícenásobná dědičnost

Někteří autoři přistupují k rozhraní jako ke specifické verzi vícenásobné dědičnosti, neboť na rozhraní se lze dívat jako specifickou variantu abstraktní třídy – na abstraktní třídu, která definuje pouze veřejné abstraktní metody. Např. jazyk C++ nepodporuje rozhraní, ale podporuje vícenásobnou dědičnost. Ve prospěch rozhraní zaznívají tyto argumenty:

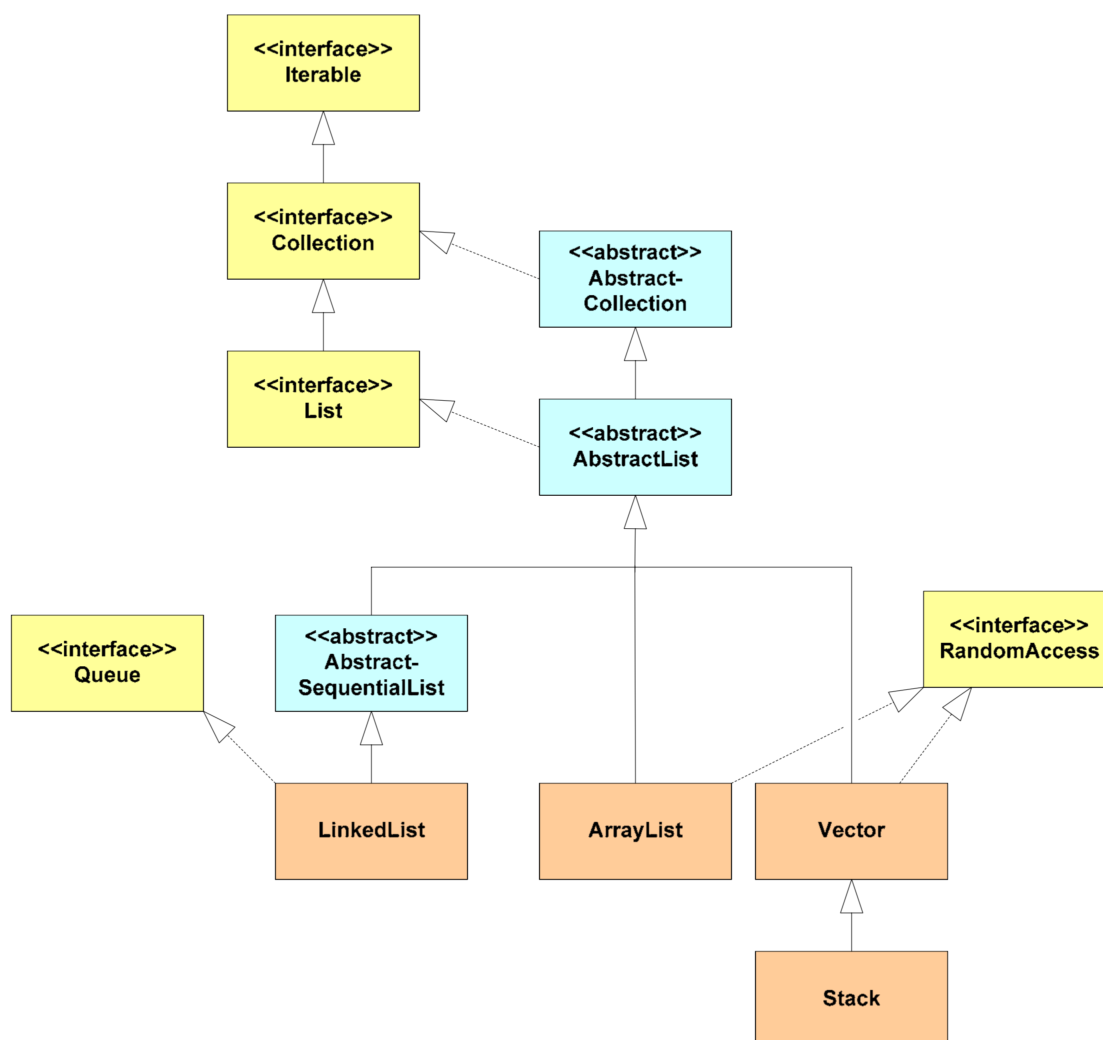
- S využitím rozhraní je návrh jednodušší, neboť je menší riziko vzniku kolizí jmen, kdy dva různí předci (dvě různá implementovaná rozhraní) definují metody stejného jména s různým významem. Je to dáno tím, že v abstraktních třídách může být více prvků, které mohou kolidovat – např. datové atributy či metody podporující implementaci.
- Jazykový element rozhraní více podporuje správný návrh API – zapouzdření a znovu-použitelnost – neboť programátor má menší možnost se v této fázi zabývat implementací. Použití abstraktních tříd programátora více svádí zabývat se implementací, neboť může do nich zapisovat datové atributy či privátní metody, které patří do implementace API.

Preference rozhraní před abstraktními třídami v Javě se prosazovala postupně – je to vidět při porovnání nejstarších částí, které preferují abstraktní třídy (např. třídy pro vstup/výstup v balíčku `java.io`) s novějšími (např. dynamické datové struktury v balíčku `java.util`).

9.2.6. Dědičnost mezi rozhraními

Existuje i dědičnost mezi rozhraními – jedno rozhraní může dědit definice od jiného, popř. jiných rozhraní. V hlavičce rozhraní se dědičnost rozhraní vyznačí klíčovým slovem **extends,** za kterým se zapisuje jedno či více rozhraní. Ukážeme si to na příkladu třídy `AbstractList`, jejíž předci, potomci a implementovaná rozhraní jsou zobrazeny na obrázku 9.4. Můžete zde vidět dědičnost mezi rozhraními – rozhraní `List` je potomkem rozhraní `Collection` (které je potomkem rozhraní `Iterable`). Hlavička rozhraní `List` vypadá následovně (nejsou uvedeny generické typy):

```
public interface List extends Collection {
```



Obrázek 9.4 Vztah mezi třídami a rozhraními v části balíčku java.util – AbstractList a potomci

Na obrázku není z důvodů přehlednosti vyznačeno, že třídy implementují též rozhraní `Cloneable` a `Serializable`.