Objektově orientované programování

Jiří Vokřínek

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 2

B0B36PJV – Programování v JAVA

Objektově orientované programování v Javě

Objektově orientované programování

Vztahy mezi objekty – dědičnost a polymorfismus

Položky třídy a instance

Konstruktor

Význam metody `main`

Neměnitelné objekty (Immutable objects)

Dědičnost a polymorfismus

- Dědičnost
- Kompozice
- Polymorfismus
- Příklad návrhu a využití polymorfismu
- Dispatch
- Double Dispatch

Část I

Objektově orientované programování v Javě

Objektově orientované programování (OOP)

OOP je přístup jak správně navrhnout strukturu programu tak, aby výsledný program splňoval funkční požadavky a byl dobře udržovatelný.

- **Abstrakce** – koncepty (šablony) organizujeme do tříd, objekty jsou pak instance tříd
- **Zapouzdření** (encapsulation)
  - Objekty mají svůj stav skrytý, poskytují svému okolí **rozhraní**, komunikace s ostatními objekty zasíláním zpráv (volání metod)
- **Dědičnost** (inheritance)
  - Hierarchie tříd (konceptů) se společnými (obecnými) vlastnostmi, které se dále specializují
- **Polymorfismus** (mnohotvárnost)
  - Objekt se stejným rozhraním může zastoupit jiný objekt téhož rozhraní.

Třídy a objekty

Objekty - reprezentují základní entity OO systému za jeho běhu.

- Mají konkrétní vlastnosti a vykazují chování
- V každém okamžiku lze popsat jejich stav
- Objekty se v průběhu běhu programu liší svým vnitřním stavem, který se během vykonávání programu mění

Třídy - popisují možnou množinou objektů. Předloha pro tvorbu objektů třídy. Mají:

- Rozhraní - definuje části objektů dané třídy přístupné zvenčí
- Tělo - implementuje operace rozhraní
- Instanční proměnné - obsahují stav objektu dané třídy

Třídy a objekty - vlastnosti

- **Zapouzdření** (encapsulation) je množina služeb, které objekt nabízí navenek.
  Odděluje rozhraní (**interface**) a jeho implementaci.
- **Stav** je určen daty objektu.
- **Chování** je určeno stavem objektu a jeho službami (metodami).
- **Identita** je odlišení od ostatních objektů (v prog. jazycích pojmenování proměnných reprezentující objekty určité třídy).

Třída

Popisuje množinu objektů – je jejich vzorem (předlohou) a definuje:

- Rozhraní – části, které jsou přístupné zvenčí

  public, **protected**, **private**, *package*

- Tělo – implementace operací rozhraní (metod), které určují schopnosti objektů dané třídy

  instanční vs statické (třídní) metody

- Datové položky – atributy základních i složitějších datových typů a struktur

  kompozice objektů

  - Instanční proměnné – určují stav objektu dané třídy
  - Třídní (statické) proměnné – společné všem instancím dané třídy

Struktura objektu

- Objekt je kombinací dat a funkcí, které pracují nad těmito daty
- Objekt je tvořen
  - **Datovými strukturami** – atributy
    - Ovlivňují vlastnosti objektu
    - Jsou to proměnné různých datových typů
    - Data jsou zpravidla přístupná pouze v rámci daného objektu a zvnějšku jsou skryta před jinými objekty

      Zapouzdření (encapsulation) / „gettery a settery"
  - **Metodami** – funkce / procedury
    - Určují chování objektu
    - Definují operace nad daty objektu
    - Metody představují služby objektu, proto jsou často veřejné

      Mohou být deklarovány jako privátní
- **Objekt** je instance třídy
  - V Javě lze vytvářet pouze dynamicky operátorem **new**
  - **Referenční proměnná**

    Hodnota proměnné „odkazuje" na místo v paměti, kde je objekt uložen

Princip zapouzdření (Encapsulation)

- „Utajení" vnitřního stavu objektu
- Jiné objekty nemohou měnit stav objektu přímo a způsobit tak chybu
- Metody objektu umožňují objektu komunikovat se svým okolím, tvoří jeho **rozhraní**
- Proměnné (data) objektu nejsou z vnějšku objektu přístupné, pro přístup k nim lze využít pouze metody
- Zapouzdření umožňuje udržovat a spravovat každý objekt nezávisle na jiném objektu. Umožňuje **modularitu** zdrojových kódů.

Datové položky objektů

- Dle principu zapouzdření jsou datové položky zpravidla **private**
- Přístup k položkám je přes metody, tzv. „accessory", které vrací/nastavují hodnotu příslušné proměnné („getter" a „setter")

```
public class DemoGetterSetter {
    private int x;
}
```

- „Accessory" lze vytvořit mechanicky a vývojová prostředí zpravidla nabízí automatické vygenerování jejich zdrojového kódu

```
public class DemoGetterSetter {
    private int x;

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }
}
```

Viz Alt+Insert v Netbeans

Vztahy mezi objekty

- V OO systému interagují objekty mezi sebou prostřednictvím zasílání zpráv (messages) požadavků na provedení služeb poskytovaných objektem
  1. Po obdržení zprávy objekt vyvolá požadovanou metodu
  2. Případně zašle výsledek
- Objekt poskytující službu se často nazývá *server*
- Objekt žádající o službu se nazývá *klient*
- Mezi objekty je relace–asociace, volá-li objekt služby jiného objektu
- S relacemi mezi objekty souvisí viditelnost a vazby mezi objekty

Datové položky třídy a instance

- Datové položky třídy
  - Jsou společné všem instancím vytvořeným z jedné třídy
  - Nejsou vázaný na konkrétní instanci
  - Jsou společné všem instancím třídy
  - V Javě jsou uvozeny klíčovým slovem **static**
- Datové položky instance
  - Tvoří vlastní sadu datových položek objektu
  - Jsou to tzv. proměnné instance
  - Jsou iniciovány při vytvoření instance

    V konstruktoru při vytvoření instance voláním `new`
  - Existují po celou dobu života instance
  - Proměnné jedné instance jsou nezávislé na proměnných instance jiné

Metody třídy a instance

- Metody třídy
  - Nejsou volány pro konkrétní instance
  - Představují zprávu zaslanou třídě jako celku
  - Mohou pracovat pouze s proměnnými třídy

    Nikoliv s proměnnými instance
  - V Javě jsou uvozeny klíčovým slovem **static**
  - Jsou to tzv. statické metody
- Metody instance
  - Jsou volány vždy pro konkrétní instanci třídy
  - Představují zprávu zaslanou konkrétní instanci
  - Pracují s proměnnými instance i s proměnnými třídy
  - Lze volat pouze až po vytvoření konkrétní instance

Přístup ke členům třídy

- Podle principu zapouzdření jsou některé členy třídy označována jako soukromé (privátní) a jiné jako veřejné.
- Programátor předepisuje k jakým položkám lze přistupovat a modifikovat je
- Přístup ke členům třídy je určen modifikátorem přístupu
  - **public:** – přístup z libovolné třídy
  - **private:** – přístup pouze ze třídy, ve které byly deklarovány
  - **protected:** – přístup ze třídy a z odvozených tříd
  - Bez uvedení modifikátoru je přístup povolen v rámci stejného balíčku package

Řízení přístup ke členům třídy

Modifikátor	Přístup			
	Třída	Balíček	Odvozená třída	„Svět"
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
bez modifikátoru	✓	✓	✗	✗
private	✓	✗	✗	✗

http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Vytvoření objektu – Konstruktor třídy

- Instance třídy (objekt) vzniká voláním operátoru **new** s argumentem jména třídy, který volá **konstruktor** třídy
- Konstruktor nemá návratový typ, jmenuje se stejně jako třída a můžeme jej přetížit pro různé typy a počty parametrů
- Jiný konstruktor třídy lze volat operátorem **this**

  *Operátorem **super** lze volat konstruktor nadřazené třídy*
- Není-li konstruktor předepsán, je vygenerován konstruktor s prázdným seznamem parametrů
  - Je-li konstruktor deklarován, implicitní zaniká
- Přetížení konstruktoru pro různé typy a počty parametrů

  overloading
- Konstruktor je zpravidla vždy **public**
- Privátní (**private**) konstruktor použijeme například pro:
  - Třídy obsahující pouze statické metody nebo pouze konstanty

    Zakážeme tak vytváření instancí.
  - Takzvané singletony (singletons)

Statická metoda **main**

- Deklarace hlavní funkce

  public static void main(String[] args) { ... }

  představuje „spouštěč" programu
- Musí být statická, je volána dříve než se vytvoří objekt
- Třída nemusí obsahovat funkci **main**
  - Taková třída zavádí prostředky, které lze využít v jiných třídách
  - Jedná se tak o „knihovnu" funkcí a procedur nebo datových položek (konstant)
- Kromě spuštění programu může funkce **main** obsahovat například testování funkčnosti objektu nebo ukázku použití metod objektu

Neměnitelné objekty (Immutable objects)

- Definice neměnitelného objektu
  - Všechny datové položky jsou **final** a **private**
  - Neimplementujeme „settery" pro modifikaci položek
  - Zákaz přepisu metod v potomcích (**final** modifikátor u metod)

*Nebo jednoduše zákaz odvozování uvedením **final class***

http://docs.oracle.com/javase/tutorial/essential/concurrency/imstrat.html

- Objekty, které v průběhu života nemění svůj stav
- Modifikace objektu není možná a je nutné vytvořit objekt nový
- Mají výhodu v případě paralelního běhu více výpočetních toků

*Typickým příkladem je třída **String***

http://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

Příklad – **final** nezaručuje neměnitelnost objektu 1/3

- Definujme **final** třídu zapouzdřující referenční **private final** proměnnou `values` typu odkaz na pole `int` hodnot
- Přístup k proměnné `values` je pouze přes *getter* `getArray()`

```
public final class ArrayWrapper {

  private final int[] values;

  public ArrayWrapper(int n) {
    values = new int[n];

    for (int i = 0; i < values.length; i++) {
      values[i] = (int) (Math.random() * n);
    }
  }

  public int[] getArray() {
    return values;
  }

  public String toString() {
    ...
  }
}
```

lec02/ArrayWrapper

Příklad – **final** nezaručuje neměnitelnost objektu 2/3

- Obsah objektu `ArrayWrapper` vypíšeme metodou `toString()`

```
public final class ArrayWrapper {
  ...
  public String toString() {
    ...
    StringBuilder sb = new StringBuilder(values.length
     > 0 ? new Integer(values[0]).toString() : "empty"
    );
    for (int i = 1; i < values.length; i++) {
      sb.append(" ");
      sb.append(values[i]);
    }
    return sb.toString();
  }
}
```

lec02/ArrayWrapper

Příklad – **final** nezaručuje neměnitelnost objektu 3/3

```
final ArrayWrapper a = new ArrayWrapper(10);

System.out.println("Random final array '" + a + "'");
```

- Po vytvoření objektu `ArrayWrapper` je obsah pole incializován v konstruktoru na náhodná čísla

  ```
  Random final array '1 5 5 9 4 9 7 1 8 1'
  ```

- Přístup k položkám pole máme přes *getter* `getArray()`

  ```
  final int[] v = a.getArray();
  for (int i = 0; i < v.length; i++) {
    v[i] = i;
  }
  System.out.println("Final object and final array can
      be still modified\n '" + a + "'");
  ```

- Původní „final" objekt tak můžeme změnit

  ```
  Final object and final array can be still modified
   '0 1 2 3 4 5 6 7 8 9'
  ```

lec02/FinalArrayDemo

Referenční datové položky neměnitelných objektů musí odkazovat na **neměnitelné objekty**.

Objekty pro základní typy

- Třídy pro základní třídy jsou immutable
- Každý primitivní typ má v Javě také svoji třídu:
  - **Char**, **Boolean**
  - **Byte**, **Short**, **Integer**, **Long**
  - **Float**, **Double**
- Třídy obsahují metody pro převod čísel a metody pro parsování čísla z textového řetězce
  - např. **Integer.parseInt(String s)**
- Dále také rozsah číselného typu minimální a maximální hodnoty
  - např. **Integer.MAX_VALUE**, **Integer.MIN_VALUE**

http://docs.oracle.com/javase/8/docs/api/java/lang/Number.html

Referenční proměnné objektů tříd primitivních typů

- Referenční proměnné objektů pro primitivní typy můžeme používat podobně jako základní typy

```
Integer a = 10;
Integer b = 20;

int r1 = a + b;
Integer r2 = a + b;
System.out.println("r1: " + r1 + " r2: " + r2);
```

- Stále to jsou však referenční proměnné (odkazující na adresu v paměti)
- Obsah objektu nemůžeme měnit, jedná se o **immutable** objekty

DemoObjectsOfBasicTypes.java

Část II

Dědičnost a polymorfismus

Polymorfismus

- Polymorfismus – mnohoznačnost / mnohotvárnost

  Vlasnost, která nám umožňuje pojmenovat nějakou konkrétní schopnost (metodu) identickým jménem, přičemž její implementace se může v jednotlivých třídách hierarchie tříd lišit.

- Základním způsobem realizace polymorfismu jsou
  - **Dědičnost** (inheritance)
  - Virtuální metody – dynamické vázání jména metody ke konkrétnímu objektu
  - Rozhraní (*interface)* a abstraktní třídy (*abstract*)
  - Překrývání metod (**override**)

Základní vlastnosti dědičnosti

- Dědičnost je mechanismus umožňující
  - Rozšiřovat datové položky tříd nebo je také modifikovat
  - Rozšiřovat nebo modifikovat metody tříd
  - Vytvářet hierarchie tříd
  - „*Předávat*" datové položky a metody k rozšíření a úpravě
  - **Specializovat** („*upřesňovat*") třídy

protected

- Mezi hlavní výhody dědění patří:
  - Zásadním způsobem přispívá k znovupoužitelnosti programového kódu
  - **Dědičnost je základem polymorfismu**

Spolu s principem zapouzdření

Příklad – Kvádr je rozšířený **obdélník**?

```
class Rectangle {
    protected double width;
    protected double height;

    Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public double getWidth() { return width; }
    public double getHeight() { return height; }

    public double getDiagonal() {
        return Math.sqrt(width*width + height*height);
    }
}
```

Příklad – **Kvádr** je rozšířený obdélník?

```
class Cuboid extends Rectangle {
    protected double depth;

    Cuboid(int width, int height, int depth) {
        super(with, height); //konstruktor predka
        this.depth = depth;
    }

    public double getDepth() { return depth; }

    @Override
    public double getDiagonal() {
        double tmp = super.getDiagonal(); //volani predka
        return Math.sqrt(tmp*tmp + depth*depth);
    }
}
```

Příklad dědičnosti – 1/2

- Třída **Cuboid** je rozšířením třídy **Rectangle** o hloubku (**depth**)
- Potomka deklarujeme rozšířením **extends**
  - `Cuboid` přebírá datové položky **width** a **height**
  - `Cuboid` také přebírá „gettery" **getWidth** a **getHeight**
  - Konstruktor se nedědí, lze ale volat v podtřídě operátorem **super**
  - Není-li konstruktor deklarován, je volán konstruktor bez parametrů
    - Konstruktor existuje vždy, buď implicitní nebo uživatelský
- Potomek doplňuje datové položky o **depth** a mění metodu **getDiagonal**

Příklad dědičnosti – 2/2

- Objekty třídy **Cuboid** mohou využívat proměnné `width`, `height` a `depth`
- Metoda **getDiagonal** překrývá původní metodu definovanou v nadřazené třídě **Rectangle**

zastínění – overriding

- Přístup k původní metodě předka je možný přes operátor **super**

- Má-li metoda stejného jména jiné parametry (počet/typ) jedná se o přetížení – overloading

Jedná se o jinou (novou) metodu!

Dědičnost – Kvádr je rozšířený obdélník

- V příkladu jsme rozšiřovali obdélník a vytvořili „specializaci" kvádr

Je to skutečně vhodné rozšíření?

Jaká je plocha kvádru? Jaký je obvod kvádru?

Dědičnost – Obdélník je speciální **kvádr**?

- Obdélník je kvádr s nulovou hloubkou

```
class Cuboid {
   protected double width;
   protected double height;
   protected double depth;
   Cuboid(int w, int h, int d) {
      this.width = w; this.height = h; this.depth = d;
   }
   public double getWidth() { return width; }
   public double getHeight() { return height; }
   public double getDepth() { return depth; }
   public double getDiagonal() {
      double tmp =
         width*width + height*height + depth*depth;
      return Math.sqrt(tmp);
   }
   }
```

Dědičnost – **Obdélník** je speciální kvádr?

```
class Rectangle extends Cuboid {
    Rectangle(int width, int height) {
        super(width, height, 0);
    }
}
```

- Obdélník je „kvádrem" s nulovou hloubkou
- Potomek se deklaruje klíčovým slovem **extends**
  - **Rectangle** přebírá všechny datové položky **width**, **height** a **depth**
  - a také přebírá všechny metody předka (přístupné mohou být, ale pouze některé)
  - Konstruktor je přístupný přes volání **super** a hodnota proměnné **depth** se nastavujeme na nulu
- Objekty třídy **Rectangle** mohou využívat všech proměnných a metod třídy **Cuboid**

Je obdélník potomek kvádru nebo kvádr potomek obdélníka?

1. Kvádr je potomek obdélníka
   - Logické přidání rozměru, ale metody platné pro obdélník nefungují pro kvádr

   obsah obdélníka
2. Obdélník je potomek kvádru
   - Logicky správná úvaha o specializaci: „vše co funguje pro kvádr funguje i pro kvádr s nulovou hloubkou"
   - Neefektivní implementace – každý obdélník je reprezentován 3 rozměry

Specializace je správná

*Vše co platí pro **předka**, musí platit pro **potomka***

V tomto konkrétním případě je však použití dědičnosti diskutabilní.

Vztah předka a potomka je typu „is-a"

- Je úsečka potomek bodu?
  - Úsečka nevyužije ani jednu metodu bodu
    is-a?: úsečka je bod? → **NE** → úsečka není potomek bodu
- Je obdélník potomek úsečky?
  is-a?: **NE**
- Je obdélník potomek čtverce nebo naopak?
  - Obdélník rozšiřuje čtverec o další rozměr, ale není čtvercem
  - Čtverec je obdélník, který má šířku a výšku stejnou

Nastavení délek stran v konstruktoru!

Substituční princip

- Vzájemný vztah mezi dvěma odvozenými třídami
- Zásady:
  - Odvozená třída je specializaci nadřazené třídy

    Existuje vztah is-a
  - Všude, kde lze použít třídu musí být použitelný i její potomek a to tak, aby uživatel nepoznal rozdíl

    Polymorfismus
  - Vztah **is-a** musí být trvalý

Kompozice objektů

- Obsahuje-li deklarace třídy členské proměnné objektového typu, pak se jedná o **kompozici objektů**
- Kompozice vytváří hierarchii objektů – nikoliv však dědičnost

  Dědičnost vytváří také hierarchii vztahů, ale ve smyslu potomek/předek.
- Kompozice je vztah objektů **agregace** – **je tvořeno** / **je součástí**
- Jedná se o strukturu typu „has-a”

Příklad kompozice 1/3

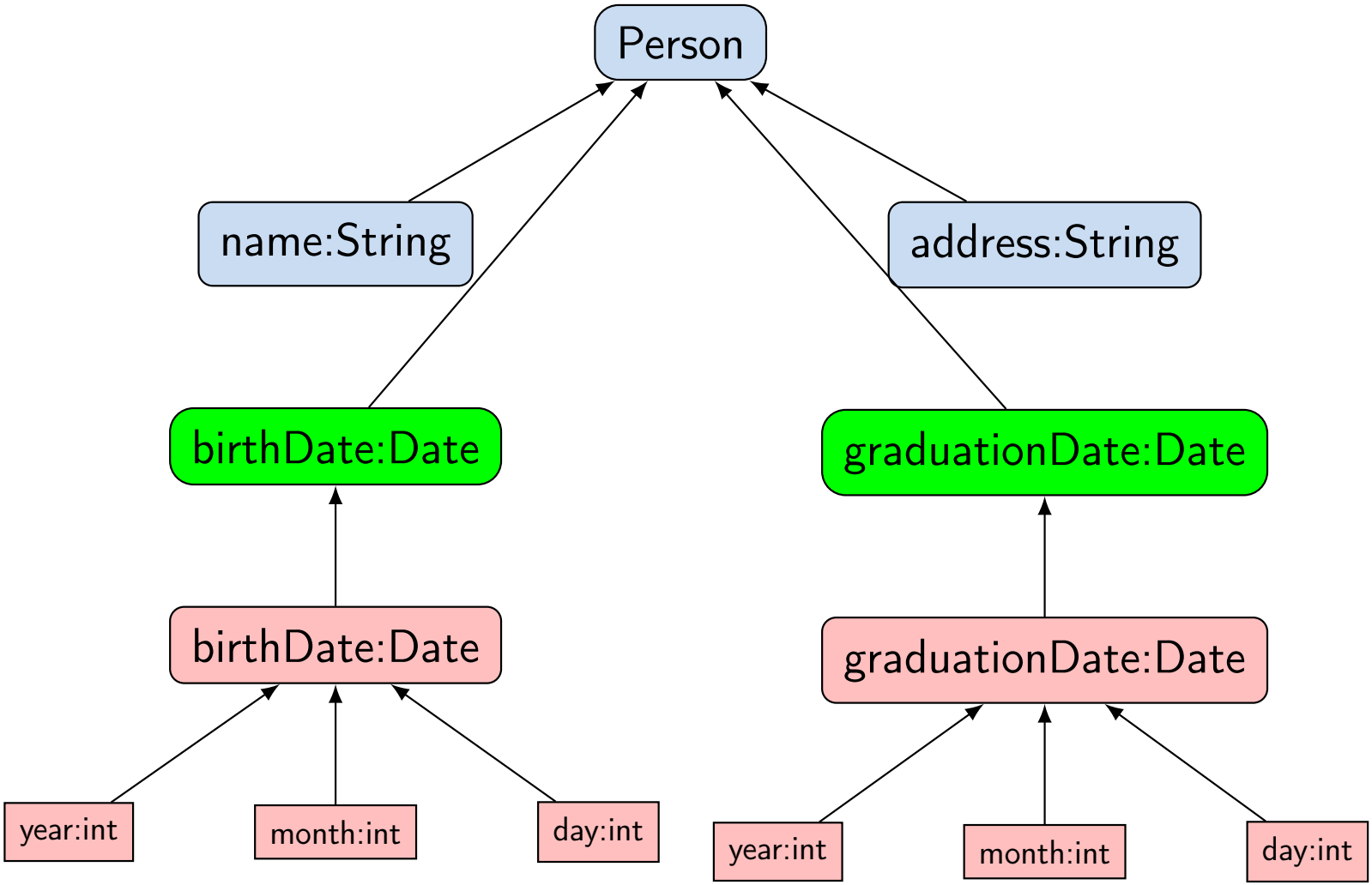
- Každá osoba je charakterizována atributy třídy `Person`
  - Jméno – `name`
  - Adresa – `address`
  - Datum narození – `birthDate`
  - Datum úspěšného ukončení studia – `graduationDate`
- Datum je charakterizováno třemi atributy (třída `Date`)
  - Den – day (`int`)
  - Měsíc – month (`int`)
  - Rok – year (`int`)

Příklad kompozice 2/3

```
class Person {
    String name;
    String address;
    Date birthDate;
    Date graduationDate;
}
```

```
class Date {
    int day;
    int month;
    int year;
}
```

Příklad kompozice 3/3



Dědičnost vs kompozice

- Vlastnosti dědění objektů:
  - Vytváření odvozené třídy (potomek, podtřída)
  - Podtřída se vytváří **extends**
  - Odvozená třída je specializací nadřazené třídy
    - Přidává proměnné *Nebo také překrývá proměnné*
    - Přidává nebo modifikuje metody
  - Na rozdíl od kompozice mění vlastnosti objektů
    - Nové nebo modifikované metody
    - Přístup k proměnným a metodám předka (bázové třídě, supertřídě)

      Pokud je přístup povolen (public/protected/„package")
- Kompozice objektů je tvořena atributy objektového typu

  Skládá objekty
- Rozlišení mezi kompozicí nebo děděním (pomůcka)
  - „Je" test – příznak dědění (is-a)
  - „Má" test – příznak kompozice (has-a)

Dědičnost a kompozice – úskalí

- Přílišné používání kompozice i dědičnosti v případech, kdy to není potřeba vede na příliš složitý návrh
- Pozor na doslovné výklady vztahu is-a a has-a, někdy se nejedná ani o dědičnost, ani kompozici

  Např. Point2D a Point3D nebo Circle a Ellipse
- Dáváme přednost kompozici před dědičností

  Jedna z výhod dědičnosti je ***polymorfismus***
- Při používání dědičnosti dochází k porušení zapouzdření

  Zejména s nastavením přístupových práv `protected`

Odvozené třídy, polymorfismus a praktické důsledky

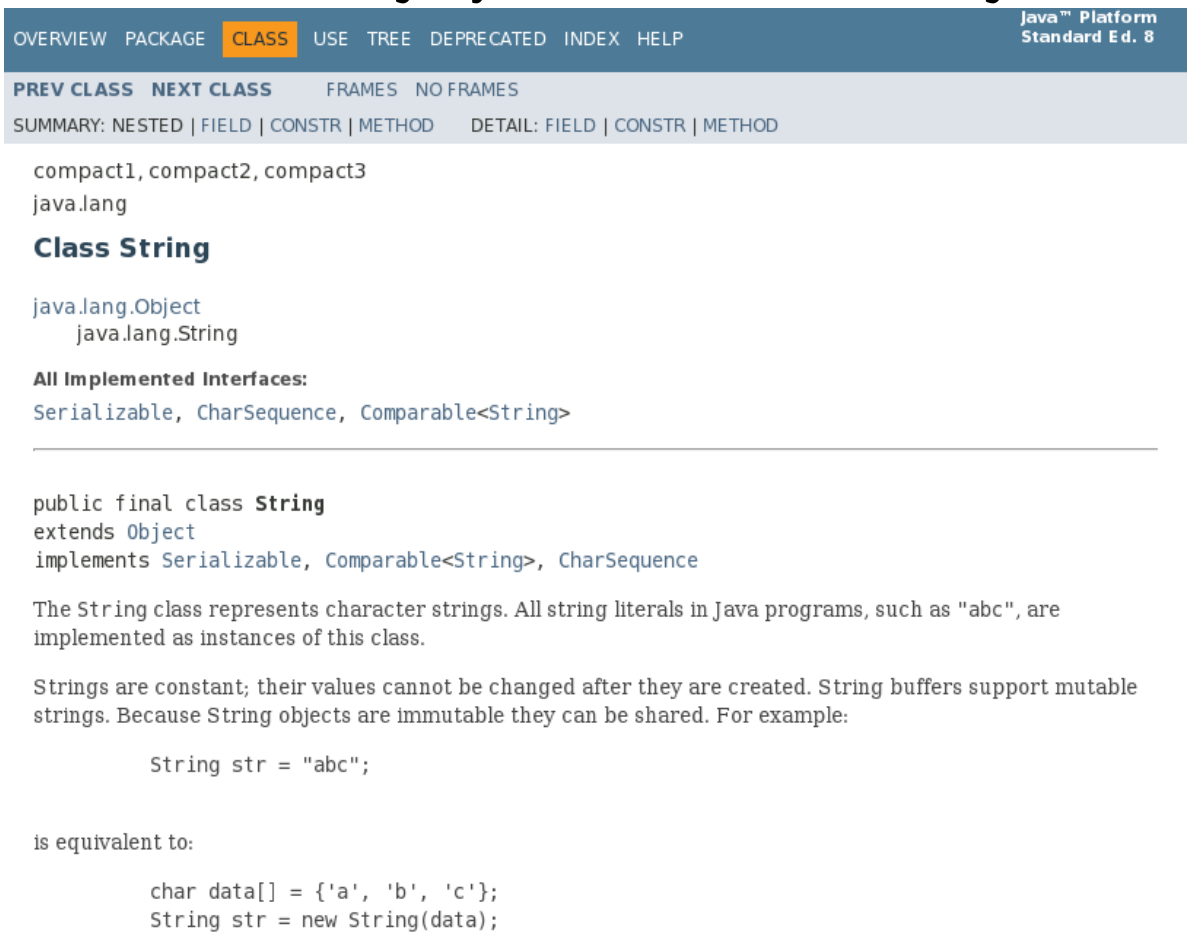
- Odvozená třída dědí metody a položky nadtřídy, ale také může přidávat položky nové
  - Můžeme rozšiřovat a specializovat schopnosti třídy
  - Můžeme modifikovat implementaci metod
- Objekt odvozené třídy může „vystupovat" místo objektu nadtřídy
  - Můžeme například využít efektivnější implementace aniž bychom modifikovali celý program.
  - Příklad různé implementace maticového násobení

viz `Matrix.java`, `DemoMatrix.java`

Projděte si samostatnou prezentaci a příklady.

Hierarchie tříd v knihovně JDK

- V dokumentaci jazyka Java můžeme najít následující obrázek



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/String.html

Třída `String`

- Třída `String` je odvozena od třídy `Object`
- Třída implementuje rozhraní `Serializable`, `CharSequence` a `Comparable<String>`
- Třída je **final** – tj. nemůže být od ní odvozena jiná třída

```
public final class String extends Object {
```

- Třída je **Immutable** – její datové položky nelze měnit

Třída `Object`

- Třída `Object` tvoří počátek hierarchie tříd v Javě
- Tvoří nadtřídu pro všechny třídy
- Každá třída je podtřídou (je odvozena od) `Object`
  - **class A {}** je ekvivalentní s **class A extends Object {}**
- Třída **Object** implementuje několik základních metod:
  - **protected Object clone();**

    Vytváří kopii objektu
  - **public boolean equals(Object o);**
  - **Class<?> getClass();**
  - **int hashCode();**
  - **public String toString(o);**

    Vrací textovou reprezentaci objektu
- Také implementuje metody pro synchronizací vícevláknových programů: **wait**, **notify**, **notifyAll**

  Každý objekt je také tzv. „monitorem".

https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html

Metoda toString()

- Metodou je zavedena implicitní typová konverze z typu objektu na řetězec reprezentující konkrétní objekt, např. pro tisk objektu metodou print

Lze využít automatické vytvoření v Netbeans

- Například metoda **toString** ve třídách Complex a Matrix

```
public class Complex {
    ...
    @Override
    public String toString() {
        if (im == 0) {
            return re + "";
        } else if (re == 0) {
            return im + "i";
        } else if (im < 0) {
            return re + " - " + (-im) + "i";
        }
        return re + " + " + im + "i";
    }
}
```

Complex.java, Matrix.java

Metoda equals()

- Standardní chování neporovnává obsah datových položek objektu, ale reference (adresy)

```
public boolean equals(Object obj) {
   return this == obj;
}
```

- Při zastínění můžeme porovnávat obsah datových položek, např.

```
public class Complex {
   @Override
   public boolean equals(Object o) {
      if (! (o instanceof Complex)) {
         return false;
      }
      Complex c = (Complex)o;
      return re == c.re && im == c.im;
} }
```

- Pro zjištění, zdali je referenční proměnná instancí konkrétní třídy můžeme použít operátor **instanceof**

Metody `equals()` a `hashCode()`

- Pokud třída modifikuje metodu `equals()` je vhodné také modifikovat metodu `hashCode()`
- Metoda `hashCode()` vrací celé číslo reprezentující objekt, které je například použito v implementaci datové struktury `HashMap`
- Pokud metoda `equals()` vrací pro dva objekty hodnotu `true` tak i metoda `hashCode()` by měla vracet stejnou hodnotu pro oba objekty
- Není nutné, aby dva objekty, které nejsou totožné z hlediska volání `equals`, měly nutně také rozdílnou návratovou hodnotu metody `hashCode()`

Zlepší to však efektivitu při použití tabulek HashMap.

Příklad geometrických objektů a jejich vizualizace

Projděte si samostatnou prezentaci a příklady.

Polymorfismus

Polymorfismus

- Polymorfismus (mnohotvárnost) se v OOD projevuje tak, že se můžeme stejným způsobem odvolávat na různé objekty
- Pracujeme s objektem, jehož skutečný obsah je dán okolnostmi až za běhu programu
- Polymorfismus objektů - Nechť třída ***B*** je podtřídou třídy ***A***, pak objekt třídy ***B*** můžeme použít všude tam, kde je očekáván objekt třídy ***A***
- Polymorfismus metod - Vyžaduje dynamické vázání, statický a dynamický typ třídy
  - Nechť třída ***B*** je podtřídou třídy ***A*** a redefinuje metodu *m()*
  - Proměnná *x* statického typu ***B***, dynamický typ může být ***A*** nebo ***B***
  - Jaká metoda se skutečně volá pro *x.m()* závisí na dynamickém typu

Dědičnost, polymorfismus a virtuální metody

- Vytvoření dynamické vazby je zpravidla v OO programovacím jazyce realizováno virtuální metodou
- Redefinované metody, které jsou označené jako virtuální, mají dynamickou vazbu na konkrétní dynamický typ

- V Javě jsou všechny metody deklarovány jako virtuální; „výjimku" tvoří
  - statické metody – volány se jménem třídy
  - skryté metody – pragmaticky na ně není přístup
  - metody deklarované s klíčovým slovem **final**

    nedovoluje překrývat metody v potomcích
  - metody deklarované ve třídě **final**

    nedovoluje od třídy odvozovat další třídy

`http://docs.oracle.com/javase/tutorial/java/IandI/final.html`

*V konstruktoru bychom měli volat pouze **final** metody, tak bude objekt inicializován podle zamýšleného způsobu*

Příklad přepsání inicializační metody 1/2

- Ve konstruktoru třídy `BaseClass` voláme metodu `doInit()`

```
public class BaseClass {
  public BaseClass() {
    doInit();
  }

  public void doInit() {
    System.out.println("Initialization of BaseClass");
} }
```

lec02/BaseClass

- `doInit()` přepíšeme v odvozené třídě `DerivedClass`

```
public class DerivedClass extends BaseClass {
  public DerivedClass() {
    super();
  }

  @Override
  public void doInit() {
    System.out.println("Init. of DerivedClass");
} }
```

lec02/DerivedClass

Příklad přepsání inicializační metody 2/2

- Po vytoření objektu třídy `DerivedClass` voláme konstruktor nadřazené třídy `super`
- Vlivem dynamické vazby se však volá implementace `doInit()` třídy `DerivedClass` a nikoliv původní třídy `BaseClass`

```
public class ConstructorDemo {
    public static void main(String[] args) {
        System.out.println(
                "Creating new instance of DerivedClass
     will not call the initilization of the BaseClass
     due to overrided doInit() in the DerivedClass");
        DerivedClass obj = new DerivedClass();
    }
}
```

lec02/ConstructorDemo

- Po spuštění se proto vypíše „*Initialization of the DerivedClass*" a nikoliv řetězec uvedený v `BaseClass`
- Proto pokud je nutné zajistit správnou inicializaci nadřazené třídy voláme v konstruktoru pouze **final** metody.

Vytvoření dynamické vazby – dědičnost

- Děděním vytváříme vazbu mezi nadřazenou a odvozenou třídou
- Za běhu programu se můžeme na odvozenou třídu „dívat" jako na nadřazenou třídu
  - Voláme metody identického jména za běhu je však určena konkrétní instance třídy a je vykonána příslušná implementace
- Vytvoření vazby můžeme provést:
  - Odvozením třídy od nadřazené třídy
  - Odvozením třídy od **abstraktní** třídy
  - Implementací rozhraní (**interface**)
- Příklad volání metody **doStep** objektu reprezentujícího hráče hrající nějakou konkrétní strategií:

```
Player player = new RandomPlayer();
player.doStep();
player = new BestPlayer();
player.doStep();
```

Příklad odvození třídy

- Nadřazená třída

```
public class Player {
    public void doStep() {
        // do some default strategy
    }
}
```

- Odvozená třída

```
public class RandomPlayer extends Player {
    public void doStep() {
        // do a random strategy
    }
}
```

Abstraktní třída

- Deklarace třídy se uvozuje klíčovým slovem **abstract**
- Abstraktní třída umožňuje deklarovat abstraktní metody (opět klíčovým slovem **abstract**)

  Abstraktní metody se mohou vyskytovat pouze v abstraktních třídách, jsou protikladem finálních metod, které nelze předefinovat.
- Abstraktní metody nemají implementaci a je nutné ji definovat v odvozených třídách

  Kontrola a podpora objektového návrhu na úrovni jazyka

- Lze je využít například pro vytvoření společného předka hierarchie tříd, které mají mít společné vlastnosti (bez konkrétní implementace), případně doplněné o datové položky

Příklad odvození od abstrakní třídy

- Nadřazená abstrakní třída

```
public abstract class Player {
    public abstract void doStep();
}
```

- Odvozená třída

```
public class RandomPlayer extends Player {
    @Override
    public void doStep() {
        // specific strategy
    }
}
```

- Explicitně uvádíme, že metodu přepisujeme
- Lze vytvořit referenční proměnnou abstraktní třídy, ale vytvořit instanci abstraktní třídy nelze

Rozhraní třídy – **interface**

- V případě potřeby „dědění" vlastností více předků lze využít rozhraní **interface**

  Řeší vícenásobnou dědičnost

- Rozhraní definuje množinu metod, které třída musí implementovat, pokud implementuje (**implemets**) dané rozhraní

  Garantuje, že daná metoda je implementována, neřeší však jak

- Rozhraní poskytuje specifický „pohled" na objekty dané třídy

  Můžeme přetypovat na objekt příslušného rozhraní

- Třída může implementovat více rozhraní

  Na rozdíl od dědění, u kterého může dědit pouze od jediného přímého předka

- Případnou „kolizi" shodných jmen metod více rozhraní řeší programátor

Příklad implementace rozhraní

- Rozhraní

```
public interface Player {
    public void doStep();
}
```

- Třída implementující dané rozhraní

```
public class RandomPlayer implements Player {
    @Override
    public void doStep() {
        // specific strategy
    }
}
```

Abstraktní třída nebo rozhraní

- **Abstraktní třída** je vhodná pro případy:
  - Odvozené třídy sdílejí implementaci
  - Odvozené třídy vyžadují přístup na položky, které nejsou **public**
- **Rozhraní** je výhodné pokud:
  - Očekáváme, že rozhraní bude implementováno v jiných, nesouvisejících třídách
  - Chceme specifikovat chování konkrétního datového typu (dané jménen rozhraní), bez ohledu na konkrétní implementaci chování
  - Chceme využít vícenásobnou dědičnost

`http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html`

Zadání úlohy – Rámec pro simulaci strategického rozhodování

- Vytvořte simulátor strategické hry (např. sázení–ruleta)
- K simulátoru se může připojit až 5 účastníků hry
- Jeden simulační krok hry lze vyvolat metodou **nextRound**
- Vytvořte tři ukázkové hráče demonstrující použití rámce
  a. Jeden hráč vždy sází na červenou (**PlayerRed**)
  b. Druhý hráč sází náhodně na čísla od 1 do 36 (**PlayerRandom**)
  c. Třetí hráč sází vždy na políčko s nejnižší hodnotou (**PlayerMin**)

Návrh základní struktury

- Rámec se skládá z
  - Herního světa—**World** definující políčka, na která lze vsadit
  - Sázky **Bet** dle pravidel světa
  - Účastníka (**Participant**) hry, který sází na políčka v herním světě
  - Vlastního simulátor—**Simulator**, který obsahuje svět, hrající hráče a zároveň umožňuje připojení hráčů do hry

    Kompozice / Agregace
  - Hráčů (**Player**) hrající strategií a, b nebo c

    Pro demonstraci použití rámce

Sázka

- Sázka – Bet – na co hráč sází a kolik

*Jednou vyřčená sázka platí a je neměnná – **immutable object***

```
public class Bet {
    private final String bet;
    private final int amount;

    public Bet(String bet, int amount) {
        this.bet = bet;
        this.amount = amount;
    }

    public String getBet() { return bet; }
    public int getAmount() { return amount; }

    @Override
    public String toString() {
        return "(" + bet + "," + amount + ")";
    }
}
```

Pro jednoduchost uvažujeme sázku na políčko jako String

Herní svět

- Herní svět World definuje políčka a umožňuje účastníkům (Participant) položit sázku (Bet)

Pro jednoduchost uvažujeme pouze políčka s čísly.

```
public class World {
    private final int MIN_NUMBER = 1;
    private final int MAX_NUMBER = 36;

    public int getMinNumber() {
        return MIN_NUMBER;
    }

    public int getMaxNumber() {
        return MAX_NUMBER;
    }
}
```

- Zapouzdřujeme rozsah číselných políček

Účastník hry – Participant

- Účastník může být implementován v jiných třídách (někým jiným),
- proto volíme pro účastníka rozhraní **interface**
- S referenční proměnnou typu **Participant** můžeme „pracovat" v simulátoru aniž bychom znali konkrétní implementaci
- Účastník má v této chvíli pouze jediné definované chování a to vsadit si (sázku **Bet**) – metoda **doStep** pro konkrétní svět

```
public interface Participant {

    public Bet doStep(World world);

}
```

- Předáváme referenční proměnnou **World**
  - Hráč se tak může informovat o aktuálním stavu světa

Simulační rámec — Simulator

- **Simulátor** obsahuje svět **World** *(agregace)*
- **Simulátor** obsahuje hráče, ale ty jsou vytváření nezávisle mimo simulátor a připojují se ke hře metodou **join** *(agregace)*
- Konkrétní implementace hráče je nezávislá, proto agregujeme účastníka hry **Participant**

```
public class Simulator {

    World world;
    ArrayList participants;
    final int MAX_PLAYERS = 5;
    int round;

    Simulator(World world) {
        this.world = world;
        participants = new ArrayList();
        round = 0;
    }

    public void join(Participant player) { ... }

    public void nextRound() { ... }
}
```

Připojení účastníka hry — Simulator – join

- Účastníky hry uložíme v kontejneru ArrayList
- Kontrolujeme maximální počet účastníků hry
- a přidáváme pouze nenulového hráče a to pouze jednou (indexOf)

```
public void join(Participant player) {
    if (participants.size() >= MAX_PLAYERS) {
        throw new RuntimeException("Too many players in
     the game");
    }
    if (player != null && participants.indexOf(player)
     == -1) {
        participants.add(player);
    }
}
```

Připojení účastníka hry — Simulator – nextRound

- Rámec odehrátí jednoho kola můžeme implementovat i bez známé implementace konkrétního hráče
- Polymorfismus zajistí dynamickou vazbu na konkrétní objekt a volání příslušné metody objektu, který je uložen v seznamu **participants**

```
public void nextRound() {
  for(int i = 0; i < participants.size(); ++i) {
    Participant player = (Participant)participants.get(i);
    Bet bet = player.doStep(world);
    System.out.println("Round " + round + " player #" + i
    + "(" + player + ") bet: " + bet);
  }
  round++;
}
```

ArrayList obsahuje referenční proměnné typu Object, proto musím explicitně přetypovat. Tomu se můžeme vyhnout využitím generických typů, viz 2. přednáška.

Hráč – Abstraktní třída **Player**

- Demo hráči mohou sdílet společný kód, např. pro vypsání svého jména,
- proto volíme abstraktní třídu

```
public abstract class Player implements Participant {
    private final String name;

    public Player(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

*Jedná se o abstraktní třídu, proto nemusíme explicitně uvádět metodu implementující rozhraní **Participant**, která je automaticky abstraktní.*

- Implementace metody **doStep** je „vynucena" v odvozených třídách pro dílčí strategie **RandomPlayer**, **RedPlayer** a **MinPlayer**

Ukázka hráčů – RedPlayer

- RedPlayer

```
public class RedPlayer extends Player {
    public RedPlayer() {
        super("Red");
    }
    @Override
    public Bet doStep(World world) {
        return new Bet("red", 1); //always bet 1
    }
}
```

Ukázka hráčů – RedPlayer a RandomPlayer

- RandomPlayer

```
public class RandomPlayer extends Player {
    Random rand;
    public RandomPlayer() {
        super("Random");
        rand = new Random();
    }
    @Override
    public Bet doStep(World world) {
        Integer bet = rand.nextInt(36)+1;
        return new Bet(bet.toString(), 1); //bet 1
}   }
```

Ukázka hráče – MinPlayer

- MinPlayer

```
public class MinPlayer extends Player {
    public MinPlayer() {
        super("Min");
    }
    @Override
    public Bet doStep(World world) {
        Integer bet = world.getMinNumber();
        return new Bet(bet.toString(), 1); //always bet 1
    }
}
```

V tomto případě hráč interaguje se světem

Ukázka použití

```
public class Demo {
    public static void main(String[] args) {
        Simulator sim = new Simulator(new World());
        sim.join(new RandomPlayer());
        sim.join(new RedPlayer());
        sim.join(new MinPlayer());

        for(int i = 0; i < 3; ++i) {
            System.out.println("Round number: " + i);
            sim.nextRound();
        }
    }
}
```

Simulator

Polymorfismus a dynamická vazba

- Za běhu programu je vyhodnocen konkrétní objekt a podle toho je volána jeho příslušná metoda
- V příkladu je to metoda **doStep** rozhraní **Participant**
- Zvolený návrh nám umožňuje doplňovat další hráče s různými strategiemi aniž bychom museli modifikovat svět nebo simulátor
- Využitím polymorfismu získáváme modulární a relativně dobře rozšiřitelný (použitelný) rámec
- Uvedené technice se také říká **single dispatch**

  Předáváme volání funkce dynamicky (za běhu programu) identifikovanému objektu

Single Dispatch

- Základním principem tohoto návrhového vzoru je dynamická vazba a vyhodnocení typu za běhu programu
- Voláním identické metody **player.doStep()** získáme pokaždé jinou sázku aniž bychom museli identifikovat příslušného hráče

Výhoda dynamické vazby – virtuální funkce

- Relativně komplexního chování jsme dosáhli interakcí více jednoduchých objektů
- Při vykonání kódu je použita dynamická vazba pouze u jednoho objektu
- Je-li volání funkce závislé na více za běhu detekovaných objektech, hovoříme o **multi dispatch**
- V případě dvou objektů se jedná o **double dispatch**

Příklad rozšíření – Přidání políčka s hodnotou nula

- Přidání políčka s hodnotou 0 realizujeme vytvořením nové třídy **WorldZero**, která rozšiřuje původní svět **World**

```
public class WorldZero extends World {
   private final int MIN_NUMBER = 0;

   public int getMinNumber() {
      return MIN_NUMBER;
   }

}
```

- Nový svět stačí předat simulátoru v konstruktoru

```
Simulator sim = new Simulator(new WorldZero());
```

- Zbytek programu zůstává identický

Příklad: Simulator

- Jak definovat nový svět s novými vlastnostmi aniž bychom museli modifikovat kompletně celý program?

Řešení je použit návrhový vzor **double dispatch**

Double Dispatch

- Principem **double dispatch** je vyhodnocení dvou objektů za běhu programu a automatická volba volání odpovídající funkce
- Podobného efektu lze dosáhnout použitím **instanceof** pro detekci příslušného typu objektu a explicitním voláním příslušné třídy
- Vzor double dispatch je však elegatnější a jednodušší

Příklad nového světa s novými vlastnostmi

- Nejdříve musíme zajistit identifikaci objektu světa za běhu
- Do světa proto přidáme metodu, ze které budeme volat **doStep** konkrétního hráče

```
public class World {
    ...
        Bet doStep(Participant player) {
            return player.doStep(this);
        }
}
```

Tak zajistíme identifikaci konkrétní implementace světa

- Metodu pojmenujeme například **doStep**
- Ve třídě **Simulator** upravíme volání `player.doStep(world)` na **world.doStep(player)**
- Tím zajistíme, že se nejdříve dynamicky identifikuje typ objektu referenční proměnné **world** a následně pak typ objektu v referenční proměnné **player**

Program nyní funguje jako předtím, navíc nám však umožňuje rozšít simulátor o novou implementaci světa

Nový svět – WorldNew

```
public class WorldNew extends World {

   private final String[] fields;

   public WorldNew() {
      super();
      fields = new String[36 + 1 + 4];
      fields[0] = "even";
      fields[1] = "odd";
      fields[2] = "red";
      fields[3] = "black";
      for (int i = 0; i <= 36; ++i) {
         fields[i + 4] = Integer.toString(i);
      }
   }
   Bet doStep(Participant player) { // we need to link
      return player.doStep(this);  //  doStep with this
   }
   public String[] getFields() { //new method
      return fields;
   }
}
```

Rozšíření účastníka a existujících hráčů

- Účastníka hry **Participant** musíme rozšířit o uvažování nového světa

```
public interface Participant {
    public Bet doStep(World world);
    public Bet doStep(WorldNew world);
}
```

- Implementaci původních hráčů provedeme v abstraktní třídě **Player**

```
public class RandomPlayer extends Player {
    public void doStep() {
        // do a random strategy
    }
}
```

*Chování původních hráčů v novém světě neřešíme, proto s výhodou modifikujeme pouze abstraktní třídu **Player**.*

Nový hráč pro nový svět – PlayerNew

```
import java.util.Random;

public class PlayerNew extends Player {
  Random rand;
  public PlayerNew() {
    super("New player");
    rand = new Random();
  }
  @Override
  public Bet doStep(World world) {
    // strategy for standard world
    return new Bet("black", 1); //always bet 1 gold
  }
  @Override
  public Bet doStep(WorldNew world) {
    // strategy for the new world
    // random choice even or odd
    return new Bet(world.getFields()[rand.nextInt(2)], 1);
  }
}
```

Nový hráč má jiné chování v původním a novém světě.

Použití nového hráče v novém světě – Demo

```
public class Demo {

   public static void main(String[] args) {
      Simulator sim = new Simulator(new WorldNew());
      sim.join(new RandomPlayer());
      sim.join(new RedPlayer());
      sim.join(new MinPlayer());
      sim.join(new PlayerNew());

      for (int i = 0; i < 3; ++i) {
         System.out.println("Round number: " + i);
         sim.nextRound();
      }
   }
}
```

SimulatorDD

- Pouze rozšíření světa nestačí, je nutné realizovat dynamickou vazbu
- Svět a hráče můžeme nyní rozšiřovat, aniž bychom museli zasahovat do simulačního rámec třídy **Simulator**

Přetížení metod „overloading" a přepsání metod „overiding"

- Přetížení metody je volba konkrétní implementace na základě typu a počtu parametrů.
- Přetížení je statická vazba a děje se při kompilaci programu
- Volání přepsané metody je vazba dynamická a děje se za běhu programu.
- Identifikovat objekt můžeme také sami operátorem **instanceof**
- Double dispatch obsahuje volání funkce navíc, ale ta je velmi krátká a tak je zpravidla „inlinov8na" za běhu

Při načtení programu i za běhu jsou prováděny optimalizace a krátké funkce tak mohou být přímo vloženy do kódu. Odpadá tak režie související s voláním a uložením „program counter" / „instruction pointer"

Shrnutí přednášky

Diskutovaná témata

- Objektově orientvaného programování v Javě
- Struktura objektu a zapouzdření
- Immutable objekty
- Vztahy mezi objekty
- Dědičnost
- Kompozice
- Polymorfismus
- Využití polymorfismu a návrhový vzor Double dispatch

- Příště: výčtové typy a kolekce v Javě, generické typy, iterátor