Programko.sk

Java 3

Vzorové riešenia úloh: https://github.com/programko-kurzy/kurz-java-3

Verzia: 3.5.2021

1. Hodina | Opakovanie – polia a ArrayList

Pripomeňte si prácu s poliami a ArrayListami a rozdiely medzi nimi riešením nasledujúcich príkladov. V zadaní nasledujúcich príkladov sa spomína pole. Môžete si vybrať, či daný príklad budete riešiť pomocou poľa alebo pomocou ArrayListu.

Príklady

1.1 Príklad

Vygenerujte náhodné celé číslo x (medzi 1 a 20).
Následne pomocou for-cyklu vytvorte pole, ktoré bude obsahovať x náhodných čísel (každé číslo medzi 1 a 100).

Hint: na vygenerovanie náhodného čísla použite **Random**.

1.2 Príklad

Vypíšte dĺžku poľa, ktoré ste vytvorili v predošlom príklade. Následne vypíšte súčet všetkých prvkov vo vašom poli. Na záver vygenerujte náhodný index z vami vytvoreného poľa a odstráňte prvok na danom indexe. Zamyslite sa aké indexy môžu byť v danom poli a podľa toho nastavte hranice pri generovaní náhodného čísla.

1.3 Príklad

Zistite hodnoty maximálneho a minimálneho prvku vo vašom poli a odčítajte ich od seba (maximum - minimum). Ak je výsledná hodnota väčšia, nanajvýš rovná ako dĺžka poľa, pridajte túto hodnotu do poľa. V opačnom prípade odstráňte z poľa prvok s indexom rovnajúcim sa vypočítanej hodnote.

1.4 Príklad

Vytvorte pole s prvkami [1, 3, 3, 41, 5, 6, 10, 13, 27, 0, -7, 9].
Vypíšte z poľa indexy, na ktorých sa nachádzajú maximálny a minimálny prvok.

1.5 Príklad

Vytvorte dve ľubovoľné polia a potom tieto dve polia spojte do jedného poľa.
Ak je dĺžka výsledného poľa väčšia ako 7 vymažte z poľa prvé 3 prvky.
V opačnom prípade vymažte z poľa prvú polovicu prvkov. Vypíšte výsledné pole.

1.6 Príklad

Vytvorte nasledujúce pole: [1, 3, 3, 41, 5, 6, 10, 13, 27, 0, -7, 9].
Vypíšte dané pole pomocou while-cyklu odpredu a pomocou for-cyklu odzadu.

1.7 Príklad

Z poľa z predošlého príkladu odstráňte n prvkov. Číslo n vygenerujte ako náhodné číslo. Uvedomte si aké hranice je potrebné vložiť do funkcie generujúcej n.

1.8 Príklad

Majme nasledujúce pole: [1, 3, 5, 6, 10, 27, 0, 9].
Postupne doplňte do poľa chýbajúce prvky tak, aby po doplnení pole vyzeralo nasledovne: [1, 3, 3, 41, 5, 6, 10, 13, 27, 0, -7, 9]

2. Hodina | Opakovanie – HashMap. Hodnota null

HashMap (mapa, slovník) je asociatívny zoznam, ktorý mapuje kľúče na hodnoty. Je podobný ArrayListu, ale jeho prvky nie sú indexované pomocou postupnosti celých čísel, ale pomocou kľúčov. Kľúčom môže byť napríklad číslo, reťazec alebo iný nemeniteľný objekt. Pripomeňte si prácu s HashMapou riešením nasledujúcich úloh.

Príklady

2.1 Príklad

Vytvorte nasledujúcu mapu s názvom points: {“Adam“ : 34, “Paul“ : 20, “Anna“ : 45}.
Do tejto mapy pridajte ďalšieho študenta s ľubovoľným počtom bodov.
Následne mapu vypíšte. Potom pomocou for-cyklu vypíšte všetky kľúče a pomocou ďalšieho for-cyklu vypíšte všetky hodnoty z mapy.

2.2 Príklad

Vytvorte program, ktorý od používateľa postupne načíta 10 slov. Vytvorte mapu, ktorá bude obsahovať štatistiku všetkých písmen, ktoré sa v zadaných slovách nachádzajú.

Ako príklad si uveďme dve slová “ahoj“ a “jano“. Štatistika z týchto dvoch slov je uložená v mape statistics = {“a“ : 2, “h“ : 1, “o“ : 2, “j“ : 2, “n“ : 1}, pretože presne toľko daných písmen sa v týchto dvoch slovách dokopy nachádza.

2.3 Príklad

Vytvorte program, ktorý bude pracovať s mapou points:
{“Adam“: 28, “Mary“: 30, “John“: 21, “Paul“: 15, “Anna“: 24, “Tina“: 18, “David“: 12, “Max : 25}
Táto mapa obsahuje mená študentov a počty bodov z kurzu informatiky. Vašou úlohou bude vytvoriť novú mapu s názvom grades, do ktorej sa uložia známky A, B, C, D, E, F vyrátané pre všetkých študentov. Stupnica pre jednotlivé známky je:
A: 30 – 26 | B: 25 – 21 | C: 20 – 16 | D: 15 – 11 | E: 10 – 6 | F: 5 – 0.

2.4 Príklad

Vytvorte program, ktorý bude pracovať s mapou points z predošlého príkladu.
Vypočítajte koľko bodov dostali študenti kurzu informatiky dohromady.

Hodnota null

Hodnota **null** úzko súvisí s pojmom premenná. Ako viete, každá premenná má typ, názov a hodnotu. V základe existujú 2 kategórie typov premenných: **primitívne a referenčné**.

Primitívne premenné môžeme nazvať aj jednoduché, pretože okrem uloženia hodnoty nám neposkytujú ďalšie funkcionality. Sú nimi napríklad **boolean, int, double, float**.

Referenčné premenné využívajú referenciu ako odkaz na nejaký objekt. Sú to napríklad **String, Integer, Double, Float, Boolean, Scanner, Random** a množstvo ďalších. Oproti primitívnym premenným majú referenčné premenné niekoľko odlišností:

- Poskytujú možnosť využiť rôzne funkcionality priamo prostredníctvom danej premennej, kde rôzne metódy voláme priamo použitím bodky za premennou. Napríklad, pre reťazec je možné zistiť jeho dĺžku (alebo zavolať akúkoľvek inú metódu na prácu či úpravu reťazca) nasledovne:

```
String retazec = “programko“;
System.out.println(retazec.length());
```

- Ak vytvorenej referenčnej premennej nezadáme hodnotu, tak jej hodnota bude **null**. Null je možné inak nazvať aj prázdna hodnota. Ak má premenná túto prázdnu hodnotu a zavoláme na nej nejakú jej metódu (napríklad zistenie dĺžky reťazca), tak program vyhodí chybu **NullPointerException**. Ak je v programe potrebné poznamenať si, že premenná nemá žiadnu hodnotu, tak je možné priradiť hodnotu null priamo do premennej.

```
String retazec1 = "programko";
System.out.println(retazec1.length());    // vypise 9

String retazec2 = "";
System.out.println(retazec2.length());    // vypise 0

String retazec3 = metodaKtoraVratiPrazdnuHodnotu();
System.out.println(retazec3);    // vypise null

String retazec4 = null;
System.out.println(retazec4.length());    // chyba NullPointerException
```

To, či má premenná hodnotu null, je možné zistiť pomocou podmienky IF:

- if (retazec == null) {/* premenná retazec má prázdnu hodnotu */}
- if (retazec != null) {/* premenná retazec má nejakú hodnotu */}.

2.5 Príklad

Vyskúšajte si prácu s hodnotou **null**. Vytvorte premenné, ktorým priradíte prázdnu hodnotu a skúste si nasimulovať chybu **NullPointerException**.

Práca so súbormi je dôležitou časťou mnohých aplikácií. Java ponúka niekoľko možností ako pracovať so súbormi a ako ich vytvárať, čítať, zapisovať do nich a mazať ich. Jedna zo základných tried na prácu so súbormi je trieda **File** (z balíka **java.io.File**). Táto trieda obsahuje množstvo užitočných metód na získavanie informácií o súboroch, ako napríklad:

- **canRead()** – kontrola, či sa zo súboru dá čítať
- **canWrite()** – kontrola, či sa do súboru dá zapisovať
- **createNewFile()** – vytvorenie súboru
- **delete()** – zmazanie súboru
- **exists()** – kontrola, či súbor existuje
- **getName()** – metóda vráti názov súboru
- **getAbsolutePath()** – metóda vráti presnú cestu k súboru
- **length()** – metóda vráti veľkosť súboru v bajtoch
- **isDirectory()** – kontrola, či položka je priečinok
- **isFile()** – kontrola, či položka je súbor
- **mkdir()** – vytvorenie priečinku

Vyskúšajte si vytvorenie a zapisovanie do súboru

```java
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

class FilesExample {

    Run | Debug
    public static void main(String[] args) throws IOException {
        File file = new File("subor.txt");

        if (file.createNewFile()) {
            System.out.println("Subor s menom " + file.getName() + " bol vytvoreny.");
        } else {
            System.out.println("Subor uz existuje.");
        }

        FileWriter writer = new FileWriter(file.getName());
        writer.append("Tento riadok zapiseme do suboru.\n");
        writer.append("A aj tento riadok zapiseme do suboru.\n");
        writer.close();
    }

}
```

Vzorový kód vyššie ukazuje, ako je možné jednoducho vytvoriť nový súbor a zapísať doňho nejaký text. V prvok kroku volaním **new File("subor.txt")** vytvoríme niečo ako odkaz na súbor, s ktorým chceme pracovať. Tento súbor ešte reálne neexistuje. Vytvorí sa až volaním metódy **createNewFile()**, pričom návratová hodnota tejto metódy **vracia boolean**. Ten nám hovorí, či daný súbor bol vytvorený, keďže už mohol reálne existovať.

Následne vytvoríme **FileWriter**, do ktorého pri vytváraní vložíme názov súboru, do ktorého budeme chcieť zapisovať. Potom pomocou metódy **append()** môžeme na koniec textového súboru zapisovať ľubovoľný reťazec. Všimnite si všetky **importy**, ktoré je potrebné pridať, aby program fungoval správne. Zároveň nám pri **main** metóde pribudla klauzula **throws IOException**. Je to jedna z možností ako zabezpečiť spracovanie chýb, ktoré pri práci so súbormi môžu nastať.

Vyskúšajte si čítanie zo súboru a získavanie rôznych informácií o ňom

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class FilesExample2 {
    Run | Debug
    public static void main(String[] args) throws FileNotFoundException {
        File file = new File("subor.txt");
        Scanner s = new Scanner(file);

        if (file.exists()) {
            System.out.println("Nazov suboru: " + file.getName());
            System.out.println("Cesta k suboru: " + file.getAbsolutePath());
            System.out.println("Velkost suboru: " + file.length());

            System.out.println("Obsah suboru:");
            while (s.hasNextLine()) {
                System.out.println(s.nextLine());
            }
        } else {
            System.out.println("Subor neexistuje.");
        }

        s.close();
    }
}
```

Druhý vzorový kód ukazuje príklad ako čítať obsah textového súboru. Všimnite si, že na čítanie zo súboru sme použili **Scanner**, s ktorým pracujeme, keď načítavame vstup od používateľa. Rozdielnom je, že teraz pri jeho vytvorení **nepoužijeme System.in** ale odkaz na súbor, z ktorého chceme čítať obsah. Následne vieme pomocou metód **hasNextLine()** a **nextLine()** načítať celý obsah súboru. Všimnite si tiež klauzulu **throws FileNotFoundException**, ktorú sme museli pridať k **main** metóde kvôli spracovaniu chýb. Nezabudnite tiež na správne **importy**.

Príklady

3.1 Príklad

Vytvorte program, ktorý vytvorí súbor a zapíše doňho nejaký reťazec.

3.2 Príklad

Vytvorte program, ktorý zistí, či súbor z prikladu 3.1 existuje.
Ak áno, vypíšte o ňom nejaké vlastnosti (napr. absolútnu cestu, veľkosť)
a taktiež načítajte a vypíšte jeho obsah.

3.3 Príklad

Vytvorte program, ktorý zmaže súbor z prikladu 3.1.
Následne v programe zistite, či bol súbor skutočne vymazaný.

3.4 Príklad

Vytvorte program, ktorý od používateľa načíta názov súboru a vytvorte súbor s takýmto názvom. Následne od používateľa načítajte 5 riadkov a uložte ich do súboru tak, aby všetky písmená boli nahradené veľkými písmenami (napr. pre vstup „Programko“ do súboru uložte „PROGRAMKO“).
Na konci programu súbor vypíšte a zmažte.

3.5 Príklad

Vytvorte program, ktorý vytvorí jeden priečinok a jeden súbor.
Následne v programe pomocou ich názvov zistite čo je priečinok a čo je súbor.

3.6 Príklad

Vytvorte program, ktorý vytvorí priečinok. Následne do vytvoreného priečinka vytvorte 2 súbory.
Do každého súboru zapíšte ľubovoľný reťazec.
Na konci programu vypíšte obsah priečinka (zoznam súborov v ňom).

3.7 Príklad

Vytvorte program, ktorý vytvorí kópiu vami zvoleného vstupného súboru,
t.j. skopíruje celý jeho obsah do nového súboru.

3.8 Príklad

Vytvorte program, ktorý vytvorí 5 rôznych súborov a naplní ich náhodne dlhými reťazcami.
Následne nájdite najväčší súbor a zmažte ho.

3.9 Príklad

Vytvorte program kalkulačka, ktorý od používateľa načíta 2 čísla a operáciu, ktorú má s nimi urobiť.
Následne celý príklad aj s výsledkom zapíšte do súboru. Výstup v súbore môže vyzerať napr. takto:
„4 + 2 = 6“.

3.10 Príklad

Vytvorte program, ktorý vypočíta faktoriál čísla, ktoré načítate od používateľa.
Výsledok zapíšte do súboru.

3.11 Príklad

Vytvorte metódu **stvorec(int pocet, String znak, String nazovSuboru)**,
ktorá do súboru s názvom **nazovSuboru** nakreslí štvorec o veľkosti **pocet** zložený zo znakov **znak**,
napr. pre vstup **4** a **„P“** bude výstup v súbore takýto:

PPPP
PPPP
PPPP
PPPP

Objektovo orientované programovanie (OOP) patrí medzi najrozšírenejšie štýly programovania. Programovací jazyk Java je jedným z najpopulárnejších OOP jazykov. Podstatu OOP tvoria **triedy** - **class** a **objekty**. Kód je tak organizovaný pomocou tried a objektov. Medzi hlavné výhody OOP patrí prehľadne organizovaný, krátky a efektívny kód, ktorý je ľahko udržiavateľný a flexibilne použiteľný na viacero účelov.

Iste ste si všimli, že aj v programoch, ktoré sme doteraz vytvárali, sme vždy na začiatku definovali **triedu** - **class**. Táto trieda mala metódu **main**, ktorá spúšťala náš program. Odteraz budeme vytvárať aj ďalšie triedy, ktoré môžu byť definované vo viacerých súboroch.

Trieda a inštancia

Ide o základné dva pojmy OOP. Ako príklad z bežného života si pod **triedou** môžeme predstaviť kategóriu, napr. auto a **inštanciami** by mohli byť konkrétne značky áut, ako napr. BMW, VW, Audi, Seat, Renault a pod.

Triedy a inštancie v Jave

```java
class Auto {
    String model;
    int cena;

    void popis() {
        System.out.println("Som " + model + " a moja cena je " + cena + " eur.");
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Auto audi = new Auto();
        audi.cena = 40000;
        audi.model = "Audi A3";
        audi.popis();

        Auto vw = new Auto();
        vw.cena = 25000;
        vw.model = "VW Passat";
        vw.popis();
    }
}
```

V uvedenom príklade sme najskôr definovali **triedu** Auto. Potom sme príkazom **new Auto()** v hlavnom programe vytvorili dve **inštancie** triedy Auto a priradili sme ich do premenných typu Auto s názvami **audi** a **vw**. Nasledovalo nastavenie atribútov pre každú inštanciu a zavolanie metódy popis.

Atribúty a metódy

Trieda Auto má **atribúty** *model* a *cena*. Atribúty sú v podstate premenné, ktoré sú definované pre triedu. Každá inštancia (*audi*, *vw*) má vlastné hodnoty atribútov. Napríklad *cena* audi je 40000 a *cena* vw je 25000. Pre každú inštanciu sa líši aj atribút *model*. Tak ako pri bežných premenných, aj pri deklarácii atribútu uvádzame typ, napr. *String* či *int*.

Okrem atribútov má trieda Auto definovanú aj **metódu** *popis*. Metódy vykonávajú rôzne operácie s atribútmi triedy. V našom prípade ide o vypísanie textu s informáciou o modeli a cene. Rovnako ako pri iných metódach, tiež musíme definovať jej návratový typ.

Trieda by mala vystihovať svoj účel. Ak by sa účel príliš rozširoval pridávaním atribútov a metód, je lepšie triedu rozdeliť do viacerých. Trieda si vďaka **atribútom** uchováva dáta, **metódy** vyjadrujú jej funkcionalitu. Triedy prostredníctvom metód a atribútov vytvárajú dôležitý princíp OOP, vďaka ktorému máme úzko spojené dáta a funkcionalitu. V spojení so správnym nastavením prístupu sa tento princíp nazýva **zapuzdrenie**.

Konštruktor

Aby sa dali inštancie jednoduchšie vytvárať, existuje špeciálna metóda - **konštruktor**. Jej účel je vytvoriť inštanciu a nastaviť jej atribúty. Na rozdiel od štandardných metód triedy, **konštruktor nemá návratový typ**.

Vyskúšajte si vytvoriť inštancie pomocou konštruktora

```java
class Auto {
    String model;
    int cena;

    //konštruktor
    Auto(String nazov, int hodnota) {
        model = nazov;
        cena = hodnota;
    }

    void popis() {
        System.out.println("Som " + model + " a moja cena je " + cena);
    }
}

public class Main {
    Run | Debug
    public static void main(String[] args) {
        Auto audi = new Auto("Audi A3", 40000);
        audi.popis();

        Auto vw = new Auto("VW Passat", 25000);
        vw.popis();
    }
}
```

Príklady

4.1 Príklad

Vytvorte nový adresár, ktorý bude obsahovať triedu s main metódou v jednom súbore a v druhom súbore bude trieda *Auto* z uvedeného príkladu vyššie. V main metóde vytvorte niekoľko inštancií triedy *Auto* a zavolajte na nich metódu *popis*.

4.2 Príklad

Vytvorte v novom súbore triedu *Kalkulacka*, ktorá bude mať celočíselné atribúty *x* a *y* a konštruktor, ktorý ich nastaví. Vytvorte metódy *plus, minus, krat*, ktoré budú vracať výsledok danej operácie na atribútoch triedy. V metóde main vytvorte inštanciu kalkulačky s ľubovoľnými číslami a zavolajte jej metódy.

4.3 Príklad

Inštancie áut vytvorené v príklade 4.1 vložte do poľa alebo ArrayListu s názvom *mojeAuta*. Ako typ poľa / ArrayListu nastavte *Auto*. V cykle prejdite cez všetky prvky poľa / ArrayListu a zavolajte metódu *popis* na každej inštancii.

4.4 Príklad

Upravte predchádzajúci príklad tak, aby sa najskôr vypísala len cena auta. Následne využite cyklus na vypočítanie súčtu cien všetkých áut v poli / ArrayListe.

4.5 Príklad

V novom súbore toho istého adresára vytvorte triedu *Kocka*. Definujte jej atribút *max*, ktorý bude celé číslo. Pridajte konštruktor, ktorý nastaví atribút *max*. Vytvorte metódu *hod*, ktorá vygeneruje náhodné číslo v rozsahu <1, *max*> a následne ho vypíše.

4.6 Príklad

Pridajte do triedy *Kocka* atribút *hody* typu *ArrayList<Integer>*. V konštruktore ho nastavte na prázdny zoznam. Metódu *hod* zmeňte tak, aby už nevypisovala hodené číslo, ale aby ho pridala do atribútu *hody*. Pridajte metódu *kolkokratPadla* s parametrom *cislo* typu *int* a návratový typ metódy nech je *int*. Metóda bude mať za úlohu prejsť zoznam a vrátiť pomocou príkazu *return* koľkokrát padlo číslo, ktoré dostala v parametri *cislo*.

4.7 Príklad

V main metóde vytvorte kocku s maximálnym číslom 6. Simulujte 5 hodení kockou. Vypíšte koľkokrát padla 6-ka ako aj históriu všetkých hodov. Potom znovu hoďte 5 krát tou istou kockou a znovu vypíšte tie isté údaje.

4.8 Príklad

Využijeme triedu *Kocka* z predchádzajúceho príkladu. V main metóde vytvorte kocku s ľubovoľnou maximálnou hodnotou a hoďte ňou 1000-krát. Koľkokrát vám padla na kocke 1-ka?

Modifikátory prístupu

Modifikátory prístupu určujú, či atribúty, metódy alebo trieda samotná je „viditeľná“ pre ostatné triedy. Ak sú viditeľné, môžeme k nim pristupovať z ostatných tried, čiže napríklad môžeme volať metódy alebo nastavovať atribúty danej triedy z ostatných tried.

public - S týmto atribútom sme sa už stretli a znamená, že trieda, atribút alebo metóda sú dostupné pre volanie z akejkoľvek inej triedy.

private - Ak označíme metódu alebo atribút týmto modifikátorom, tak budú použiteľné len v rámci danej triedy.

Vyskúšajte pridať do triedy modifikátory

```
1   public class Auto {
2       private String model;
3       private int cena;
4
5       public Auto(String nazov, int hodnota){
6           model = nazov;
7           cena = hodnota;
8       }
9
10      public void popis() {
11          System.out.println("Som " + model + " a moja cena je " + cena + " eur.");
12      }
13  }
```

Všimnime si, že atribúty sme nastavili ako *private*. Týmto dosiahneme, že mimo triedy môžeme k nim pristupovať len nepriamo – pomocou konštruktora a metódy *popis*.

žiadny (default) - Ak modifikátor *public* alebo *private* vynecháme (ako v predchádzajúcej kapitole), prístup bude zabezpečený v rámci toho istého adresára (presnejšie v rámci toho istého *package*-u).

Iné modifikátory

Nasledujúce modifikátory nie sú alternatívou k predchádzajúcim. Ak chceme alebo potrebujeme, môžeme ich s nimi kombinovať. Pomáhajú nám dosiahnuť ďalšie zaujímavé vlastnosti.

Modifikátor static

Ak je atribút alebo metóda statická, znamená to, že **nesúvisí s inštanciou, ale s triedou**. Na príklade s autami vidíme statický atribút *pocetKolies*, ktorý súvisí s triedou. Ide o údaj, ktorý je spoločný pre celú kategóriu (triedu) áut. Nie je dovolené, aby každý druh auta (každá inštancia) mal vlastnú hodnotu tohto atribútu.

Podobne metódy označené ako statické majú zmysel na úrovni triedy, nie inštancie.

Vyskúšajte pridať do triedy statické atribúty a metódy

```java
public class Auto {
    private String model;
    private int cena;

    public static int pocetKolies = 4;

    public Auto(String nazov, int hodnota){
        model = nazov;
        cena = hodnota;
    }

    public void popis() {
        System.out.println("Som " + model + " a moja cena je " + cena + " eur.");
    }

    public static void nastavPocetKolies(int novyPocetKolies){
        pocetKolies = novyPocetKolies;
    }

    public static void vypisPocetKolies(){
        System.out.println("Auto ma " + pocetKolies + " kolies");
    }
}
```

K statickým atribútom a metódam pristupujeme väčšinou cez triedu (Auto), nie cez konkrétnu inštanciu (ako napr. audi, vw, ...).

Prístup k statickým atribútom a metódam „zvonku“

```java
Auto.pocetKolies = 4;
Auto.nastavPocetKolies(4);
Auto.vypisPocetKolies();
```

Modifikátor final

Tento modifikátor používame, ak chceme zabezpečiť **nemennosť**. Zamerajme sa na vytváranie konštánt. Konštanty môžu byť čísla, ktoré sa nemenia, ale môžu byť aj iného typu, napr. textové reťazce. Pri vytváraní konštánt používame okrem atribútu *final* aj atribút *static*. Modifikátor prístupu môžeme vynechať (*default*) alebo nastaviť *private* alebo *public*, podľa toho, čo potrebujeme. Je zvykom písať názvy konštánt veľkými písmenami.

Vyskúšajte si vytvoriť konštantu

```java
public class Auto {

    public static final int POCET_KOLIES = 4;

    public static void vypisPocetKolies(){
        System.out.println("Auto ma " + POCET_KOLIES + " kolies");
    }
}
```

V uvedenom príklade sme dosiahli, že nebude možné na inom mieste v kóde nastaviť novú hodnotu atribútu POCET_KOLIES.

Príklady

5.1 Príklad

Autobus má na svojej trase 5 zastávok a na každej zastávke nastupuje určitý počet ľudí. V hlavnom programe vytvorte pole dĺžky 5 s názvom *cakajuci*, ktoré naplňte počtom čakajúcich na jednotlivých zastávkach podľa vášho uváženia. Vytvorte triedu *Autobus*, ktorá bude mať atribút *aktualnyPocetLudi*. Vytvorte metódu *info*, ktorá vypíše informáciu s počtom ľudí, ktorý je práve v autobuse.

5.2 Príklad

V hlavnom programe vytvorte inštanciu triedy *Autobus*. V cykle budeme simulovať jazdu autobusu po jeho trase. Na každej zastávke čakajúci nastúpia, preto prirátajte počet čakajúcich do atribútu *aktualnyPocetLudi* (priradením cez =) a vypíšte informáciu s aktuálnym počtom ľudí v autobuse na danej zastávke.

5.3 Príklad

Nastavte modifikátor atribútu *aktualnyPocetLudi* na *private* a všimnite si aké chyby sa ukážu v programe. Do triedy autobus pridajte metódu *nastupuje*, ktorá dostane parameter *pocet* a priráta túto hodnotu do atribútu *aktualnyPocetLudi*. V hlavnom programe upravte navýšenie počtu cestujúcich na zastávke pomocou volania metódy *nastupuje*.

5.4 Príklad

Do triedy Autobus pridajte atribút *pocetMiest,* ktorý bude vyjadrovať maximálny počet cestujúcich. Nastavte ho v konštruktore. Upravte metódu *nastupuje* tak, aby vracala počet ľudí, ktorý musel zostať čakať na zastávke, pretože sa už nezmestili do autobusu. Ak sa niekto nezmestil, vypíšte v hlavnom programe, koľko ľudí sa nezmestilo. Skúste meniť kapacitu autobusu pri každom spustení programu a sledujte, či program funguje správne.

5.5 Príklad

Skúste vytvoriť podobnú triedu *Autobus2*, ktorá bude používať len statické atribúty a metódy a nebude mať konštruktor. Kapacitu autobusu spravte nemennú pomocou konštanty.

5.6 Príklad

Na predchádzajúcom príklade vysvetlite aké výhody a nevýhody prinieslo použitie modifikátora *static*.

Dedičnosť je významný princíp OOP, vďaka ktorému je možné jednoducho upravovať funkcionalitu tried. Vysvetlime si to na príklade.

Vytvorte rodičovskú triedu Zviera

```
1   public class Zviera {
2
3       public String meno;
4
5       public void nastavMeno(String menoZvierata){
6           meno = menoZvierata;
7       }
8
9       public void predstavSa(){
10          System.out.println("Som zviera. Moje meno je " + meno + ".");
11      }
12
13  }
```

Od triedy *Zviera* odvodíme ďalšie triedy. Odvodené triedy sa nazývajú aj **potomkovia** alebo **detské triedy** a hovoríme, že **dedia** od **rodičovskej** triedy. Dedenie realizujeme pomocou rezervovaného slova **extends**.

Vytvorte potomka triedy Zviera s názvom Pes

```
1   public class Pes extends Zviera {
2
3       public void predstavSa(){
4           System.out.println("Som PES. Moje meno je " + meno + ".");
5       }
6
7   }
```

Trieda *Pes* sa líši od triedy *Zviera* len v implementácii metódy *predstavSa*. Hovoríme, že metóda *predstavSa* je v triede *Pes* **predefinovaná**. Predefinovaná metóda musí mať rovnaký názov a parametre ako pôvodná metóda. Ostatné atribúty a metódy, ktoré sme nepredefinovali budú zdedené z rodičovskej triedy.

Referencia na rodičovskú triedu

Vyskúšajte použitie *super* v predefinovanej metóde

```
1   public class Macka extends Zviera {
2
3       public void predstavSa(){
4           System.out.print("Som MACKA. ");
5           super.predstavSa();
6       }
7   }
```

Ak chceme predefinovať metódu rodičovskej triedy, môžeme pritom využiť existujúcu logiku z pôvodnej rodičovskej triedy. Pomocou rezervovaného slova **super** pristupujeme k metódam v rodičovskej triede. V našom príklade sme v metóde *predstavSa* triedy *Macka* vypísali text pre

mačku a následne sme zavolali metódu *predstavSa* z triedy *Zviera*. Na nasledujúcom príklade si všimnite volanie metódy *predstavSa* na inštanciách rôzneho typu.

Vyskúšajte vytvoriť inštancie tried a zavolajte na nich metódu predstavSa

```
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Zviera zviera = new Zviera();
        Pes pes = new Pes();
        Macka macka = new Macka();

        zviera.nastavMeno("Jerry");
        pes.nastavMeno("Gandalf");
        macka.nastavMeno("Elza");

        zviera.predstavSa();
        pes.predstavSa();
        macka.predstavSa();
    }
}
```

Polymorfizmus

Z dedičnosti tried vyplýva aj dedičnosť typov, čo nám umožňuje uchovať inštancie detských tried v premenných rodičovského typu. Pre nás to znamená, že môžeme uchovať inštancie typu *Pes* alebo *Macka* v premenných typu *Zviera*.

Otázka: Ak je v premennej typu *Zviera* inštancia typu *Pes* a zavoláme na tejto premennej metódu *predstavSa*, zavolá sa metóda, ktorá je definovaná v triede *Zviera* alebo *Pes*?

Vďaka dôležitému princípu OOP, ktorý sa volá **polymorfizmus**, sa vždy zavolá metóda z tej triedy, akého je daná inštancia typu (nie podľa typu premennej). Takže odpoveď na otázku je, že sa zavolá metóda *predstavSa*, ktorá je predefinovaná v triede *Pes*.

Vyskúšajte si ako funguje polymorfizmus

```
public class Main {
    Run | Debug
    public static void main(String[] args) {
        Zviera zviera = new Zviera();
        Pes pes = new Pes();
        Macka macka = new Macka();

        zviera.nastavMeno("Jerry");
        pes.nastavMeno("Gandalf");
        macka.nastavMeno("Elza");

        Zviera [] zvierata = {zviera, pes, macka};

        for (Zviera z : zvierata){
            z.predstavSa();
        }
    }
}
```

Predstavíme si triedy z balíkov **javax.swing** a **java.awt**. Tieto triedy nám umožnia vybudovať rozhranie schopné zobrazovať grafické prvky.

Vyskúšajte si nasledujúci príklad

```java
import java.awt.Color;
import java.awt.Dimension;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class UI {
    Run | Debug
    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setVisible(true);
        frame.setTitle("Moje okno");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel();
        panel.setPreferredSize(new Dimension(500, 500));
        panel.setBackground(Color.BLUE);

        frame.add(panel);
        frame.pack();
    }
}
```

Tým, že budeme používať triedy zo spomenutých balíkov, musíme sa oboznámiť s tým, ako je zamýšľané ich použitie a náš kód tomu patrične prispôsobiť. Aj preto môže náš kód na prvý pohľad vyzerať komplikovane oproti tomu, aký výsledok sa zobrazí ☺. Netreba sa však nechať odradiť, prinesie to svoje ovocie.

Uvedený program nám zobrazí okno s prázdnou plochou modrej farby. Triedy, na ktoré sa budeme sústrediť sú **JFrame** a **JPanel**. *JFrame* reprezentuje vonkajšiu časť okna a môžeme si ho predstaviť ako obal alebo rám. Na rozdiel od toho *JPanel* reprezentuje vnútro okna a je kontajnerom pre ďalšie prvky, ktoré neskôr budeme umiestňovať dovnútra, ako napr. grafické objekty, tlačidlá, formulárové polia a podobne. *JPanel* je zodpovedný za zobrazenie a usporiadanie vnútorných prvkov okna.

Pomocou rezervovaného slova *new* vytvárame v programe inštancie pre *JFrame* a *JPanel*. Tieto inštancie, konkrétne zhotovenia tried *JFrame* a *JPanel*, nastavíme podľa našich potrieb a zobrazíme. Robíme to volaním metód na inštanciách tried:

JPanel

- *setPreferredSize* – Nastaví veľkosť vnútorného obsahu okna. Ako parameter sa posiela inštancia triedy *Dimension* s nastavením šírky a výšky.
- *setBackgroundColor* – Nastaví farbu pozadia vnútorného obsahu okna. Ako parameter sa posiela inštancia triedy *Color*, pričom využívame definovanú konštantu pre modrú farbu.

JFrame

- *setVisible* - Ako parameter prijíma *boolean*, podľa toho, či chceme aby bolo okno viditeľné alebo nie.
- *setDefaultCloseOperation* – Určuje, čo sa má stať, keď používateľ zatvorí okno. Konštanta JFrame.EXIT_ON_CLOSE, ktorú uvádzame ako parameter zabezpečí, že spolu so zatvorením okna skončí celý program.
- *add* – Slúži na pridanie komponentu do okna. V našom príklade pridávame kontajner *panel*, čím spájame rám okna s jeho vnútorným obsahom.
- *pack* – Voľne povedané, necháme triedu *JFrame* usporiadať prvky, ktoré obsahuje. Toto zahŕňa napríklad prispôsobenie veľkosti okna (*frame*) veľkosti vnútorného obsahu (*panel*). Všimnite si, že v programe sme veľkosť definovali len pre vnútro.

Využitie dedičnosti

V predchádzajúcej časti sme všetko vytvárali v *main* metóde. S rozširujúcou sa funkcionalitou a pribúdajúcim kódom by to nebolo najlepšie riešenie. Jedným z možných riešení je využiť dedičnosť.

Upravte predchádzajúci príklad. Vytvorte triedy, ktoré budú dediť z JFrame a JPanel.

```java
import javax.swing.*;
import java.awt.*;

class MyPanel extends JPanel {

    public MyPanel() {

        setPreferredSize(new Dimension(500, 500));
        setBackground(Color.BLUE);

    }
}

class MyFrame extends JFrame {

    public MyFrame(MyPanel myPanel){

        setVisible(true);
        setTitle("Moje okno");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(myPanel);
        pack();
    }
}

public class UI {
    Run | Debug
    public static void main(String[] args) {

        MyPanel panel = new MyPanel();
        new MyFrame(panel);
    }
}
```

Ako prvú sme vytvorili vlastnú triedu *MyPanel*, ktorá dedí od *JPanel*. Metódy, ktoré sme predtým volali na inštancii *JPanel* (*setPreferredSize, setBackground*) sú teraz metódami aj našej novej triedy – aj keď ich nevidíme v kóde, zdedili sme ich. Sú teda prístupné aj zvnútra triedy a môžeme ich zavolať v konštruktore našej novej triedy *MyPanel*.

Rovnakým štýlom postupujeme aj s triedou *MyFrame*, ktorá dedí od *JFrame*. Avšak navyše do konštruktora pridáme parameter *myPanel*. Potrebujeme ho, aby sme mali túto referenciu k dispozícii, keď budeme pridávať *myPanel* do *myFrame* pomocou metódy *add*.

Keďže všetku logiku súvisiacu s nastavením inštancií sme presunuli do konštruktorov, v *main* metóde nemusíme robiť nič viac, len vytvoriť inštancie. Všimnite si, že inštanciu triedy *MyFrame* ani nepriraďujeme do premennej. Inštanciu triedy *MyPanel* sme uložili do premennej *panel*, aby sme ju posunuli na nasledujúcom riadku do konštruktora *MyFrame*.

Zobrazenie grafických prvkov

V prípade, že chceme predefinovať nejakú zdedenú metódu, môžeme pridať anotáciu **@Override**. Tá jednak dáva informáciu, že ide o predefinovanú metódu, a zároveň aj vygeneruje kompilačnú chybu v prípade, že metóda s daným názvom a parametrami predefinováva neexistujúcu metódu z rodičovskej triedy. Môže to byť napríklad kvôli chybe v názve.

Pridajte do triedy MyPanel metódu paintComponent

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  class MyPanel extends JPanel {
5
6      public MyPanel() {
7
8          setPreferredSize(new Dimension(500, 500));
9          setBackground(Color.BLUE);
10     }
11
12     @Override
13     public void paintComponent(Graphics g){
14
15         super.paintComponent(g);
16         g.setColor(Color.yellow);
17         g.drawRect(200, 30, 100,100);
18         g.fillRect(10, 30, 100, 100);
19         g.fillOval(380, 30, 100, 100);
20     }
21 }
```

V tomto príklade sme predefinovali metódu *paintComponent*. V implementácii ponechávame funkcionalitu z rodičovskej triedy (volanie *super.paintComponent*) a pridávame vykreslenie niekoľkých útvarov.

Metóda *paintComponent* určuje ako sa má náš panel vykresliť. Nie je určená na volanie z nášho kódu, ale je automaticky zavolaná systémom pri inicializácii. Automaticky je poskytnutá aj inštancia triedy *Graphics* a my len povieme, čo chceme s ňou spraviť. Ďalšie metódy na vykreslenie útvarov ako aj popis parametrov možno nájsť na tejto adrese.

Príklady

Príklad 7.1

Vytvorte vlastné okno s vlastným nadpisom. Nastavte mu veľkosť 640x480 a pozadie zelenej farby.

Príklad 7.2

Vyskúšajte si vykresliť rôzne vlastné útvary, napr. štvorec, kruh. Nastavte im rôzne farby.

Príklad 7.3

Nastavte farbu pozadia vytvorením inštancie farby *Color(255,0,0)*. Potom zmeňte program tak, aby sa nastavovala náhodná farba pomocou vygenerovania troch náhodných čísel v intervale <0, 255> a ich vloženia do konštruktora triedy *Color*.

Príklad 7.4

Vytvorte triedu *Bod*, ktorá bude mať atribúty *x* a *y* typu *int* a pridajte jej konštruktor, ktorý ich nastaví. V hlavnom programe vytvorte *ArrayList<Bod>* a naplňte ho niekoľkými bodmi, pričom x a y budú ich súradnice. Pridajte atribút *body* do triedy *MyPanel* a upravte konštruktor, aby nastavil tento atribút. V metóde *paintComponent* v triede *MyPanel* vykreslite štvorce veľkosti 30 pre každý bod uložený v atribúte *body* tak, aby bod určoval ľavý horný roh štvorca.

Príklad 7.5

Premenujte triedu Bod z predchádzajúceho príkladu na *Stvorec*, pridajte atribúty *size* typu *int*, *color* typu *Color* a *vypln* typu *boolean.* Atribúty *x* a *y* ponechajte. V main metóde naplňte ArrayList<Stvorec> niekoľkými útvarmi, pričom všetky atribúty triedy *Stvorec* nech sú náhodne generované. Prispôsobte triedu *MyPanel*, aby vedela pracovať s novým objektom analogicky ako v predchádzajúcom príklade. V triede *MyPanel* vykreslite štvorce podobným štýlom, pričom ak je atribút *vypln True* štvorec vyplňte, ak je False vykreslite len obrysy. Nastavte pre každý útvar požadovanú farbu.

V ďalších príkladoch sa budeme stretávať aj s **interface**-ami, vysvetlíme si preto trochu teórie. Ak ste si pozreli implementáciu tried z java balíkov, možno ste si už všimli, že niekedy sa v deklarácii triedy vyskytovalo rezervované slovo **implements** nasledované názvom jedného, alebo viacerých **interface**-ov. Pozrime sa na *Scanner* a interface *Closeable*.

Deklarácia triedy java.util.Scanner

```
310  public final class Scanner implements Iterator<String>, Closeable {
```

Interface java.io.Closeable

```
37  public interface Closeable extends AutoCloseable {
38
39      /**
40       * Closes this stream and releases any system resources associated
41       * with it. If the stream is already closed then invoking this
42       * method has no effect.
43       *
44       * <p> As noted in {@link AutoCloseable#close()}, cases where the
45       * close may fail require careful attention. It is strongly advised
46       * to relinquish the underlying resources and to internally
47       * <em>mark</em> the {@code Closeable} as closed, prior to throwing
48       * the {@code IOException}.
49       *
50       * @throws IOException if an I/O error occurs
51       */
52      public void close() throws IOException;
53  }
```

Všimnite si, že metóda *close* v interface-i Closeable má len jeden riadok a nemá žiadnu implementáciu. Implementáciu musí obsahovať trieda, ktorá deklaruje, že interface *Closeable* implementuje, v našom prípade trieda *Scanner*. Ak si otvoríme triedu *Scanner*, nájdeme tam implementáciu metódy *close*. Rovnako tam musí byť aj implementácia metód z inerface-u *Iterator<String>,* keďže *Scanner* deklaruje, že implementuje aj tento interface.

Interface nám dáva záruku, že jeho metódy budú implementované v triedach, ktoré deklarujú, že tento interface implementujú. Zároveň týmto triedam poskytuje aj rodičovský typ. Vďaka tomu inštancia, ktorá je typu *Scanner*, je zároveň aj typu *Closeable*.

Udalosti

V používateľskom rozhraní budeme spracovávať rôzne udalosti, napríklad uplynutie intervalu časovača alebo stlačenie klávesu. Aby sa nám to spracovalo v správnom poradí a správne zobrazilo, zabalíme spustenie nášho kódu do nasledujúceho bloku.

Použite EventQueue.invokeLater

```
public static void main(String[] args) {
    EventQueue.invokeLater(() -> {
        MyPanel panel = new MyPanel();
        new MyFrame(panel);
    });
```

Timer

Ak chceme dať objekty, ktoré vykresľujeme do pohybu, budeme potrebovať časovač, ktorý nám poskytuje trieda *Timer*. Prvý parameter jej konštruktora je interval s akou frekvenciou má časovač generovať udalosti. Hodnota je v milisekundách. Druhý parameter je referencia na objekt typu *ActionListener*, ktorý tieto udalosti prijíma. Udalosť v tomto prípade je uplynutie intervalu časovača.

Náš panel teraz bude implementovať interface *ActionListener*, ktoré prináša metódu *actionPerformed*. Tá má parameter udalosť (*ActionEvent e*), ktorá nastala. V tomto prípade nás samotný objekt udalosti nezaujíma, preto s týmto parametrom nič nerobíme. Stačí nám, že udalosť nastala, čo sa prejaví tým, že metóda *actionPerformed* je zavolaná.

Pridajte časovač do panela

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.util.Random;
5
6  class MyPanel extends JPanel implements ActionListener {
7
8      private Timer timer = new Timer(1000, this);
9      private Random random = new Random(400);
10
11     public MyPanel() {
12         setPreferredSize(new Dimension(500, 500));
13         timer.start();
14     }
15
16     @Override
17     public void actionPerformed(ActionEvent e){
18         repaint();
19     }
20
21     @Override
22     public void paintComponent(Graphics g){
23         super.paintComponent(g);
24         g.setColor(Color.yellow);
25         g.fillRect(random.nextInt(400), random.nextInt(400), 100, 100);
26     }
27 }
```

Pripomíname, že metóda *paintComponent* nie je určená na priame volanie z nášho kódu a rovnaký efekt dosiahneme metódou *repaint*, ktorá spôsobí jej nepriame zavolanie. Zároveň platí, že vďaka interface-u *ActionListener* náš panel získal aj rodičovský typ *ActionListener,* a preto ho môžeme nastaviť ako prijímač udalostí pre *Timer*. Robíme to cez *this*, ktoré posielame do konštruktora *Timera*. *this* je referencia vlastnej inštancie.

Príklady

Príklad 8.1

Vytvorte program, ktorý každú sekundu vykreslí štvorec, vždy na inej náhodnej pozícii.

Vytvorte vlastnú triedu, ktorá bude dediť z KeyAdapter

```java
import java.awt.event.*;

public class MyKeyAdapter extends KeyAdapter {

    boolean flag = true;

    @Override
    public void keyPressed(KeyEvent e) {
        if (flag){
            flag = false;
        } else {
            flag = true;
        }
    }
}
```

Na zachytávanie udalostí, ktoré sa vygenerujú pri stlačení klávesu vytvoríme triedu, ktorá dedí z *KeyAdapter* a predefinuje metódu *keyPressed*. Pri každom stlačení klávesy budeme nastavovať *boolean* atribút *flag* striedavo na *true* a *false*. Aby sme mohli zachytávať udalosti po stlačení klávesy, musíme upraviť aj panel, a to nastaviť *setFocusable* a zaregistrovať cez metódu *addKeyListener* náš *keyAdapter*.

Pridajte MyKeyAdapter do panelu

```java
class MyPanel extends JPanel implements ActionListener {

    private Timer timer = new Timer(1000, this);
    private Random random = new Random(400);
    private MyKeyAdapter keyAdapter = new MyKeyAdapter();

    public MyPanel() {
        setPreferredSize(new Dimension(500, 500));
        setFocusable(true);
        addKeyListener(keyAdapter);
        timer.start();
    }

    // ...
    // ...
```

Teraz môžeme v metóde *paintComponent* pristupovať k inštancii *keyAdapter* typu *MyKeyAdapter* a vykresliť iný objekt, keď je jeho atribút *flag* nastavený na *true* a iný keď je nastavený na *false*.

V *MyKeyAdapteri* by sme si mohli zapamätať, aké písmeno bolo stlačené naposledy a podľa toho nastaviť farbu. K tomu už potrebujeme spracovať udalosť, ktorá nám prišla ako parameter do metódy *keyPressed*. Trieda *KeyEvent* nám poskytuje metódu *getKeyCode*, ktorá nám vráti kód stlačeného klávesu. Aby sme si nemuseli pamätať číselné kódy jednotlivých klávesov a aby bol kód čitateľnejší, používame definované konštanty v triede *KeyEvent*. Napríklad pre "*R*" je to konštanta *KeyEvent.VK_R*. Avšak, ak by sme chceli, mohli by sme namiesto konštanty použiť aj pridelený číselný kód.

Na základe stlačeného klávesu, nastavte farbu

```java
import java.awt.*;
import java.awt.event.*;

public class MyKeyAdapter extends KeyAdapter {
    Color color = Color.BLACK;

    @Override
    public void keyPressed(KeyEvent e) {

        int key = e.getKeyCode();
        //kláves R
        if (key == KeyEvent.VK_R) {
            color = Color.RED;
        }
        //kláves B
        if (key == KeyEvent.VK_B) {
            color = Color.BLUE;
        }
        //kláves G
        if (key == KeyEvent.VK_G) {
            color = Color.GREEN;
        }
    }
}
```

Príkaz switch

V predchádzajúcom príklade pri zisťovaní aký kláves bol stlačený sme použili niekoľkokrát príkaz *if*. Kvôli prehľadnosti môžeme použiť aj príkaz **switch**.

Použite príkaz switch, aby ste zistili, aký kláves bol stlačený

```java
switch (key){
    case KeyEvent.VK_R:
        color = Color.RED;
        break;
    case KeyEvent.VK_B:
        color = Color.BLUE;
        break;
    case KeyEvent.VK_G:
        color = Color.GREEN;
        break;
    default:
        color = Color.BLACK;
}
```

Za *switch* nasleduje v zátvorkách premenná, ktorú testujeme. Ďalšou jeho časťou je blok v zložených zátvorkách. Tento blok je tvorený *case* vetvami, prípadne vetvou *default* na konci. Po príkaze *case* nasleduje vždy možnosť, ktorá by mohla nastať na testovanej premennej v príkaze switch. Za dvojbodkou nasleduje kód, ktorý sa má vykonať v prípade, že daná možnosť nastala. Nesmie chýbať príkaz *break*, pretože inak by sa vykonali aj nasledujúce možnosti aj napriek tomu, že daná situácia nenastala. Slovo *default* sa používa pre všetky ostatné možnosti, ktoré predtým neboli vymenované za *case* príkazmi.

Príklady

Príklad 9.1

Upravte príklad 8.1 tak, aby sa pri stlačení ľubovoľného klávesu začali vykresľovať kruhy namiesto štvorcov a pri opätovnom stlačení znovu kruhy. Toto nech sa opakuje až do ukončenia programu.

Príklad 9.2

Vykreslite do panelu kruh.
Podľa toho aká šípka je stlačená, nech sa nastaví aktuálny smer a nech sa kruh rozhýbe daným smerom, ale nech nikdy „nevyjde“ mimo rám. Pohyb môže byť napr. o 10 bodov v aktuálnom smere každú sekundu.

Príklad 9.3

Upravte predchádzajúci príklad tak, že ak kruh „narazí“ na rám, tak nech program zastaví a vykreslí sa čierne pozadie. Časovač zastavte pomocou metódy *Timer.stop()*.

10. Hodina – Hra Snake

V tejto kapitole poskytneme kostru programu. Vašou úlohou bude doprogramovať chýbajúci kód na miestach označených ako *TODO* tak, aby sme dostali funkčnú verziu **hry Snake**.

Skopírujte si do vášho prostredia nasledujúci kód:

Snake.java

```
import java.awt.*;

public class Snake {
    public static void main(String[] args) {
       EventQueue.invokeLater(() -> {
            Game game = new Game();
            MyPanel panel = new MyPanel(game);
            new MyFrame(panel);
       });
    }
}
```

MyFrame.java

```
import javax.swing.*;

public class MyFrame extends JFrame {

    public MyFrame(MyPanel myPanel){
        setVisible(true);
        setTitle(""); //TODO: nastavte názov vašej hry
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(myPanel);
        pack();
    }
}
```

MyKeyAdapter.java

```
import java.awt.event.*;

public class MyKeyAdapter extends KeyAdapter {
    private int keyCode = KeyEvent.VK_LEFT;

    @Override
    public void keyPressed(KeyEvent e) {
        keyCode = e.getKeyCode();
    }

    public int getKeyCode(){
        return keyCode;
    }
}
```

MyPanel.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class MyPanel extends JPanel implements ActionListener {

    private Game game;
    private Timer timer = new Timer(100, this);
    private MyKeyAdapter keyAdapter = new MyKeyAdapter();

    public MyPanel(Game game) {
        this.game = game;
        setPreferredSize(new Dimension(Game.WIDTH, Game.HEIGHT));
        setFocusable(true);
        addKeyListener(keyAdapter);
        timer.start();
    }

    @Override
    public void actionPerformed(ActionEvent e){
        //TODO
    }

    @Override
    public void paintComponent(Graphics g){
        super.paintComponent(g);
        g.setColor(Game.COLOR);
        game.drawSnake(g);
        game.takeFruit();
        game.drawFruit(g);
    }
}
```

Point.java

```
public class Point{

    private int x;
    private int y;

    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }

    public int getX(){
        return x;
    }

    public int getY(){
        return y;
    }
}
```

Game.java

```java
import java.awt.*;
import java.awt.event.*;
import java.util.ArrayList;
import java.util.Random;

public class Game {

    public static int WIDTH = 500;
    public static int HEIGHT = 500;
    public static Color COLOR = Color.BLUE;
    private String direction = "left";

    private Random random = new Random();
    private ArrayList<Point> snake = new ArrayList<>();
    private Point fruit;
    private boolean fruitTaken = false;

    private Point createFruit(){
        //TODO
        return null;
    }

    public Game(){
        //TODO
    }

    public void move(int keyCode){
       //TODO
    }

    public void takeFruit(){
        //TODO
    }

    public boolean gameOver(){
        //TODO
        return true;
    }

    public void drawSnake(Graphics g){
        //TODO
    }

    public void drawFruit(Graphics g){
        //TODO
    }
}
```

Po vytvorení príslušných súborov a tried by ste mali dostať spustiteľný program. Spúšťame triedu *Snake*, ktorá obsahuje metódu *main*. Kód z ukážky vyššie zobrazí len prázdne okno. V nasledujúcich príkladoch postupne vytvoríme hru. Po správnom dokončení každého príkladu by váš program mal byť spustiteľný.

Príklady

Príklad 10.1

i) Prezrite si kód

Dobre si prezrite pripravenú kostru kódu a skúste sa zamyslieť, ako by to mohlo fungovať po doplnení TODO častí.

ii) Nastavte názov hry

Vymyslite názov pre hru a upravte program tak, aby sa váš názov zobrazoval ako titulok rámu. Následne vymažte TODO v súbore MyFrame.java

Príklad 10.2

i) Vytvorte reprezentáciu hada

Hada budeme reprezentovať ako 5 štvorčekov veľkosti 10x10, pričom každý štvorček bude určený súradnicami jeho ľavého horného rohu. Súradnice bodov nech sú násobky 10. V konštruktore *Game* vytvorte hada tak, že naplníte atribút *snake* typu *ArrayList<Point>* piatimi bodmi. Na reprezentovanie súradníc bodov využite poskytnutú triedu *Point*. Nastavte súradnice bodov tak, aby pri vykreslení boli štvorčeky vedľa seba zľava doprava. Umiestnite hada približne do stredu, aby po rozhýbaní hneď nenarazil na stenu.

ii) Vykreslite hada

Implementujte metódu *drawSnake* triedy *Game.* Prejdite *ArrayList snake* v cykle a vykreslite jednotlivé body, ktoré tvoria hada ako štvorčeky 10 x 10.

Príklad 10.3

i) Posuňte informáciu o stlačenej klávese do triedy *Game*

MyKeyAdapter nám drží informáciu o tom, aký kláves bol stlačený naposledy. Zároveň poskytuje metódu, aby sa tento kód dal získať zvonku. V triede *MyPanel* v metóde *actionPerformed* získajte kód tejto klávesy a následne zavolajte metódu *move* na inštancii hry. Nakoniec zavolajte metódu *repaint*.

ii) Vypíšte informáciu o stlačenom klávese

V metóde *move* zistite aká šípka bola stlačená a podľa toho do terminálu vypíšte: vľavo, vpravo, hore, dole. Kódy klávesov nájdete v triede *java.awt.event.KeyEvent.*

Príklad 10.4

i) Posuňte hada v správnom smere.

Vykonáme to tak, že odstránime jeho posledný bod (chvost) a pridáme mu na začiatok nový bod (hlavu). Aktuálnu hlavu aj chvost vieme nájsť správnym indexom v *ArrayListe*, ktorý tvorí hada. Nová hlava (bod, ktorý ju reprezentuje) bude mať súradnice odvodené od aktuálnej hlavy podľa toho, aká šípka bola naposledy stlačená, teda podľa toho, ktorým smerom ide had. Ak *<x, y>* sú súradnice aktuálnej hlavy a had ide vpravo, nová hlava bude mať súradnice *<x+10, y>.* Novú hlavu pridajte do *ArrayListu* pomocou *add(0, ...)*.

Príklad 10.5

i) Ukončenie hry

Implementujte metódu *gameOver* aby vrátila *true* v prípade, že hlava hada narazí na okraj rámu. V opačnom prípade vráťte *false*. Metódu *gameOver* potom volajte v metóde *actionPerformed* triedy *MyPanel* po volaní *move* na inštancii hry. Metódu *repaint* zavolajte len ak hra ešte neskončila. V opačnom prípade vypnite *timer* volaním *timer.stop().*

Príklad 10.6

i) Vytvorte ovocie

Ovocie budeme reprezentovať ako kruh. Implementujte metódu *createFruit*, ktorá vráti inštanciu triedy *Point* s náhodnými súradnicami v rámci okna. V konštruktore *Game* priraďte vygenerované ovocie do atribútu *fruit*.

ii) Vykreslite ovocie

Implementuje metódu *drawFruit*, ktorá vykreslí ovál rozmerov 10x10 na súradniciach určených atribútom *fruit*.

Príklad 10.7

i) Had žerie ovocie

Ak sú súradnice určujúce hlavu hada a ovocie rovnaké znamená to, že had zožerie ovocie. V tom prípade musíme vygenerovať nové ovocie na inom mieste. Implementujte metódu *takeFruit* tak, že keď had žerie ovocie, atribút *fruit* nastavíte pomocou metódy *createFruit* na nové ovocie.

ii) Kontrola súradníc

Dajte pozor na to, že hada posúvame vždy o 10 bodov a ovocie generujeme nielen na súradniciach násobku 10. Preto sa veľmi často stane, že súradnice hlavy hada a ovocia sa nebudú zhodovať – nastavte preto nejakú toleranciu na túto kontrolu.