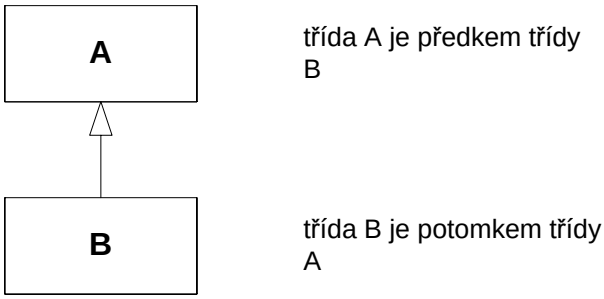
11.Dědičnost

V této kapitole si vysvětlíme jeden ze základních pojmů objektově orientovaného programování – dědičnost (inheritance). S ní souvisejí i následující témata:

- předek a potomek třídy,
- klíčová slova `extends` a `super`,
- přetypování referenčních typů,
- abstraktní třídy,
- abstraktní metody,
- překrývání metod,
- pozdní vazba.

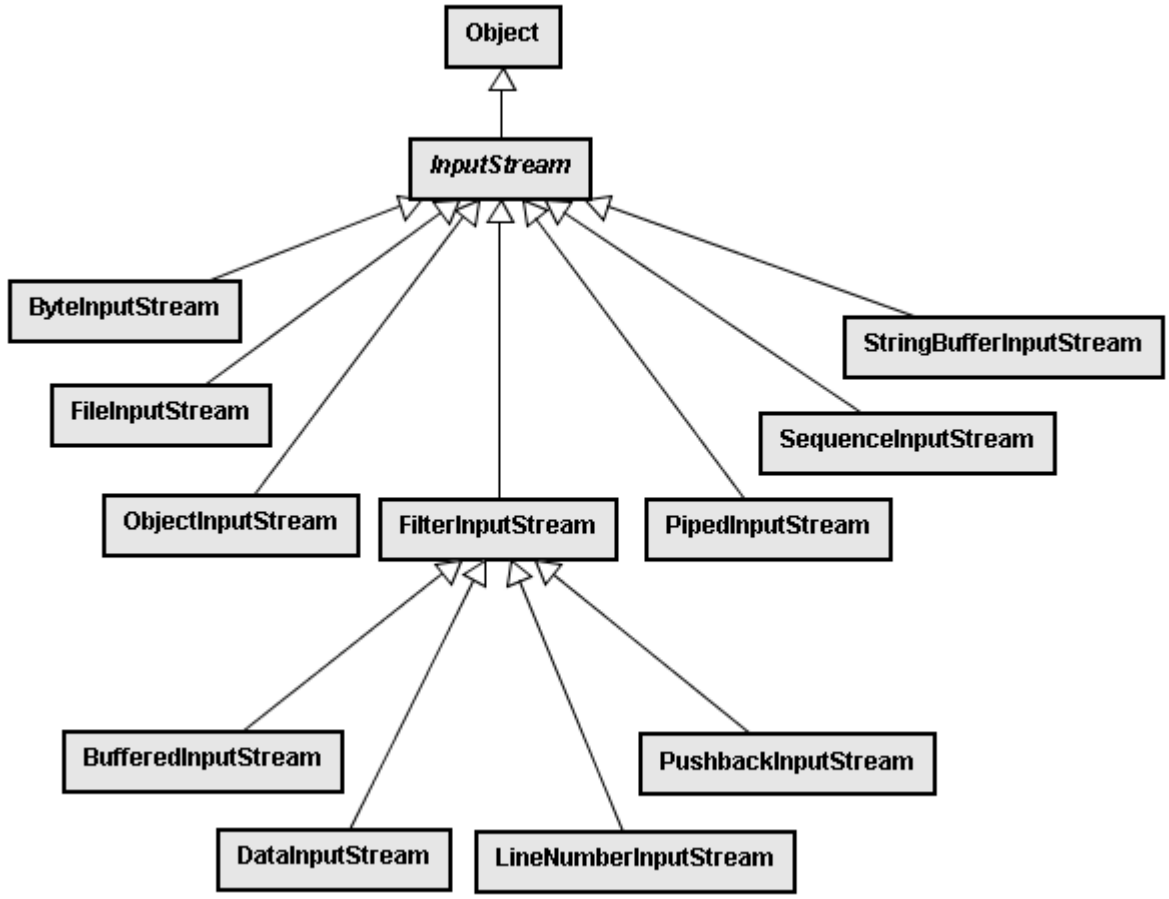
Tato kapitola navazuje na základní informace o objektech v kapitole 2, zde se však nebudeme zabývat jednou třídou, ale vztahy mezi třídami a objekty.

Dědičnost je jednou z forem **znovupoužitelnosti** – vytvářená třída (potomek) do sebe absorbuje datové atributy a dědí metody z jiné třídy (předek) a dále je rozšiřuje a upravuje. V diagramu tříd se dědičnost vyznačuje pomocí šipky, u které trojúhelník na konci směřuje k předkovi.



Obrázek 11.1 Dědičnost, předek a potomek

Dědičnost není pouze jednoúrovňová – potomek nějaké třídy může mít dále své potomky. Tito potomci dědí metody od všech tříd na vyšší úrovni. Takto vzniká hierarchie tříd, ve které není omezen počet úrovní. Na obrázku 11.2. je část dědičné hierarchie balíčku `java.io` (viz kapitola 13) – je zahrnuta pouze abstraktní třída `InputStream` a její potomci. Zobrazena je i dědičnost ze třídy `Object`, která se u aplikací v Javě obvykle nekreslí.
Java podporuje jednonásobnou dědičnost – každá třída může a musí mít právě jednoho předka. Stromová hierarchie tříd začíná třídou `Object` (tato jediná třída nemá předka), všechny třídy jsou přímým či nepřímým potomkem této třídy.
Většina programovacích jazyků podporuje jednonásobnou dědičnost, některé (např. C++ či Eifel) podporují vícenásobnou dědičnost – třída může mít více předků současně.



Obrázek 11.2 Dědičná hierarchie části balíčku java.io – InputStream a potomci

Příklad dědičnosti

Pro vysvětlení dědičnosti použijeme příklad s účty z kapitoly 2. Třída `Ucet` vypadala takto:

```
public class Ucet {

     private int cisloUctu;
     private String vlastnik;
     private double stav = 0;

     public Ucet (int cisloUctu, String vlastnik){
          this(cisloUctu, vlastnik, 0); // volání druhého konstruktoru
     }

     public Ucet (int cisloUctu, String vlastnik,
               double pocatecniVklad){
          this.cisloUctu = cisloUctu;
          this.vlastnik = vlastnik;
          stav = pocatecniVklad;
     }

     public double getStav(){
          return stav;
     }
```

```
     public void vloz (double castka){
          stav = stav + castka;
     }

     public boolean vyber (double castka){
          if ((stav – castka) >= 0) {
               stav = stav – castka;
               return true;
          }
          else {
               return false;
          }
     }
}
```

Nyní vytvoříme další typ účtu – účet s kontokorentem (žirový účet), u kterého lze vybírat do předem stanoveného zůstatku. Vzhledem k tomu, že základní datové atributy a metody jsou podobné, použijeme dědičnosti – v hlavičce třídy uvedeme klíčové slovo **extends** a za ním jméno třídy předka[24]:

```
public class ZiroUcet extends Ucet {
```

Žirový účet bude mít navíc jeden datový atribut – limit kontokorentu. Metody `getStav()` a `vloz()` se nemění – tj. je možné je zdědit, nemusí se tudíž psát do kódu potomka. Metoda `vyber()` bude odlišná – musí se povolit výběr až do limitu kontokorentu. Uvedení metody se stejnou hlavičkou v potomkovi se nazývá **překrytí metody** (overriding). Pokud má metoda v potomkovi jinou hlavičku (liší se počtem či typy parametrů), nejedná se o překrytí metody, ale o **přetížení metody** – viz kapitola 9.
V příkladu je komplikace i s datovým atributem `stav` – tento datový atribut je `private` ve třídě `Ucet` a tudíž není dostupný ani pro potomky. Řešením je vytvořit ve třídě `Ucet` metodu `setStav()` s modifikátorem přístupu `protected`:

```
     protected void setStav(double castka) {
          stav = castka;
     }
```

Metoda `vyber()` by poté ve třídě `ZiroUcet` vypadala takto (doplněna je též metoda pro zjištění limitu, naopak chybí konstruktory, které si popíšeme později):

```
public class ZiroUcet extends Ucet {
     private double limit = 0;
     // konstruktor(y)
     public boolean vyber (double castka) {
          if ((getStav()+limit – castka) >= 0) {
               setStav(getStav()– castka);
               return true;
          }
          else {
               return false;
          }
     }
     public double getLimit() {
          return limit;
     }
}
```

[24] Pokud je předkem třídy třída `Object`, nemusí se `extends` uvádět – tuto dědičnost automaticky doplní překladač.

Pokud bychom nemohli zasahovat do kódu předka, tj. pokud bychom do předka nemohli doplnit metodu `setStav()`, museli bychom si vybrat některou z následujících variant:

- nastavovat stav pomocí metody `vyber()` s tím, že se použijí záporné hodnoty,
- v potomkovi si udržovat samostatný datový atribut, který by obsahoval částku vybranou z kontokorentu. V této variantě je potřeba upravit i další metody.

11.1. Dědičnost a konstruktory, klíčové slovo super

Ještě je potřeba do třídy `ZiroUcet` doplnit konstruktory. Vytvoříme tři, první umožní nastavit všechny čtyři datové atributy (číslo účtu, vlastníka, počáteční vklad a limit). Druhý a třetí konstruktor odpovídají přetíženým konstruktorům z předka – cílem je, aby i z hlediska vytváření se mohl potomek používat stejně jako třída předka. I tyto dva konstruktory je nutné napsat, neboť v Javě se **konstruktory nedědí**. Z konstruktoru potřebujeme volat konstruktor předka – pro volání konstruktoru předka se používá klíčové slovo **super.** Super se používá podobně jako `this` při volání jiného konstruktoru ve stejné třídě, opět musí být prvním příkazem v konstruktoru.

```
public class ZiroUcet extends Ucet {
     private double limit = 0;

     public ZiroUcet (int cisloUctu, String vlastnik,
                   double pocatecniVklad, double limit){
          super(cisloUctu, vlastnik, pocatecniVklad);
          this.limit = limit;
     }

     public ZiroUcet (int cisloUctu, String vlastnik,
                   double pocatecniVklad){
          this(cisloUctu, vlastnik, pocatecniVklad, 0);
     }

     public ZiroUcet (int cisloUctu, String vlastnik){
          this(cisloUctu, vlastnik, 0, 0); //volání prvního konstruktoru
     }
```

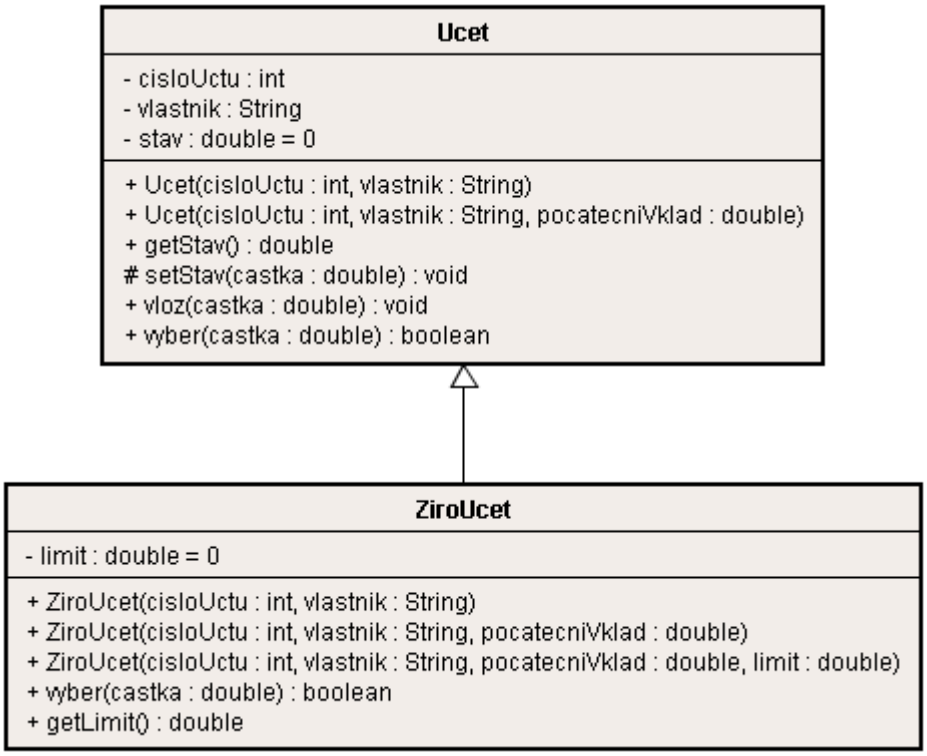
V prvním konstruktoru se nejdříve pomocí super volá konstruktor předka – předávají se mu tři parametry. Na dalším řádku se nastavuje datový atribut limit. Druhý a třetí konstruktor je napsán s využitím prvního konstruktoru – doplní se defaultní hodnoty na místě chybějících parametrů[25]. Vztah mezi třídami `Ucet` a `ZiroUcet` je na obrázku 11.3.

Pokud v konstruktoru potomka neuvedeme volání konstruktoru předka (`super`) či volání jiného konstruktoru stejné třídy (`this`), překladač automaticky na první řádek konstruktoru doplní volání:

```
super ();
```

Tj. volá se konstruktor předka bez parametrů. Pokud předek takový konstruktor nemá, vznikne chyba při překladu. Třída `Object` (předek všech tříd) má konstruktor bez parametrů.

[25] Některé objektové programovací jazyky podporují psaní defaultních hodnot již přímo do deklarace hlavičky metody. Tím se lze někdy vyhnout psaní přetěžovaných metod a konstruktorů. Příkladem takových jazyků je Python či Ruby.



Obrázek 11.3 Vztah tříd Ucet a ZiroUcet (diagram tříd)

Klíčové slovo **super** má vedle volání konstruktoru předka ještě dvě použití:

- lze se pomocí něho odkazovat na datový atribut předka, pokud není `private` (vhodnější je mít všechny datové atributy `private` a přistupovat k nim pomocí metod), např.

```
super.stav
```

- lze se pomocí super odkázat na metodu předka. Používá se v situaci, kdy v předkovi a v potomkovi dochází k překrytí metod a potřebujeme zavolat z potomka metodu předka. V metodě `vyber()` ve třídě `ZiroUcet` by se mohlo nejdříve zkusit vybrat pomocí metody `vyber()` z předka, která má stejnou hlavičku:

```
public boolean vyber (double castka) {
     if (super.vyber(castka)) {
          // pokračování metody
     }
}
```

11.2. Vytvoření instance při dědičnosti

Je potřeba odlišovat dědičnost v době psaní kódu (při překladu) a při vlastním vytváření instancí. Při psaní kódu máme z potomka přístup k datovým atributům a metodám předka, pokud nejsou `private`. Pomocí `super` lze přistupovat i k datovým atributům předka, které mají stejné jméno jako datové atributy v potomkovi. Obdobně lze přistupovat i k metodám předka se stejnou hlavičkou. Odlišná situace je u vytvořené instance. Každá instance má:

- Přiřazen paměťový prostor pro všechny datové atributy definované ve třídě i ve všech předcích (včetně privátních datových atributů v předcích, ke kterým není přímý přístup).
- Přiřazeny všechny své konstruktory.
- Konstruktory předků nejsou přímo dostupné, jsou ve stavu, kdy je lze volat pouze z potomka.

- Přiřazeny všechny své metody a metody předků, které nejsou v potomkovi překryty (v rámci dědičné hierarchie). Pokud je v předkovi a v potomkovi metoda se stejnou hlavičkou, tak je přiřazena pouze ta z potomka.

Každá instance třídy `ZiroUcet` z našeho příkladu s účty bude mít:

- datové atributy `cisloUctu`, `vlastnik`, `stav` a `limit`,
- tři konstruktory ze třídy `ZiroUcet`,
- metodu `vyber(double castka)` ze třídy `ZiroUcet`,
- tři metody ze třídy `Ucet` – `getStav()`, `setStav()` a `vloz()`,
- metody ze třídy `Object` – `equals()`, `hashCode()`, `finalize()`, `toString()`, `clone()`, `getClass()`, `notify()`, `notifyAll()` a tři varianty metody `wait()`,
- možná nějaké privátní datové třídy ze třídy `Object` – třída `Object` nemá žádné přístupné datové atributy, může mít však privátní.

11.3. Modifikátor přístupu protected

Modifikátor přístupu **protected** se používá v situaci, kdy chceme nějakou metodu zpřístupnit pouze pro potomky (tj. není veřejně dostupná). Používá se v situaci, kdy není vhodné, aby se metoda veřejně používala, ale současně předpokládáme, že tuto metodu využije více potomků.
Modifikátor `protected` lze používat i u datových atributů – v tomto případě je však nutno zvážit, zda není vhodnější deklarovat datové atributy `private` a z potomků k nim přistupovat pomocí vhodných metod (get/set metody), které budou chráněné (`protected`).

11.4. Abstraktní třídy

Abstraktní třída se používá jako předek v dědičné hierarchii a představuje třídu, od které nemá smysl vytvářet instance. Abstraktní třída může mít vedle konkrétních datových atributů a metod i abstraktní metody (datové atributy nemohou být abstraktní). Abstraktní metoda obsahuje pouze hlavičku, potomek této abstraktní třídy musí tuto metodu implementovat (nebo musí být potomek též abstraktní).
Abstraktní třídy si vysvětlíme na příkladu se včelami a motýly, které létají po louce. Třídy `Motyl` a `Vcela` by mohly vypadat následovně (většina kódu je pouze naznačena):

```
public class Vcela {
     private Pozice pozice;
     public void jedenPohyb () {
          if (vUlu() && maNektar()) {
               // odevzdat nektar
          }
          else {
               if (plnyKosicek()) {
                    // let k úlu
               }
               else {
                    if (naKvetineSNektarem()) {
                         // sbirej nektar
                    }
                    else {
                         preletni();
                    }
               }
          }
     }
```

```
     private void preletni() {
          // vyber náhodně květinu v nejbližším okolí
          // přesuň se na vybranou květinu
     }
     private boolean maNektar() {
          // obsah metody
     }
     private boolean vUlu() {
          // obsah metody
     }
     private boolean plnyKosicek() {
          // obsah metody
     }
     private boolean naKvetineSNektarem() {
          if (pozice.jeKvetina()) {
               Kvetina kvetina = pozice.getKvetina();
               return kvetina.maNektar();
          }
          else {
               return false;
          }
     }
}
```

Třída *`Motyl`* vypadá následovně:

```
public class Motyl {
     private Pozice pozice;
     public void jedenPohyb () {
          if (naKvetineSNektarem()) {
               // sbirej nektar
          }
          else {
               preletni();
          }
     }
     private void preletni() {
          // vyber náhodně květinu v nejbližším okolí
          // přesuň se na vybranou květinu
     }
     private boolean naKvetineSNektarem() {
          if (pozice.jeKvetina()) {
               Kvetina kvetina = pozice.getKvetina();
               return kvetina.maNektar();
          }
          else {
               return false;
          }
     }
}
```

Když porovnáte obě třídy, zjistíte, že mají stejné dvě metody – *`preletni()`* a *`naKvetineSNektarem()`*. Proto by bylo vhodné vytvořit předka, který by obsahoval tyto metody. Nemá však smysl vytvářet instance tohoto předka, proto předek bude abstraktní.
V hlavičce abstraktní třídy musí být uvedeno slovo **abstract**, v abstraktní třídě mohou být abstraktní metody. Abstraktní metoda je taková, u které se při návrhu požaduje, aby ji měl implementovanou každý potomek abstraktní třídy. V abstraktní třídě se uvede pouze hlavička metody s modifikátorem *`abstract`*.

Abstraktní třída může obsahovat datové atributy i konstruktory. I když nelze vytvořit instanci abstraktní třídy, musí konstruktor abstraktní třídy existovat – jak jsme si již řekli dříve, tak v konstruktoru potomka se nejdříve volá konstruktor předka a teprve poté se provádí další příkazy v konstruktoru potomka. Pokud nenaimplementujeme alespoň jeden konstruktor, bude stejně jako u „obyčejné“ třídy překladačem doplněn implicitní konstruktor.
Překladač Javy kontroluje, zda kdekoli v kódu není vytvářena instance abstraktní třídy.
Třída `LetajiciHmyz` bude obsahovat abstraktní metodu `jedenPohyb()`, konkrétní metody `preletni()` a `naKvetineSNektarem()`. Součásti třídy je i datový atribut `pozice`, který je potřeba v obou konkrétních metodách. Metody `preletni()` a `naKvetineSNektarem()` mají modifikátor přístupu `protected`, aby se mohly používat v potomcích.

```
public abstract class LetajiciHmyz {
     private Pozice pozice;
     public abstract void jedenPohyb ();
     protected void preletni() {
          // vyber náhodně květinu v nejbližším okolí
          // přesuň se na vybranou květinu
     }
     protected boolean naKvetineSNektarem() {
          if (pozice.jeKvetina()) {
               Kvetina kvetina = pozice.getKvetina();
               return kvetina.maNektar();
          }
          else {
               return false;
          }
     }
}
```

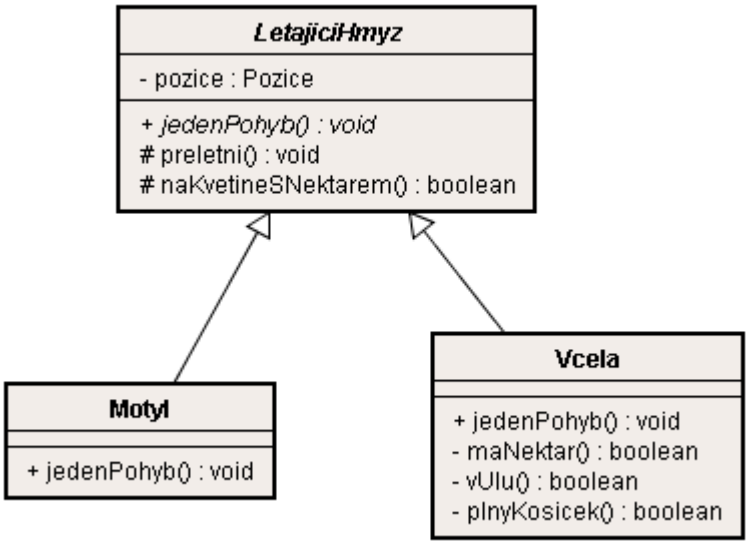
Třída `Motyl` potom bude vypadat takto:

```
public class Motyl extends LetajiciHmyz {
     public void jedenPohyb () {
          if (naKvetineSNektarem()) {
               // sbirej nektar
          }
          else {
               preletni();
          }
     }
}
```

Úpravy ve třídě `Vcela` budou velmi podobné.
Nelze vytvořit přímo instanci abstraktní třídy, tj. není možné přímo zavolat konstruktor `new LetajiciHmyz()`. Lze však vytvořit instanci některého z potomků a přetypovat ho na typ `LetajiciHmyz`, např. takto:

```
LetajiciHmyz hmyz = new Motyl();
```

V diagramu tříd se jméno abstraktní třídy píše kurzívou, obdobně i jméno abstraktní metody se píše kurzívou. Druhou alternativou je vyznačit abstraktní třídy a metody pomocí stereotypu.



Obrázek 11.4 Vztah abstraktní třídy LetajiciHmyz a tříd Motyl a Vcela

11.5. Polymorfismus a překrytí metod

Polymorfismus ve spojitosti s dědičností souvisí s tématy přetypování referenčních typů, překrývání metod a pozdní vazba. Většinou jsme se již s nimi seznámili, zde si je ještě zopakujeme.

11.5.1. Přetypování referenčních typů

Jak jsme si již uvedli v kapitole 3, i u referenčních typů lze používat přetypování. U těchto typů je přetypování závislé na dědičnosti – lze přetypovávat na typ předka (implicitní přetypování), někdy lze přetypovávat na potomka.
Ukážeme si to na příkladu tříd `Ucet` a `ZiroUcet`. Instanci třídy `ZiroUcet` je možné používat všude, kde lze použít instanci třídy `Ucet`. Pro třídu `SeznamUctu` si ukážeme dvě metody – metodu `pridej()` pro přidání účtu a metodu `vypis()`, která bude vypisovat seznam účtů.

```
public class SeznamUctu {
     private List<Ucet> seznam;

     public void pridej(Ucet ucet) {
          seznam.add(ucet);
     }
     public void vypis() {
          for (Ucet ucet : seznam) {
               System.out.println(ucet.getVlastnik() + "\tstav: " +
                              ucet.getStav());
}}}
```

Nyní můžeme přidávat do seznamu jak obyčejné účty, tak i žirové účty. Pokud bychom však pro žirové účty chtěli vypisovat i limit kontokorentu, musíme v metodě `vypis()` pomoci operátoru `instanceof` zjišťovat, že se jedná o instanci třídy `ZiroUcet`, a tu poté explicitně přetypovat a zavolat metodu `getLimit()`.

```
    public void vypis() {
         for (Ucet ucet : seznam) {
              if (ucet instanceof ZiroUcet) {
                   ZiroUcet ziro = (ZiroUcet) ucet;
                   System.out.println(ziro.getVlastnik() + "\tstav: " +
                             ziro.getStav()+"\tlimit: "+ ziro.getLimit());
              }
              else {
                   System.out.println(ucet.getVlastnik() + "\t" +
                             ucet.getStav());
              }
         }
    }
```

Na další stránce si ukážeme vhodnější řešení metody *vypis()*, kdy místo rozskoku použijeme pro rozlišení polymorfismus.
Dalším příkladem přetypování může být metoda *equals()*, kterou jsme si uvedli u operátoru *instanceof* v kapitole 3:

```
    public boolean equals (Object o) {
         if (o instanceof Mistnost) {
              Mistnost druha = (Mistnost)o;
              return nazev.equals(druha.nazev);
         }
         else {
              return false;
         }
    }
```

Parametr metody *equals()* je typu *Object*, tj. lze uvést instanci libovolné třídy. Při volání metody *equals()* v následujícím kódu se automaticky přetypuje instance *komora* třídy *Mistnost* na typ *Object*, uvnitř metody *equals()* se přetypuje zpět na typ *Mistnost*.

```
    Mistnost komora = new Mistnost("komora", "sklad vedle kuchyně");
       // ... mnoho řádků kódu ...
    if (sousedniMistnost.equals(komora)) {
```

Vlastní instance se při přetypování nemění. Metody má instance přiřazeny v okamžiku vytvoření instance, žádným přetypováním nelze změnit přiřazení metod. Na druhou stranu přetypování ovlivňuje překlad – pokud potomka přetypujeme na předka, tak překladač bude kontrolovat, zda používáme pouze metody, které zná předek. Proto je např. nutné ve výše uvedené metodě *equals()* přetypovat *Object* zpět na místnost, aby byl dostupný datový atribut *nazev*.

11.5.2. Překrývání metod, pozdní vazba a polymorfismus

Do třídy *Ucet* doplníme metodu *getPopis()*:

```
    public String getPopis() {
         return ucet.getVlastnik() + "\tstav: " + ucet.getStav();
    }
```

Ve třídě *ZiroUcet* metoda *getPopis()* vrátí též *limit*:

```
    public String getPopis() {
         return ziro.getVlastnik() + "\tstav: " +
                   ziro.getStav()+"\tlimit: "+ ziro.getLimit();
    }
```

Pokud již máme takto připraveny třídy *Ucet* a *ZiroUcet*, bude metoda *vypis()* ve třídě *SeznamUctu* velmi jednoduchá.

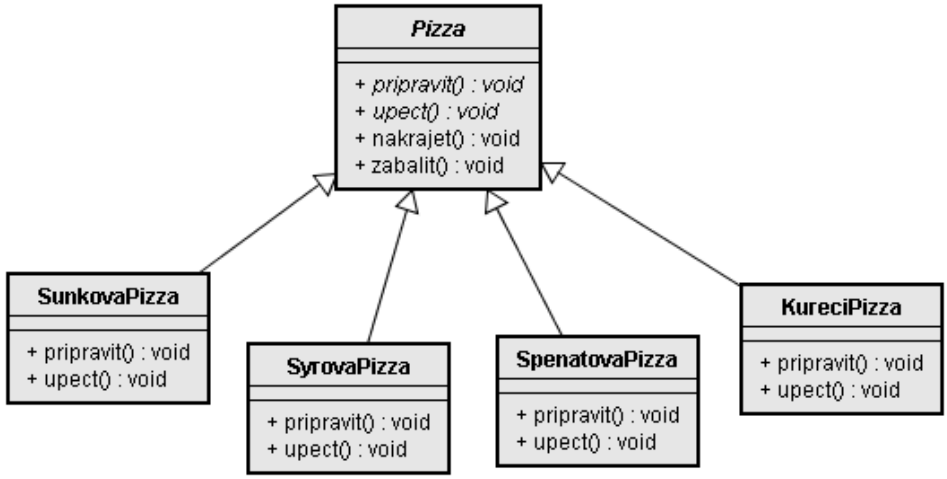
```
    public void vypis() {
         for (Ucet ucet : seznam) {
              System.out.println(ucet.getPopis());
         }
    }
```

Při tomto použití se již uplatňuje **polymorfismus** – volají se různé metody, byť to v kódu není přímo vidět. Pokud se ze seznamu vybere instance třídy `Ucet`, zavolá se metoda `getPopis()` ze třídy `Ucet`, pokud se vybere instance třídy `ZiroUcet`, zavolá se metoda `getPopis()` z této třídy.
Toto chování je umožněno **pozdní vazbou** (late binding či dynamic binding) – když se vytvoří instance, přiřazují se k ní jednotlivé metody. Pokud se některé metody v hierarchii dědičnosti překrývají, přiřadí se z nich ta metoda, která je nejblíže. Obvykle je to metoda stejné třídy. Pokud se volá nějaká metoda instance, hledá se v seznamu metod přiřazených instanci.
V některých programovacích jazycích se používá též **časná vazba** (early binding či static binding) – volaná metoda se přiřazuje při překladu. V našem příkladu s metodou `vypis()` by se při použití časné vazby vždy volala metoda `getPopis()` ze třídy `Ucet`, neboť při překladu se vybírá metoda dle uvedeného typu. V kódu je řečeno, že proměnná `ucet` je typu `Ucet`, tudíž se vezme metoda `getPopis()` z této třídy. Java časnou vazbu nepodporuje.

11.5.3. Příklad polymorfismu

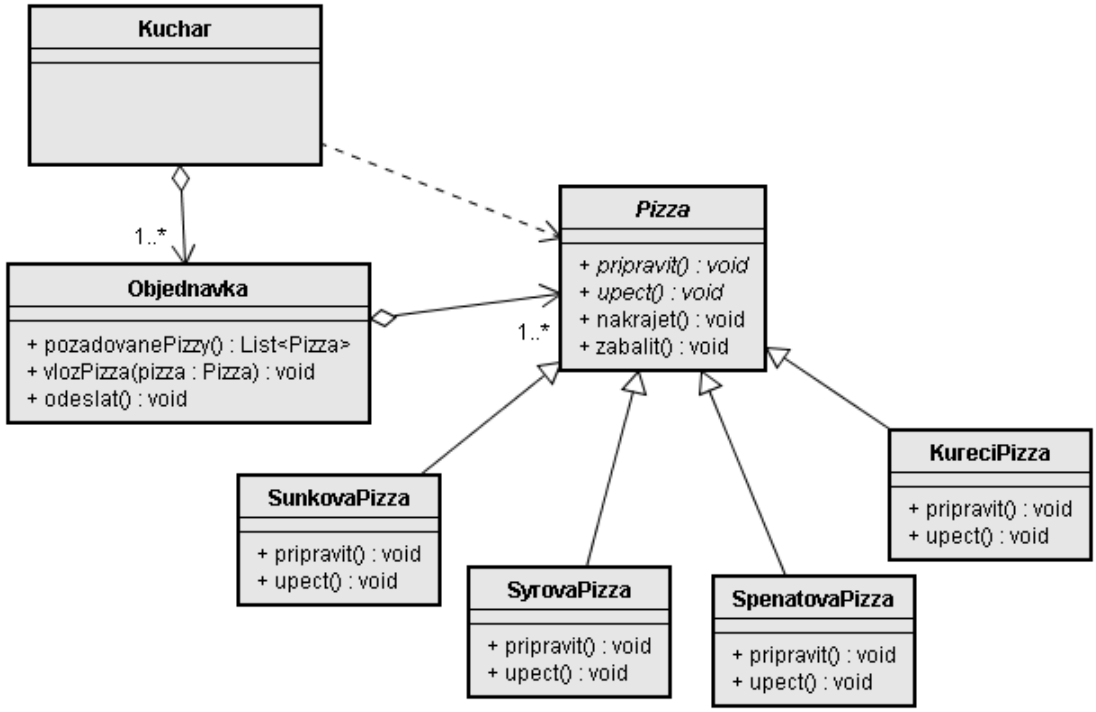
Následuje další příklad dědičnosti s abstraktní třídou `Pizza` a čtyřmi podřízenými konkrétními třídami. Ve třídě `Pizza` jsou dvě abstraktní metody (`pripravit()` a `upect()`) a dvě konkrétní metody (`nakrajet()` a `zabalit()`), viz diagram na obrázku 11.5.
Aby bylo možno mluvit o polymorfismu, musí existovat alespoň jedna další třída, která bude pracovat s větším počtem instancí různých potomků třídy Pizza a volat některé z metod, které jsou přetíženy v potomcích (tj. metody `pripravit()` a `upect()`).
Můžeme si představit, že máme zásobník objednávek, do kterého jsou vloženy jednotlivé objednávky na pizzu. Následuje kód, který představuje činnost kuchaře: vezme objednávku, z objednávky postupně připraví jednotlivé požadované pizzy a nakonec všechny pizzy z objednávky odešle.

```
    while (!zasobnikObjednavek.empty()) {
         Objednavka objednavka = zasobnikObjednavek.getObjednavka();
         for (Pizza pizza : objednavka.pozadovanePizzy()) {
              pizza.pripravit();
              pizza.upect();
              pizza.nakrajet();
              pizza.zabalit();
              objednavka.vlozPizza();
         }
         objednavka.odeslat();
    }
```



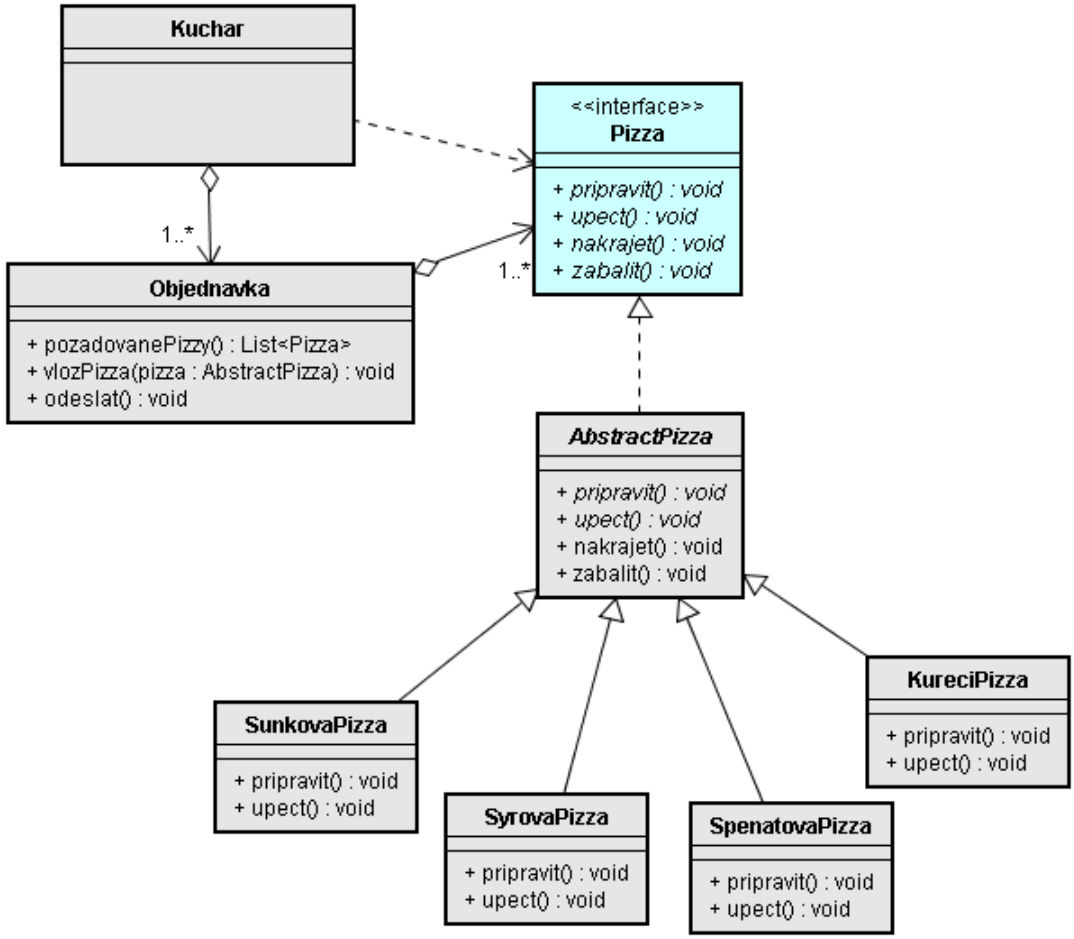
Obrázek 11.5 Abstraktní třída Pizza a její potomci

Použití polymorfismu se obvykle též projeví v diagramu tříd. Třída Kuchar se neodkazuje na jednotlivé instance, ale na předka, viz obrázek 11.6.



Obrázek 11.6 Vztah mezi třídami při polymorfismu

Použití polymorfismu s dědičností je velmi podobné polymorfismu při použití rozhraní. Z důvodů větší nezávislosti a pružnosti návrhu se obvykle navrhuje rozhraní, které definuje požadované chování (požadované metody). Toto rozhraní implementuje abstraktní třída – její hlavní význam je v úspoře kódu při psaní konkrétních potomků. Výsledná struktura je pro náš příklad s pizzou zobrazena na následujícím obrázku. Při pohledu do knihoven Javy zjistíme, že mnoho tříd je navrženo tímto způsobem, např. kolekce a mapy.



Obrázek 11.7 Rozhraní, abstraktní třída a konkrétní třídy

11.6. Použití dědičnosti

Dědičnost by se měla používat v situacích, kdy potomek je podtypem svého předka, tj. existuje mezi nimi vztah „je nějaký“ (v angličtině se používá „is-a“). Pokud má být nějaká třída B potomkem třídy A, měli bychom si kladně odpovědět na tyto otázky: „B je A?“ či „Je každý B také A?“. Nemůžeme-li na takou otázku odpovědět kladně, neměli bychom používat dědičnost. V případě bankovních účtů lze odpovědět kladně na výroky „Žirový účet je účtem?“ a „Je každý žirový účet také účtem?“. Důsledky použití dědičnosti v situaci, kdy je porušen vztah „je nějaký“, si ukážeme dále v této kapitole.
Vedle vztahu „je nějaký“ jsou vztahy, které lze vyjádřit pomocí „je částí“ („part-of“) a „má“ („has-a“). Pokud jsou mezi třídami tyto vztahy, nepoužívá se dědičnost, ale většinou kompozice.
Důležité je též si uvědomit, že dědičnost vyjadřuje vztah tříd, ne vztah instancí.
Pro objektový jazyk C++ s vícenásobnou dědičností se obvykle doporučuje, aby základem nějaké dědičné hierarchie byla abstraktní třída. Toto neplatí plně pro jazyky s rozhraními, jako je např. Java, kde se jako základ dědičné hierarchie obvykle používá rozhraní.

11.6.1. Důvody pro použití dědičnosti

Dědičnost se využívá v různých situacích, za různými účely. V praxi lze důvody pro použití dědičnosti obtížně odlišit. Zde si uvedeme tři nejčastější důvody k použití dědičnosti.

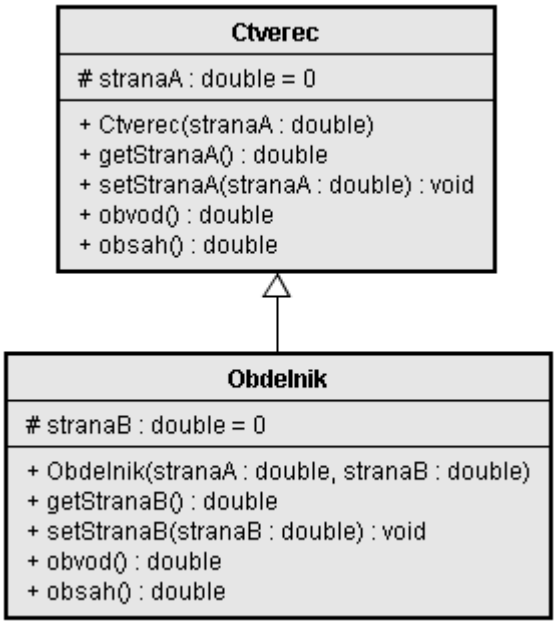
Specializace
Jedním z nejčastějších důvodů pro použití dědičnosti je specializace existujících tříd a objektů. Při specializaci získává třída nové datové atributy a chování proti původní třídě. Ukázkou specializace je příklad s bankovním účtem a žirovým účtem.
Jinou formou specializace je většina situací, kdy předkem je abstraktní třída – viz příklad s pizzou.

Překrývání metod a polymorfismus
Častým důvodem k dědičnosti je možnost využití překrývání metod a následně polymorfismu – různí potomci mají rozdílně implementována některá chování (některé metody). Při volání takové metody programátor nemusí uvažovat o tom, které konkrétní instanci posílá zprávu, neboť každá instance má k sobě přiřazen svůj specifický kód.

Znovupoužití kódu
Jednou z prvních motivací pro dědičnost bylo umožnit nové třídě znovu použít kód, který již existoval v jiné třídě.
Pokud vede k dědičnosti pouze tento motiv, vznikne hierarchie, kdy věcně nelze přetypovat potomka na předka. Příklad si uvedeme v podkapitole věnované chybám v návrhu dědičnosti. V těchto situacích se preferuje před dědičností jiná technika – kompozice. Příklad kompozice najdete ke konci této kapitoly.

11.6.2. Porušení vztahu „je nějakým“ při návrhu dědičnosti

Nejčastějším úskalím používání dědičnosti je situace, kdy programátor využívá dědičnosti pouze z důvodu úspory kódu a z tohoto důvodu poruší vztah „**is-a**“. Uvedeme si dva příklady.
V prvním příkladu vytváříme dvě třídy představující grafické tvary – třídu `Ctverec` a třídu `Obdelnik`. U třídy `Ctverec` postačuje jeden datový atribut – délka strany a, u obdélníku potřebujeme ještě stranu b. Tj. třída `Obdelnik` rozšiřuje třídu `Ctverec` a tudíž je kandidátem na potomka třídy `Ctverec`. Na následujícím diagramu jsou zobrazeny datové atributy a metody obou tříd.



Obrázek 11.8 Struktura tříd Ctverec a Obdelnik

Tyto třídy na první pohled fungují tak, jak mají – lze vytvářet čtverce i obdélníky, správně se počítají obvody i obsahy. Nyní dostaneme za úkol vytvořit metodu, která zdvojnásobí plochu čtverce:

```
public void zdvojnasobitPlochu(Ctverec ctvr) {
     ctvr.setStranaA(ctvr.getStranaA() * Math.sqrt(2));
}
```

Na první pohled metoda funguje dobře, problém nastane v okamžiku, kdy zavoláme tuto metodu s instancí třídy *Obdelnik*. Plochu obdélníka metoda nezdvojnásobí – metoda nedělá to, co jsme chěli. Problém spočívá v nesprávně použité dědičnosti, obdélník není čtvercem – není zde vztah „is-a“.
Druhý příklad chybného použití dědičnosti si ukážeme na příkladu ze standardních knihoven Javy. V balíčku *java.util* je třída *Stack*, která nabízí datovou strukturu zásobník. Zásobník má tři základní operace: vkládání prvků na vrchol zásobníků (*push*), odebírání prvků z vrcholu zásobníků (*pop*) a získání prvku z vrcholu zásobníku (*peek*). Pro realizaci zásobníku je výhodné použít seznam (*List* – viz kapitola 10), neboť ten již obsahuje metody pro vkládání prvků do seznamu, vyndávání prvků ze seznamu. Autor třídy *Stack* použil dědičnosti – třída *Stack* je potomkem třídy *Vector*, což je jedna z implementací seznamu v Javě. Tím, že je třída *Stack* potomkem seznamu jsou však porušena pravidla pro zásobník – ze třídy *Vector* se dědí další metody, které umožňují vkládat dovnitř zásobníku či vybírat zevnitř zásobníku. Tuto chybu v návrhu dědičnosti však již nelze odstranit, neboť takto definovaná třída *Stack* se již začala používat v mnoha aplikacích.

11.6.3. Dědičnost narušuje zapouzdření

Problémem používání dědičnosti je, že narušuje jinou základní objektovou vlastnost – zapouzdření. Programátor třídy potomka musí vědět nejen to, co dělají metody předka, ale i jak to dělají. Projevuje se to i v dokumentaci metod – u metod, které se mohou překrývat, by měl popis zahrnovat i vnitřní fungování metody.
Potomek je též velmi závislý na změnách rodičovské třídy. Představme si, že máme třídy *Kosodelnik* a *Obdelnik*. Obdélník je speciálním případem kosodélníku (úhel 90º), proto může být třída *Obdelnik* potomkem třídy *Kosodelnik*. Následuje jednoduchý návrh třídy *Kosodelnik* (úhel je potřeba zadávat v radiánech):

```
public class Kosodelnik {
     protected double stranaA = 0;
     protected double stranaB = 0;
     protected double uhel = 0;

     public Kosodelnik(double stranaA, double stranaB, double uhel) {
          this.stranaA = stranaA;
          this.stranaB = stranaB;
          this.uhel    = uhel;
     }
     public double obvod() {
          return 2 * (stranaA + stranaB);
     }
     public double obsah() {
          return stranaA * stranaB * Math.sin(uhel);
     }
}
```

Při využití dědičnosti bude třída *Obdelnik* poměrně krátká (úhel 90º odpovídá hodnotě polovina π v radiánech):

```
public class Obdelnik extends Kosodelnik {
     public Obdelnik(double stranaA, double stranaB) {
          super(stranaA, stranaB, Math.PI/2);
     }
}
```

Když se však nyní rozšíří třída `Kosodelnik` o metodu `setUhel()` pro změnu úhlu, tak dojde k narušení dědičnosti objektů[26]. Tuto metodu zdědí i potomek, po jejím zavolání u instance třídy `Obdelnik` potom najednou budeme mít obdélníky bez pravých úhlů. To však odporuje definici obdélníka.
Jedním řešením je doplnit do třídy `Obdelnik` metodu `setUhel()` a v ní vyvolat výjimku (např. `UnsupportedOperationException`). Tj. v této variantě je potřeba při změně předka upravit i (některé) potomky – to však předpokládá, že autor úpravy v předkovi má k dispozici všechny potomky.
Dalším řešením je nedávat do třídy `Kosodelnik` metodu `setUhel()`, tj. navrhnout ji jako read-only třídu obdobně jako u třídy `String`. Pokud se má změnit některý ze základních datových atributů jako úhel či strana, tak se vytvoří nová instance třídy `Kosodelnik`.
Třetím řešením je při návrhu třídy `Obdelnik` nepoužívat dědičnost, ale použít **kompozici** – ve třídě `Obdelnik` bude datový atribut typu `Kosodelnik` a většina volání metod třídy `Obdelnik` bude přesměrována na odpovídající metody třídy `Kosodelnik`:

```
public class Obdelnik {
     private Kosodelnik kosodelnik;
     public Obdelnik(double stranaA, double stranaB) {
          kosodelnik = new Kosodelnik(stranaA, stranaB, Math.PI/2);
     }
     public double obvod() {
          return kosodelnik.obvod();
     }
     public double obsah() {
          return kosodelnik.obsah();
     }
}
```

Nevýhodou varianty s kompozicí však je, že při používání těchto tříd nelze využít výhod polymorfismu (lze to obejít při vhodném využití rozhraní). Pokud se již dědičnosti mezi třídami `Kosodelnik` a `Obdelnik` někde používá, tak toto řešení nepřichází v úvahu.

[26] Zde máme na mysli narušení věcné/logické dědičnosti tříd, formální pravidla dědičnosti v Javě se tímto nenaruší.