Informatika 2

Princípy polymorfizmu

doc. Ing. Ján Janech, PhD.
26. 2. 2024

ŽILINSKÁ UNIVERZITA V ŽILINE
Fakulta riadenia
a informatiky

Pojmy zavedené v 1. prednáške (1)

- implementačná závislosť – minimálna
- súdržnosť – maximálna
- duplicita kódu – súdržnosť
- priamy prístup k atribútom – závislosť, zodpovednosť

- refaktoring
  - zmena kódu bez zmeny jeho funkcie
  - úpravy pre zlepšenie miery závislosti, súdržnosti a zodpovednosti

Pojmy zavedené v 1. prednáške (2)

- pokročilé možnosti platformy Java na sprehľadnenie kódu
  - kontajner HashMap
  - Optional
  - var
  - rozšírený switch

- balíčky

Cieľ prednášky

- polymorfizmus
- interface

- príklad: hra Mravenci

Naposledy zabudnuté – viditeľnosť package-private (1)

- minulý semester:

- hlavička triedy

```
public class AutomatMHD
```

- hlavička metódy

```
public int getCenaListka()
```

- hlavička enumu

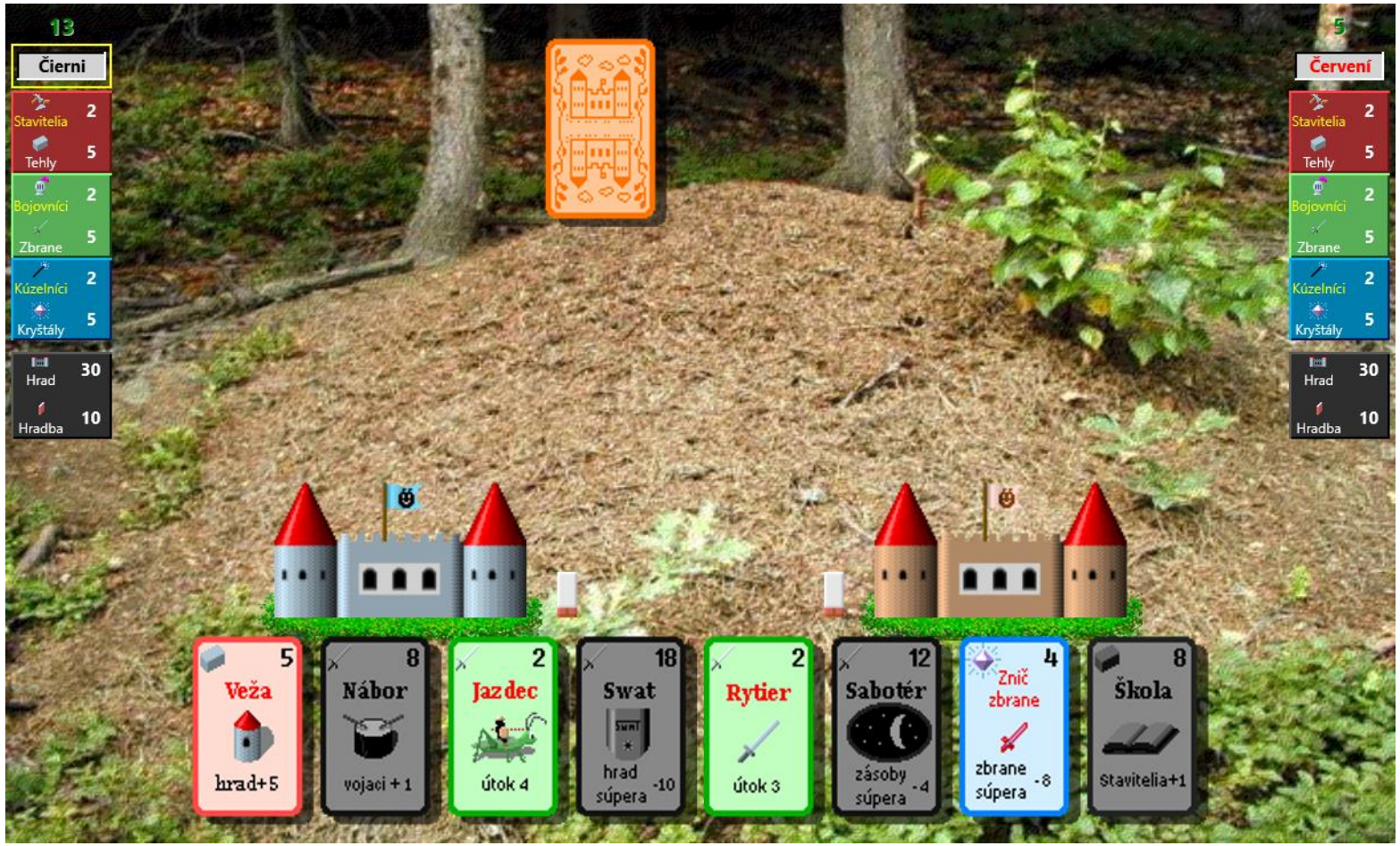
```
public enum StavHry
```

Naposledy zabudnuté – viditeľnosť package-private (2)

- čo ak vynecháme public?
  - skúšali sme minulý semester
  - tvárilo sa všetko rovnako

- prístupové právo package-private
  - neuvádza sa – je predvolené, ak sa neuvedie iné
  - prístup v rámci balíčka
    - v rámci balíčka rovnako ako public
    - mimo balíčka rovnako ako private

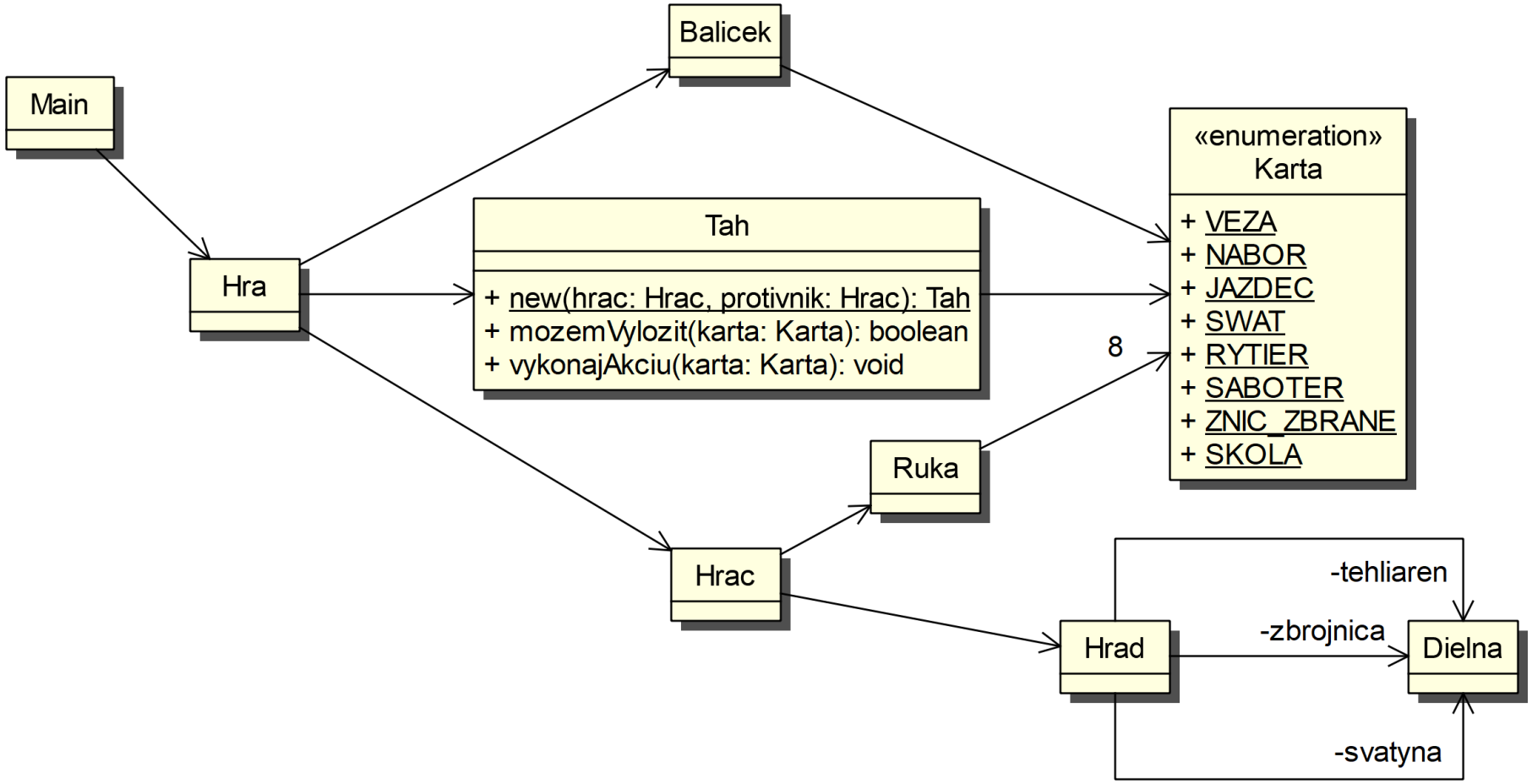
Hra Mravenci (1)



Hra Mravenci (2)

- kartová hra
- hrá sa v ťahoch
- červené aj čierne mravce majú hrad s hradbou
- v každom hrade sú pracovníci (stavitelia, vojaci, čarodejníci)
- pracovníci generujú v na začiatku ťahu suroviny (tehly, zbrane, kryštály)
- karty spotrebovávajú suroviny a vykonávajú akcie
  - útok
  - zväčšenie hradu/hradieb
  - najatie nového pracovníka
  - ...

Implementácia hry



Enum Karta

```
public enum Karta {
    VEZA(5, Surovina.TEHLA, "Veža", "Hrad +5"),
    NABOR(8, Surovina.ZBRAN, "Nábor", "Vojaci +1"),
    JAZDEC(2, Surovina.ZBRAN, "Jazdec", "Útok 4"),
    SWAT(18, Surovina.ZBRAN, "SWAT", "Hrad súpera -10"),
    RYTIER(2, Surovina.ZBRAN, "Rytier", "Útok 3"),
    SABOTER(12, Surovina.ZBRAN, "Sabotér", "Zásoby súpera -4"),
    CARUJ_ZBRANE(4, Surovina.KRYSTAL, "Čaruj zbrane", "Zbrane +8"),
    SKOLA(8, Surovina.TEHLA, "Škola", "Stavitelia +1"),
...
```

Metóda Tah.vykonajAkciu (1)

```
public void vykonajAkciu(Karta karta) {
    Hrad hradHraca = this.hrac.getHrad();
    Hrad hradProtivnika = this.protivnik.getHrad();
    hradHraca.getDielna(karta.getSurovina())
        .zmenSuroviny(-karta.getMnozstvoSurovin());
```

Metóda Tah.vykonajAkciu (2)

```
    switch (karta) {
        case VEZA -> {
            hradHraca.zmenVyskuHradu(+5);
        }
        case NABOR -> {
            hradHraca.getDielna(Surovina.ZBRAN).zmenRobotnikov(+1);
        }
        case JAZDEC -> {
            hradProtivnika.prijmiUtok(4, false);
        }
        case SWAT -> {
            hradProtivnika.prijmiUtok(10, true);
        }
...
```

Problém s implementáciou

- nízka súdržnosť
  - karta je implementovaná vo viac triedach
  - základné informácie o karte -> enum Karta
  - akcia karty -> metóda Tah.vykonajAkciu

- zložité pridávanie nových kariet
- zložité na refaktoring a úpravu vlastností
- veľká pravdepodobnosť chyby
- ťažká čitateľnosť

Rozumnejšie riešenie

- každá karta = samostatná trieda
  - implementuje vlastnosti karty
  - implementuje vykonávanie akcie karty

- viac písania
  - nevadí, píšeme len raz a prostredie pomáha
- veľký počet tried (30 typov kariet = 30 tried)
  - nevadí, už nemáme BlueJ

Príklad implementácie karty – veža

```
public class Veza {
    public int getMnozstvoSurovin() {
        return 5;
    }
    public Surovina getSurovina() {
        return Surovina.TEHLA;
    }
    public String getNazov() {
        return "Veža";
    }
    public String getPopis() {
        return "Hrad +5";
    }
    public void vykonajAkciu(Tah tah) {
        tah.getHradHraca().zmenVyskuHradu(+5);
    }
}
```

Metóda Tah.vykonajAkciu s použitím nových kariet (1)

```
public void vykonajAkciu(Karta karta) {
    Hrad hradHraca = this.hrac.getHrad();
    Hrad hradProtivnika = this.protivnik.getHrad();
```

Metóda Tah.vykonajAkciu s použitím nových kariet (2)

```
    switch (karta) {
        case VEZA -> {
            Veza karta = new Veza();
            hradHraca.getDielna(karta.getSurovina())
                .zmenSuroviny(-karta.getMnozstvoSurovin());
            karta.vykonajAkciu(this);
        }
        case NABOR -> {
            Nabor karta = new Nabor();
            hradHraca.getDielna(karta.getSurovina())
                .zmenSuroviny(-karta.getMnozstvoSurovin());
            karta.vykonajAkciu(this);
        }
...
```

Problém s implementáciou

- aj tak treba switch
- dopracovali sme sa od zlého k horšiemu riešeniu
- alebo?

Čo ak by šlo spraviť

```
public void vykonajAkciu(??? karta) {
    Hrad hradHraca = this.hrac.getHrad();
    Hrad hradProtivnika = this.protivnik.getHrad();
    hradHraca.getDielna(karta.getSurovina())
        .zmenSuroviny(-karta.getMnozstvoSurovin());
    karta.vykonajAkciu(this);
}
```

Polymorfizmus (1)

- riešenie možné vďaka polymorfizmu

- polymorfizmus:
  využitie situácie, keď sú dva rôzne objekty schopné prijať tú istú správu a pritom každý z nich môže reagovať inak – použije inú metódu

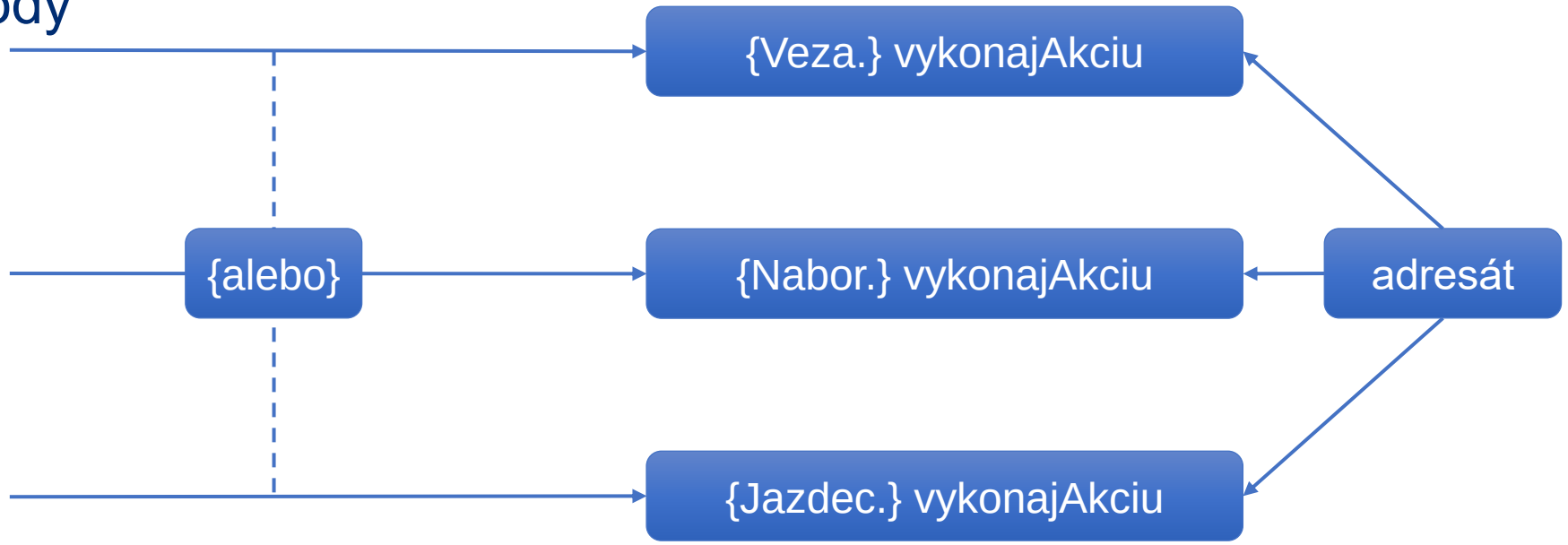
Polymorfizmus (2)

- odosielateľ správy sa nestará o typ adresáta
  - zjednodušenie na strane odosielateľa
- postačuje, že adresát je schopný správu prijať
  - má správu v rozhraní
- každý adresát použije svoju metódu

Protokol: správa → metóda

- jedna správa

- rôzne metódy



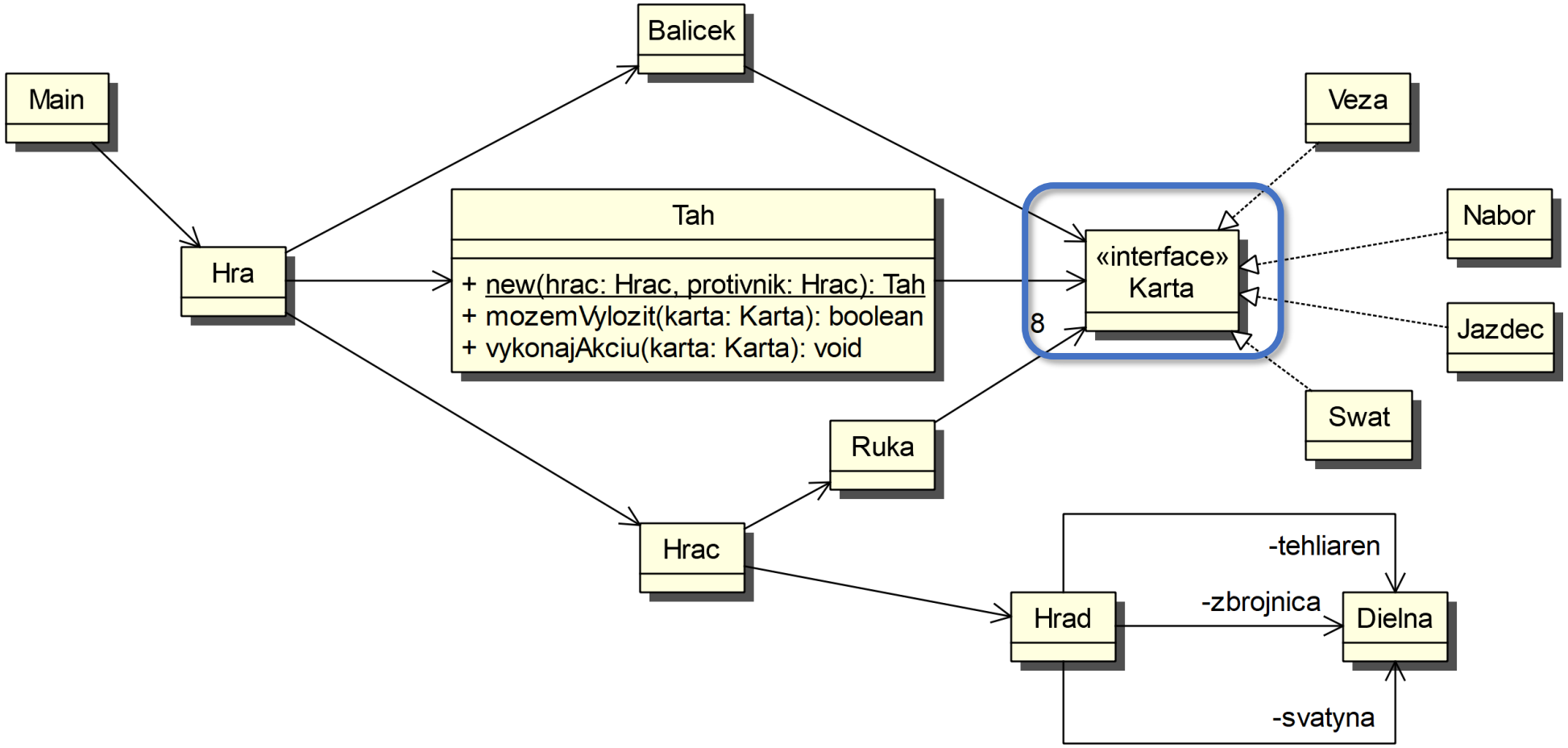
Prostriedky jazyka Java

- správa v rozhraní – podmienka pre jej poslanie

- interface
  - explicitná definícia zoznamu správ v rozhraní
  - definuje typ objektových premenných

- trieda implementuje interface
  - zaručuje, že má implementované všetky metódy pre správy uvedené v interface
  - inštancie sú schopné chovať sa polymorfne

Aplikácia interface do hry



Interface v jazyku Java

- kľúčové slovo interface
- správy bez modifikátora prístupu ukončené ;
- vždy public

Interface Karta

```
public interface Karta {
    int getMnozstvoSurovin();
    Surovina getSurovina();
    String getNazov();
    String getPopis();
    void vykonajAkciu(Tah tah);
}
```

Implementácia interface

- explicitné uvedenie interface v hlavičke triedy
- ľubovoľný počet implementovaných interface

- metódy pre všetky správy z interface
  - musia byť verejné
  - správy sa musia presne zhodovať

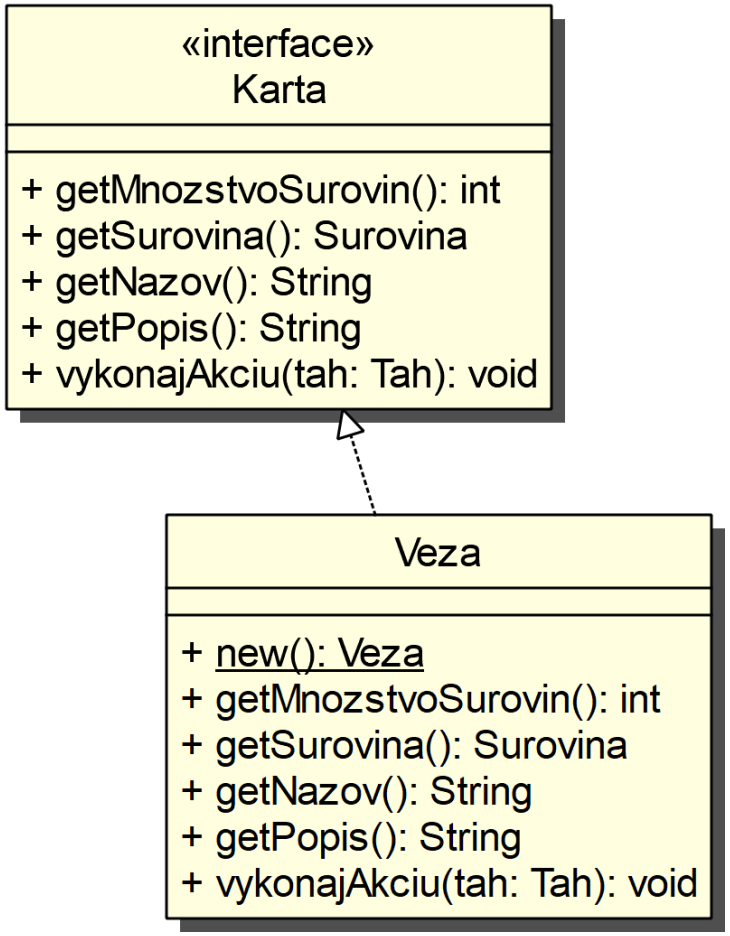
Java – implementácia rozhrania

```
public class Veza implements Karta {
    public int getMnozstvoSurovin() {
        return 5;
    }
    public Surovina getSurovina() {
        return Surovina.TEHLA;
    }
    public String getNazov() {
        return "Veža";
    }
    public String getPopis() {
        return "Hrad +5";
    }
    public void vykonajAkciu(Tah tah) {
        tah.getHradHraca().zmenVyskuHradu(+5);
    }
}
```

Interface v UML (1)

- obdĺžnik – ako trieda
- stereotyp «interface» nad menom
- len správy inštancii

Interface v UML (2)



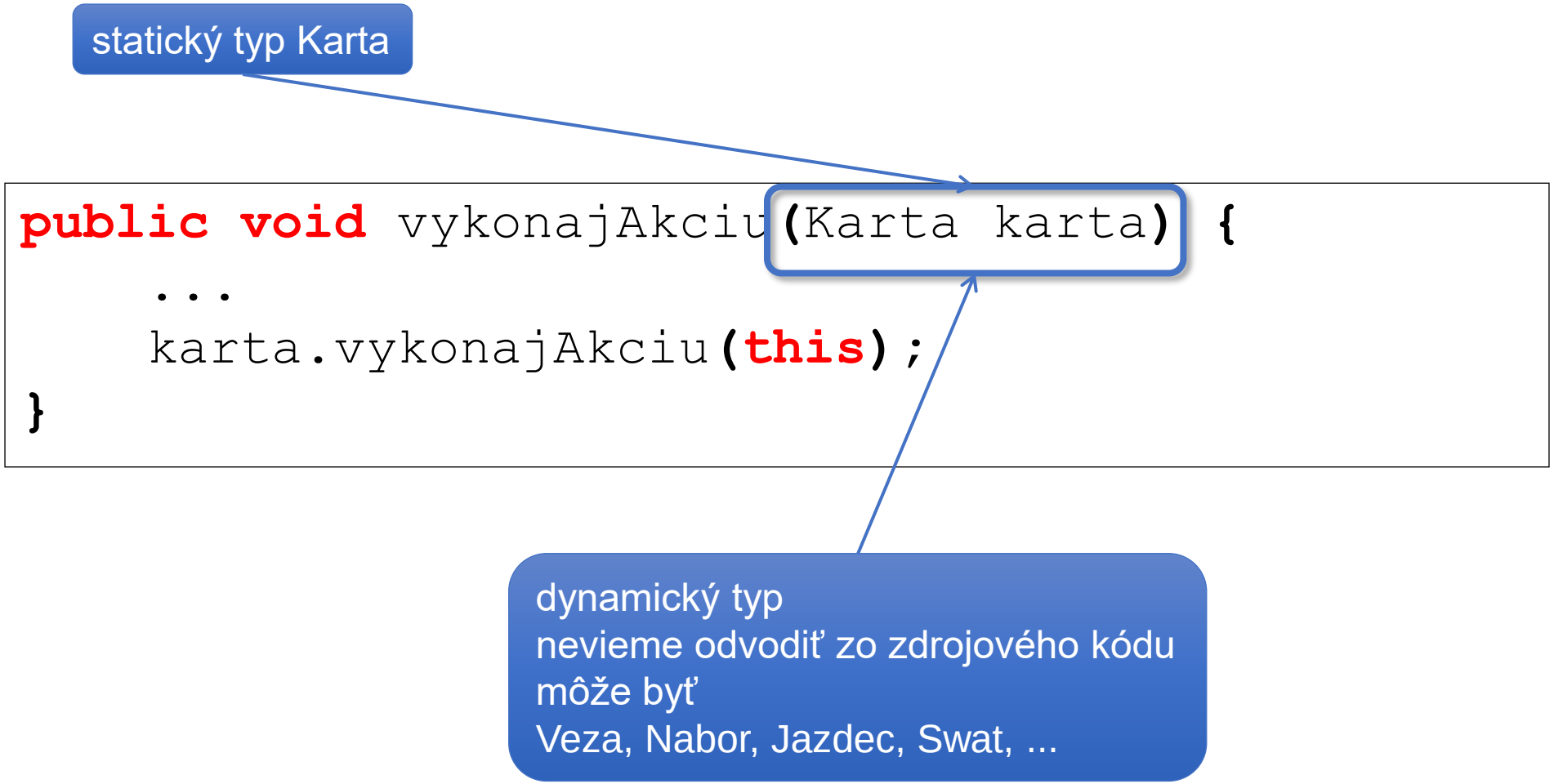
Poslanie správy cez premennú typu interface

```
public void vykonajAkciu(Karta karta) {
    Hrad hradHraca = this.hrac.getHrad();
    Hrad hradProtivnika = this.protivnik.getHrad();
    hradHraca.getDielna(karta.getSurovina())
        .zmenSuroviny(-karta.getMnozstvoSurovin());
    karta.vykonajAkciu(this);
}
```

Statický kontra dynamický typ (1)

- statický typ
  - uvedený v definícii objektovej premennej
  - určuje sa pri preklade
  - určuje množinu správ, ktoré je možno poslať (iné nie)
- dynamický typ
  - skutočný typ objektu referencovaného premennou
  - určuje sa za behu pri priradení hodnoty premennej
  - určuje chovanie objektu, reakcie na správy

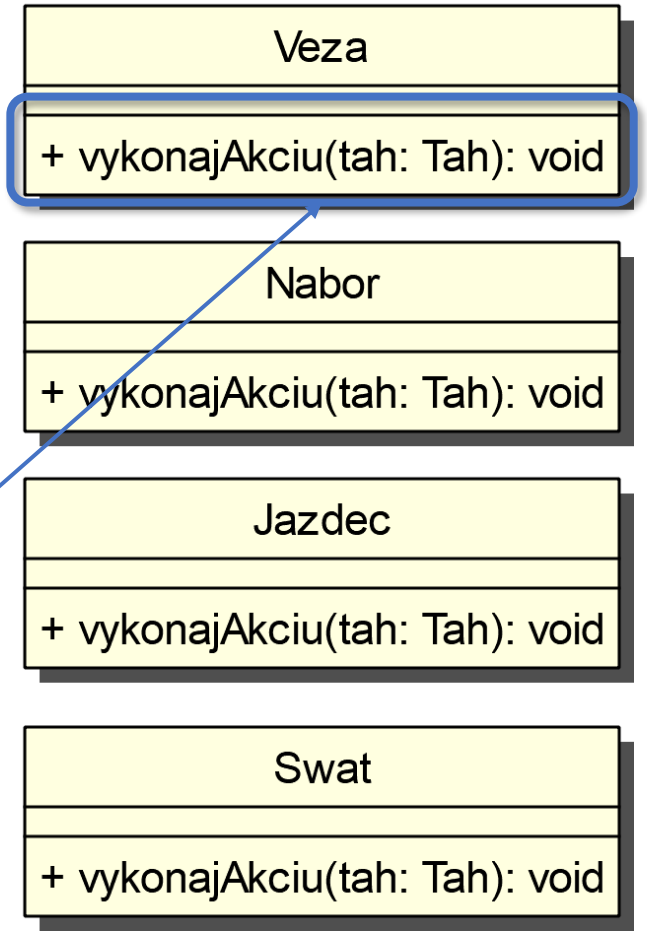
Statický kontra dynamický typ (2)



Spracovanie správy pri polymorfizme (1)

```
tah.vykonajAkciu(new Veza());
```

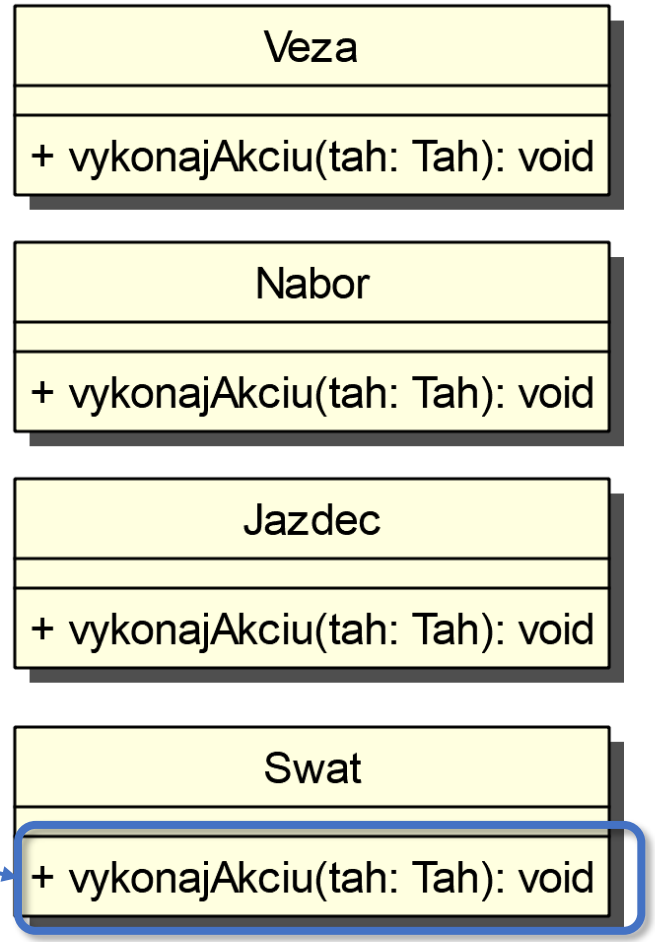
```
public void
vykonajAkciu(Karta karta) {
    ...
    karta.vykonajAkciu(this);
}
```



Spracovanie správy pri polymorfizme (2)

```
tah.vykonajAkciu(new Swat());
```

```
public void
vykonajAkciu(Karta karta) {
    ...
    karta.vykonajAkciu(this);
}
```



Spracovanie správy pri polymorfizme (3)

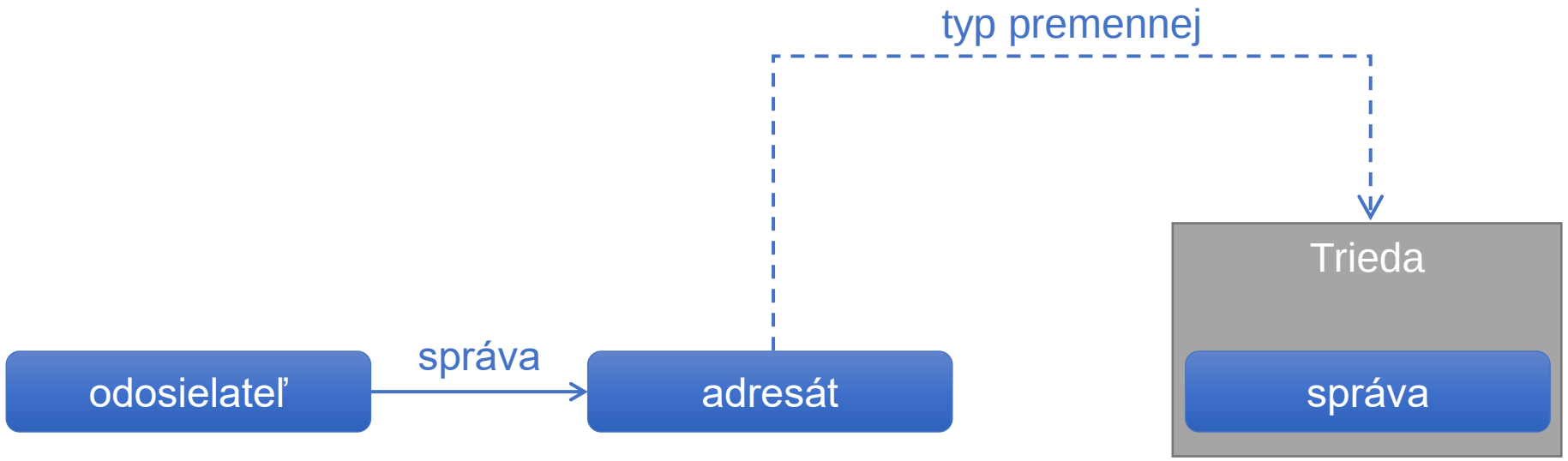
- príkaz bol rovnaký
- statický typ je rovnaký

- vykonávajú sa rôzne metódy

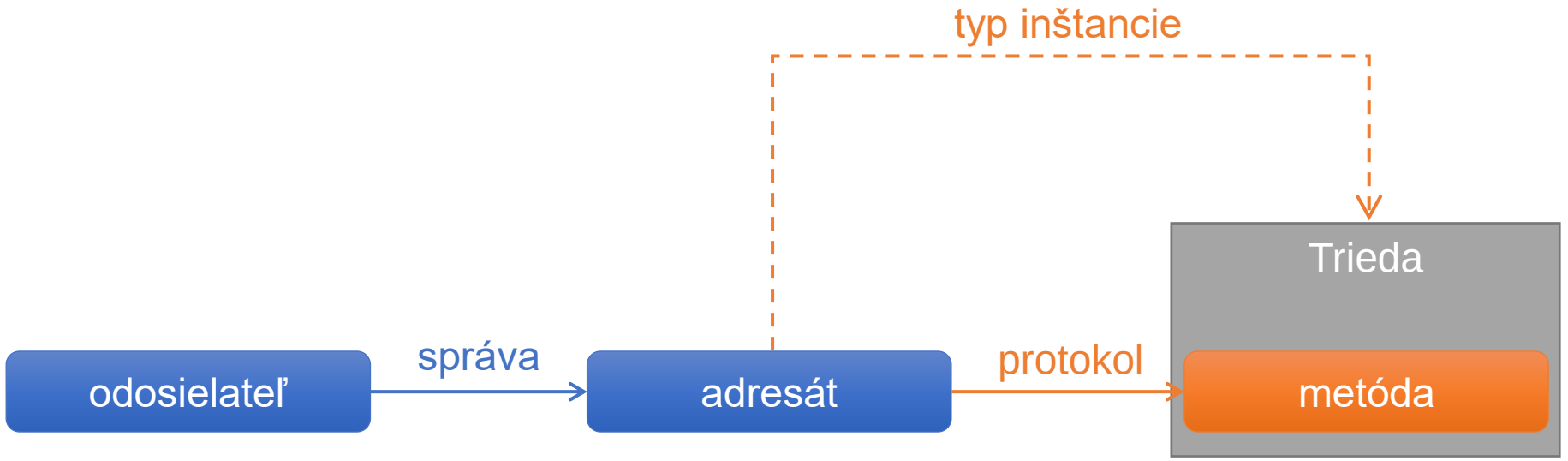
Rozhranie – vonkajší pohľad

Implementácia – vnútorný pohľad

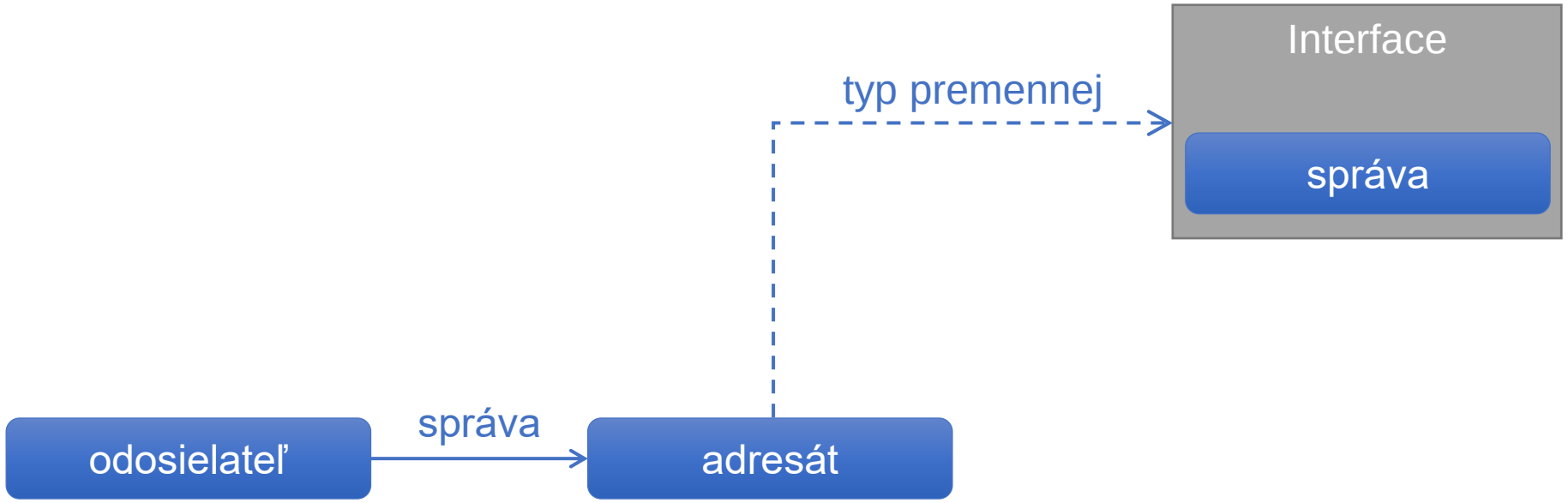
Jazyk – vonkajší pohľad



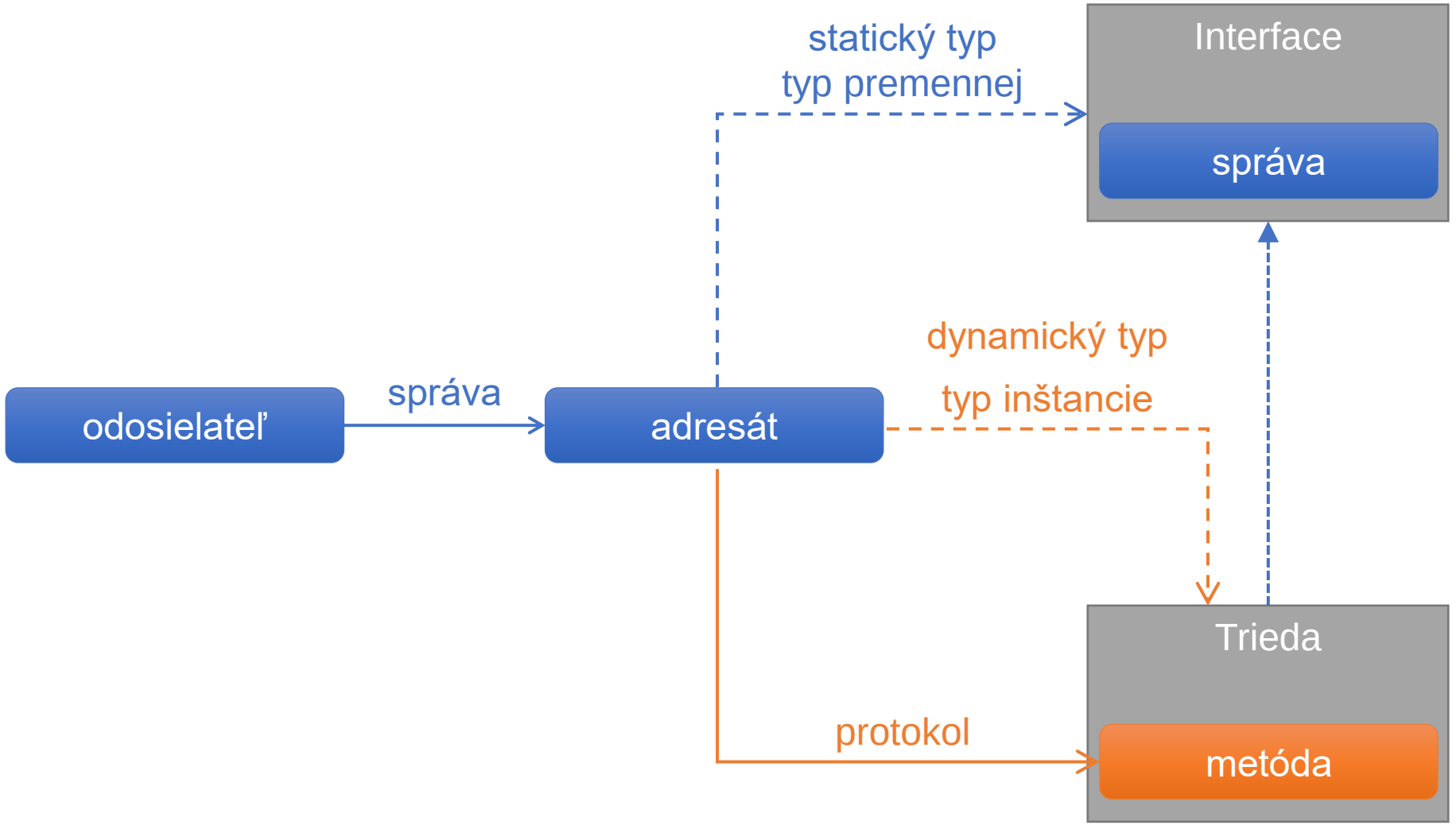
Jazyk – vnútorný pohľad



Polymorfizmus – vonkajší pohľad



Polymorfizmus – vonkajší pohľad



Typová kompatibilita a interface

- inštancia je typovo kompatibilná s interface ak jej trieda implementuje daný interface

- hodnota null je typovo kompatibilná so všetkými interface

Metóda zoberKartu v triede Balik

```
public Karta zoberKartu() {
    switch (this.nahodneCisla.nextInt(30)) {
        case 0:
            return new Veza();
        case 1:
            return new Nabor();
        case 2:
            return new Jazdec();
        case 3:
            return new Swat();
...
    }
}
```

```
public class Veza implements Karta
```

Polymorfizmus – pozor

- krásny princíp
  - veľké dosahy pre prax

- dať si pozor na
  - veľké množstvo tried
  - Liskovej substitučný princíp

Barbara Liskov (*1939)

- 1987 – princíp substitúcie pre typy
- 2004 – medaila Jon von Neumana
- 2008 – Turingova cena

The Liskov Substitution Principle (LSP)

- *Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T.*

- Nech q(x) je preukázateľná vlastnosť objektu x typu T. Potom q(y) by mala platiť pre objekt y typu S, kde S je podtyp typu T.

Liskovej princíp substitúcie – slovensky

- ak existuje vlastnosť interface, ktorú implementácia nespĺňa = porušený substitučný princíp.

- vlastnosti – uvedené v dokumentácii

Drobná pripomienka - @Override

- označenie – metóda implementujúca interface

```
@Override
public void vykonajAkciu(Tah tah) {
```

- nepovinné
- len pripomienka pre prekladač a dokumentácia
- odporúčam písať
  - IntelliJ IDEA automaticky