Jasně! Teď se pustíme do Javy a pokusím se to vysvětlit co nejpodrobněji a ukázat ti co nejvíce tipů a technik, které ti mohou pomoct při práci s tímto jazykem.

---

**1. Základní struktura programu v Javě**

Každý Java program se skládá z **tří hlavních částí**:

- **Třída** (class) – obsahuje metody a vlastnosti.
- **Metoda** (method) – obsahuje logiku, kterou bude program vykonávat.
- **Hlavní metoda** (main) – místo, kde začíná vykonávání programu.

```
public class HelloWorld {
   public static void main(String[] args) {
      // Tento kód bude vykonán, když spustíš program.
      System.out.println("Hello, World!");
   }
}
```

Tento program vypíše do konzole Hello, World!.

---

**2. Základní datové typy**

Java má několik základních datových typů, které lze rozdělit do dvou kategorií: **primitive types** a **reference types**.

- **Primitive types**:
    - int – celočíselné hodnoty
    - double – desetinné hodnoty
    - char – znak (jediný znak v uvozovkách)
    - boolean – pravda nebo nepravda
    - byte, short, long, float – jiné číselné typy pro specifické účely

Příklad:

int vek = 25;

double cena = 19.99;

boolean jeDost = true;

char pismeno = 'A';

- **Reference types**:
    - **String** – řetězec znaků (důležitý typ, ale technicky není primitivní)
    - **Array** – pole
    - **Class** – objekty tříd, které si vytvoříš
    - **Interface** a **Enum** – pokročilé typy pro design patterny

Příklad:

String jmeno = "Anna";

int[] cisla = {1, 2, 3};

---

**3. Podmínky a cykly**

V Javě používáme klasické **if**, **else if**, **else**, a **switch** pro podmínky. Dále cykly **for**, **while**, a **do-while**.

- **if/else**:

```
int vek = 20;
if (vek >= 18) {
   System.out.println("Plnoletý");
} else {
   System.out.println("Neplnoletý");
}
```

- **switch**:

```
int den = 3;
switch (den) {
   case 1:
      System.out.println("Pondělí");
      break;
   case 2:
      System.out.println("Úterý");
```

```
    break;
  case 3:
    System.out.println("Středa");
    break;
  default:
    System.out.println("Neznámý den");
    break;
}
```

- **for cyklus**:

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);  // Výstup: 0, 1, 2, 3, 4
}
```

- **while cyklus**:

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

---

**4. Třídy a objekty**

V Javě je vše organizováno do **tříd**. Třída je šablona, která definuje, jaké vlastnosti a metody bude objekt mít.

- **Definice třídy**:

```
class Osoba {
  String jmeno;
  int vek;

  // Konstruktor
```

```
    public Osoba(String jmeno, int vek) {
        this.jmeno = jmeno;
        this.vek = vek;
    }

    // Metoda
    public void pozdrav() {
        System.out.println("Ahoj, jmenuji se " + jmeno);
    }
}
```

- **Vytvoření objektu**:

```
public class Main {
    public static void main(String[] args) {
        Osoba osoba1 = new Osoba("Anna", 25);
        osoba1.pozdrav();  // Výstup: Ahoj, jmenuji se Anna
    }
}
```

---

**5. Konstruktor a inicializace objektů**

Konstruktor je speciální metoda, která se používá pro inicializaci objektu při jeho vytvoření. Můžeš mít více konstruktorů s různými parametry, což se nazývá **přetížení konstruktoru**.

```
class Osoba {
    String jmeno;
    int vek;

    // Konstruktor
    public Osoba(String jmeno, int vek) {
        this.jmeno = jmeno;
```

```
        this.vek = vek;
    }

    public Osoba(String jmeno) {
        this.jmeno = jmeno;
        this.vek = 0;  // Pokud není věk zadaný, nastaví se na 0
    }
}
```

---

**6. Dědičnost (Inheritance)**

Java podporuje **dědičnost**, což znamená, že můžeš vytvořit novou třídu, která dědí vlastnosti a metody z existující třídy.

- **Příklad dědičnosti**:

```
class Zvire {
    public void zvuk() {
        System.out.println("Zvuk zvířete");
    }
}

class Pes extends Zvire {
    @Override
    public void zvuk() {
        System.out.println("Haf!");
    }
}

public class Main {
    public static void main(String[] args) {
```

```
    Pes pes = new Pes();
    pes.zvuk();  // Výstup: Haf!
  }
}
```

Třída **Pes** dědí metodu **zvuk()** od třídy **Zvire**, ale přetíží ji pro vlastní implementaci.

---

**7. Polymorfismus (Polymorphism)**

**Polymorfismus** znamená, že objekt může být více než jednoho typu. Můžeš například volat metody objektu, aniž bys věděl přesně, jaký typ objektu je, pokud všechny implementují stejnou metodu.

```
class Zvire {
  public void zvuk() {
    System.out.println("Zvuk zvířete");
  }
}

class Pes extends Zvire {
  @Override
  public void zvuk() {
    System.out.println("Haf!");
  }
}

class Kocka extends Zvire {
  @Override
  public void zvuk() {
    System.out.println("Mňau!");
  }
}
```

```
public class Main {
    public static void main(String[] args) {
        Zvire z1 = new Pes();
        Zvire z2 = new Kocka();

        z1.zvuk();  // Výstup: Haf!
        z2.zvuk();  // Výstup: Mňau!
    }
}
```

I když jsou objekty **Pes** a **Kocka** různých typů, mohou být oba přiřazeny do proměnné typu **Zvire** a vyvolat odpovídající metodu.

---

**8. Abstraktní třídy a rozhraní**

- **Abstraktní třída** je třída, která nemůže být přímo instanciována, ale může obsahovat abstraktní metody, které musí být implementovány v podtřídách.

```
abstract class Zvire {
    abstract void zvuk();  // Abstraktní metoda, která musí být implementována v podtřídách
}

class Pes extends Zvire {
    @Override
    void zvuk() {
        System.out.println("Haf!");
    }
}
```

- **Rozhraní (interface)** definuje pouze metody, které musí třída implementovat, ale nenabízí žádnou implementaci.

```
interface Zvuk {
    void vydatZvuk();
}

class Pes implements Zvuk {
    @Override
    public void vydatZvuk() {
        System.out.println("Haf!");
    }
}
```

**9. Kolekce (Collections)**

V Javě máme různé **kolekce** (například **List**, **Set**, **Map**), které jsou součástí balíčku **java.util**.

- **ArrayList**:

```
import java.util.ArrayList;

ArrayList<String> seznam = new ArrayList<>();
seznam.add("Jablko");
seznam.add("Banán");
System.out.println(seznam.get(0));  // Výstup: Jablko
```

- **HashMap** (Klíč-hodnota):

```
import java.util.HashMap;

HashMap<String, Integer> mapa = new HashMap<>();
mapa.put("Anna", 25);
mapa.put("Petr", 30);
System.out.println(mapa.get("Anna"));  // Výstup: 25
```

---

**10. Výjimky (Exceptions)**

Java má silný mechanismus pro zachytávání a zpracování **výjimek** pomocí try, catch a finally.

```
try {
   int result = 10 / 0;  // Tady dojde k výjimce
} catch (ArithmeticException e) {
   System.out.println("Chyba: dělení nulou");
} finally {
   System.out.println("Toto se provede vždy, i když dojde k výjimce.");
}
```

---

Pokračujeme, pojďme se podívat na další pokročilé koncepty v Javě, které ti mohou pomoci nejen pro maturitu, ale i pro komplexnější aplikace.

---

**11. Generika (Generics)**

**Generika** umožňují psát kód, který je **typově bezpečný** a zároveň **flexibilní**. Pomocí generik můžeš psát třídy, metody nebo rozhraní, které pracují s různými typy, ale přitom zachovávají typovou bezpečnost.

- **Generické třídy**:

```
class Box<T> {
   private T value;

   public void set(T value) {
      this.value = value;
   }
```

```
    public T get() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        Box<Integer> intBox = new Box<>();
        intBox.set(10);
        System.out.println(intBox.get());  // Výstup: 10

        Box<String> strBox = new Box<>();
        strBox.set("Hello");
        System.out.println(strBox.get());  // Výstup: Hello
    }
}
```

- **Generické metody**:

```
public class Utility {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.println(element);
        }
    }

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3};
        String[] strArray = {"A", "B", "C"};
```

```
        printArray(intArray);  // Výstup: 1 2 3
        printArray(strArray);  // Výstup: A B C
    }
}
```

Generika ti umožní pracovat s různými datovými typy, aniž bys musela psát opakující se kód pro každý datový typ.

---

**12. Lambda výrazy a funkcionální programování**

Lambda výrazy byly přidány v JDK 8 a umožňují funkcionální styl programování v Javě. Používají se pro definování **anonymních funkcí**, které mohou být předány metodám.

- **Syntaxe lambda výrazu**:

```
interface Operace {
    int vypocitej(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        // Lambda výraz pro sečítání
        Operace soucet = (a, b) -> a + b;
        System.out.println(soucet.vypocitej(5, 3));  // Výstup: 8
    }
}
```

Lambda výraz (a, b) -> a + b znamená, že metoda vypocitej s parametry a a b vrátí součet těchto dvou čísel.

- **Použití s kolekcemi**:

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
```

```
    List<String> jmena = Arrays.asList("Anna", "Petr", "Eva");


    // Použití lambda výrazu pro iteraci přes seznam
    jmena.forEach(jmeno -> System.out.println(jmeno));
  }
}
```

Lambda výrazy jsou užitečné pro zjednodušení kódu, když používáš funkce jako map, filter, reduce a další funkcionální operace.

---

**13. Stream API (JDK 8)**

Stream API umožňuje efektivní práci s kolekcemi a provádění operací jako filtrace, transformace nebo agregace dat bez nutnosti explicitního psaní cyklů.

- **Příklad Stream API pro filtraci a zpracování seznamu**:

```
import java.util.*;


public class Main {
  public static void main(String[] args) {
    List<Integer> cisla = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);


    // Filtrace a zpracování dat
    cisla.stream()
        .filter(cislo -> cislo % 2 == 0)  // Filtrujeme pouze sudá čísla
        .map(cislo -> cislo * 2)        // Násobíme každý prvek 2
        .forEach(System.out::println);    // Vytiskneme výsledky
  }
}
```

Tento kód:

1. Filtruje pouze sudá čísla.
2. Každé číslo vynásobí dvěma.

3. Výsledky vytiskne na obrazovku.

---

**14. Kolekce a rozhraní List, Set, Map**

V Javě existují různé kolekce pro ukládání a manipulaci s daty. Nejčastěji používané jsou **List**, **Set** a **Map**.

- **List** (např. **ArrayList**):
    - Ukládá hodnoty v pořadí.
    - Může obsahovat duplicitní hodnoty.

```
List<String> seznam = new ArrayList<>();
seznam.add("Jablko");
seznam.add("Banán");
seznam.add("Jablko");  // Může obsahovat duplicity
```

- **Set** (např. **HashSet**):
    - Neumožňuje duplicity.
    - Nezaručuje pořadí prvků.

```
Set<String> ovoce = new HashSet<>();
ovoce.add("Jablko");
ovoce.add("Banán");
ovoce.add("Jablko");  // Duplicitní prvek bude ignorován
```

- **Map** (např. **HashMap**):
    - Ukládá páry **klíč-hodnota**.
    - Klíče musí být unikátní.

```
Map<String, Integer> ceny = new HashMap<>();
ceny.put("Jablko", 25);
ceny.put("Banán", 15);
```

---

**15. Synchronizace a vlákna (Threads)**

Pro práci s **vlákny** a **synchronizací** je v Javě několik metod. Každé vlákno může běžet paralelně s ostatními a je nutné ošetřit přístup ke sdíleným datům, aby nedocházelo k chybám.

- **Vytvoření vlákna**:

```
class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("Vlákno běží!");
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();  // Spustí nové vlákno
    }
}
```

- **Synchronizace metod**: Pokud má více vláken přístup k nějaké metodě nebo proměnné, je potřeba metodu **synchronizovat**, aby se předešlo problémům se sdílením dat.

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }
}
```

Tímto způsobem se zajistí, že vlákna nebudou do proměnné **count** zapisovat současně.

---

**16. Výjimky a vlastní výjimky**

V Javě můžeš vytvořit **vlastní výjimky**, které ti umožní lépe zpracovávat specifické chyby v aplikaci.

- **Vytvoření vlastní výjimky**:

```
class MojeVyjimka extends Exception {
    public MojeVyjimka(String message) {
        super(message);
    }
}

public class Main {
    public static void main(String[] args) {
        try {
            throw new MojeVyjimka("Tohle je vlastní výjimka!");
        } catch (MojeVyjimka e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Tato výjimka je děděná od základní **Exception** a můžeš ji použít k zajištění lepšího zachytávání chyb.

---

Jasně, pokračujme tedy s opravdu **základními** a **klíčovými** věcmi v Javě, které bys měla mít dobře zvládnuté pro maturitu. Tentokrát se zaměříme na **opravdové základy**, které jsou často základem každého zadání a pro správné pochopení ostatních pokročilejších konceptů.

---

### 1. Základní struktura třídy a objektu

Každý program v Javě je postaven kolem třídy a objektů. Třída je **šablona** pro objekty, které obsahují **atributy** (nebo proměnné) a **metody** (nebo funkce).

- **Třída a objekt**:

```
class Osoba {
    // Atributy (proměnné)
    String jmeno;
    int vek;

    // Metoda (funkce)
    void predstavSe() {
        System.out.println("Ahoj, jmenuji se " + jmeno + " a je mi " + vek + " let.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Vytvoření objektu
        Osoba osoba1 = new Osoba();
        osoba1.jmeno = "Eva";
        osoba1.vek = 25;
        osoba1.predstavSe();  // Výstup: Ahoj, jmenuji se Eva a je mi 25 let.
    }
}
```

Třída **Osoba** definuje dva atributy (jmeno, vek) a jednu metodu (predstavSe()). V metodě **main** vytváříme objekt **osoba1** a nastavujeme jeho vlastnosti.

---

### 2. Konstruktor třídy

Konstruktor je speciální metoda, která se používá k inicializaci objektu při jeho vytvoření. Když vytvoříš nový objekt třídy, automaticky se zavolá konstruktor.

- **Konstruktor**:

```
class Osoba {
    String jmeno;
    int vek;

    // Konstruktor
    Osoba(String jmeno, int vek) {
        this.jmeno = jmeno;
        this.vek = vek;
    }

    void predstavSe() {
        System.out.println("Ahoj, jmenuji se " + jmeno + " a je mi " + vek + " let.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Použití konstruktoru pro inicializaci objektu
        Osoba osoba1 = new Osoba("Petr", 30);
        osoba1.predstavSe();  // Výstup: Ahoj, jmenuji se Petr a je mi 30 let.
    }
}
```

Tento kód ukazuje, jak používat **konstruktor** pro inicializaci hodnot při vytváření objektu.

---

**3. Datové typy a proměnné**

Jazyk Java je silně **typově bezpečný**, což znamená, že každá proměnná musí mít jasně definovaný typ. Existují **primitivní datové typy** a **referenční datové typy**.

- **Primitivní datové typy**:
    - **int** – celočíselný typ
    - **double** – desetinný typ
    - **char** – znak
    - **boolean** – logická hodnota (true nebo false)
    - **byte** – malý celočíselný typ
    - **short** – střední celočíselný typ
    - **long** – velký celočíselný typ
    - **float** – desetinný typ s menší přesností než double

Příklad:

int cislo = 10;

double cena = 12.99;

boolean jePravda = true;

char znak = 'A';

- **Referenční datové typy**:
    - **String** – řetězec znaků
    - **Pole** (Array) – kolekce hodnot stejného typu
    - **Objekty** – třídy, které definují strukturu a chování objektů.

---

**4. Podmínky (if, else, switch)**

Podmínky jsou základem každé logiky v programu. Používají se k rozhodování, zda provést určitou akci nebo ne na základě toho, jestli nějaká podmínka platí.

- **if-else**:

```
int vek = 18;


if (vek >= 18) {
   System.out.println("Jsi dospělý.");
```

```
} else {
   System.out.println("Jsi mladistvý.");
}
```

- **switch-case**: Pokud máš více možností, můžeš použít **switch** pro efektivní rozhodování mezi různými hodnotami.

```
int den = 3;
switch (den) {
   case 1:
      System.out.println("Pondělí");
      break;
   case 2:
      System.out.println("Úterý");
      break;
   case 3:
      System.out.println("Středa");
      break;
   default:
      System.out.println("Neznámý den");
      break;
}
```

---

**5. Cyklus (for, while, do-while)**

Cyklus slouží k **opakování určité akce** několikrát. Existují různé typy cyklů v Javě, ale nejběžnější jsou for, while, a do-while.

- **for cyklus** (používá se, když víme, kolikrát chceme cyklus spustit):

```
for (int i = 1; i <= 5; i++) {
   System.out.println("Hodnota i: " + i);
}
```

- **while cyklus** (používá se, když nevíme, kolikrát chceme cyklus spustit, ale víme, že cyklus bude pokračovat, dokud podmínka bude pravdivá):

```
int i = 1;
while (i <= 5) {
  System.out.println("Hodnota i: " + i);
  i++;
}
```

- **do-while cyklus** (podobné while, ale cyklus se provede alespoň jednou):

```
int i = 1;
do {
  System.out.println("Hodnota i: " + i);
  i++;
} while (i <= 5);
```

---

**6. Metody**

Metody slouží k **organizaci kódu** a **opakovanému využívání** funkcí. V Javě metody definují určité chování objektů nebo třídy.

- **Syntaxe metody**:

```
public class Main {
  // Metoda bez návratové hodnoty (void)
  static void pozdrav(String jmeno) {
    System.out.println("Ahoj, " + jmeno + "!");
  }

  public static void main(String[] args) {
    pozdrav("Petr");  // Výstup: Ahoj, Petr!
  }
}
```

Metoda pozdrav přijímá jeden argument (jméno) a vypíše pozdrav.

---

**7. Třídy a objekty (pokračování)**

Pokud chceš, aby tvá třída fungovala jako objektová aplikace, je důležité pochopit také **konstruktory**, **getter a setter metody**, a princip **zapouzdření**.

- **Getter a setter** metody slouží k **přístupu** a **změně hodnoty** atributů objektu. **Zapouzdření** znamená, že bychom neměli přímo měnit hodnoty atributů, ale použít metody.

```
class Osoba {
  private String jmeno;  // Atribut je soukromý, přístup je pouze přes metody

  // Getter metoda
  public String getJmeno() {
    return jmeno;
  }

  // Setter metoda
  public void setJmeno(String jmeno) {
    this.jmeno = jmeno;
  }
}
```

---

Pokračujme tedy s dalšími **základy** v Javě, které jsou opravdu klíčové pro **porozumění objektově orientovanému programování** a pro maturitu. Pojďme se podívat na:

**1. Dědičnost (Inheritance)**

Dědičnost je **základní princip** objektově orientovaného programování, který umožňuje jedné třídě dědit vlastnosti a metody jiné třídy. Tímto způsobem se **zjednodušuje kód** a **znovu použitelnost**.

- **Super třída** a **podtřída**:

```
class Zviratko {
  String jmeno;
```

```
    // Konstruktor
    Zviratko(String jmeno) {
        this.jmeno = jmeno;
    }

    void zvuk() {
        System.out.println("Zvířátko vydává zvuk.");
    }
}

class Pes extends Zviratko {
    // Konstruktor podtřídy volá konstruktor nadtřídy
    Pes(String jmeno) {
        super(jmeno);
    }

    // Přepsání metody
    @Override
    void zvuk() {
        System.out.println(jmeno + " štěká.");
    }
}

public class Main {
    public static void main(String[] args) {
        Zviratko mojeZviratko = new Zviratko("Zvířátko");
        Pes mujPes = new Pes("Benny");
```

```
    mojeZviratko.zvuk();  // Výstup: Zvířátko vydává zvuk.
    mujPes.zvuk();  // Výstup: Benny štěká.
  }
}
```

V tomto příkladu třída **Pes** dědí od třídy **Zviratko** a přepisuje metodu **zvuk()**, která je v základní třídě. **super()** volá konstruktor nadtřídy.

---

**2. Polymorfismus (Polymorphism)**

Polymorfismus znamená, že můžeme mít metody, které **mají stejný název**, ale **jinou implementaci** v různých třídách. Také to umožňuje používání **obecných referencí** na objekty různých tříd.

- **Příklad polymorfismu:**

```
class Zviratko {
  void zvuk() {
    System.out.println("Zvířátko vydává zvuk.");
  }
}

class Pes extends Zviratko {
  @Override
  void zvuk() {
    System.out.println("Pes štěká.");
  }
}

class Kocka extends Zviratko {
  @Override
  void zvuk() {
```

```
    System.out.println("Kočka mňouká.");
  }
}

public class Main {
  public static void main(String[] args) {
    Zviratko mojeZviratko = new Zviratko();
    Zviratko mujPes = new Pes();
    Zviratko mojeKocka = new Kocka();

    mojeZviratko.zvuk();  // Výstup: Zvířátko vydává zvuk.
    mujPes.zvuk();       // Výstup: Pes štěká.
    mojeKocka.zvuk();    // Výstup: Kočka mňouká.
  }
}
```

I když všechny objekty mají typ **Zviratko**, při volání metody **zvuk()** se použije implementace konkrétní třídy objektu (v tomto případě **Pes** nebo **Kocka**). To je příklad polymorfismu.

---

**3. Abstraktní třídy a rozhraní (Interfaces)**

Abstraktní třídy a rozhraní slouží k tomu, aby určovaly základní strukturu pro ostatní třídy, ale samotné nemohou být **instanciovány**. Tato vlastnost je užitečná pro definování **obecných pravidel** pro různé třídy.

- **Abstraktní třída**:

```
abstract class Zviratko {
  abstract void zvuk();  // Abstraktní metoda (žádná implementace)

  void spinkej() {
    System.out.println("Zvířátko spí.");
```

```
    }
}

class Pes extends Zviratko {
    @Override
    void zvuk() {
        System.out.println("Pes štěká.");
    }
}

public class Main {
    public static void main(String[] args) {
        Zviratko pes = new Pes();
        pes.zvuk();  // Výstup: Pes štěká.
        pes.spinkej();  // Výstup: Zvířátko spí.
    }
}
```

Abstraktní třída **Zviratko** má abstraktní metodu **zvuk()**, kterou musí všechny podtřídy implementovat.

- **Rozhraní (Interface)**:

```
interface Zvuk {
    void zvuk();  // Všechna rozhraní mají implicitně veřejné a abstraktní metody
}

class Pes implements Zvuk {
    @Override
    public void zvuk() {
        System.out.println("Pes štěká.");
```

```
    }
}


public class Main {
    public static void main(String[] args) {
        Pes pes = new Pes();
        pes.zvuk();  // Výstup: Pes štěká.
    }
}
```

Třída **Pes** implementuje rozhraní **Zvuk** a poskytuje konkrétní implementaci metody **zvuk()**.

---

**4. Výjimky (Exceptions)**

Ve většině programů se občas stane něco, co nelze předem předvídat (např. dělení nulou, přístup k neexistujícímu souboru). **Výjimky** umožňují bezpečně zachytit a ošetřit tyto nečekané situace.

- **Základní syntaxe pro výjimky**:

```
public class Main {
    public static void main(String[] args) {
        try {
            int a = 10;
            int b = 0;
            int c = a / b;  // Tento kód způsobí výjimku
        } catch (ArithmeticException e) {
            System.out.println("Chyba: dělení nulou!");
        } finally {
            System.out.println("Tento blok se vykoná vždy.");
        }
    }
```

```
}
```

- **try-catch**: Pokusí se provést kód v bloku try. Pokud dojde k výjimce, je zachycena v bloku catch.
- **finally**: Tento blok se vždy vykoná, bez ohledu na to, zda došlo k výjimce nebo ne.

---

**5. Kolekce (Collections)**

V Javě existují **kolekce**, které nám umožňují efektivně manipulovat s **seznamy**, **sady** a **mapami**. K nejběžnějším kolekcím patří **ArrayList**, **HashSet**, **HashMap**.

- **ArrayList**:

```
import java.util.ArrayList;


public class Main {
   public static void main(String[] args) {
      ArrayList<String> seznam = new ArrayList<>();
      seznam.add("Petr");
      seznam.add("Eva");
      seznam.add("Jan");

      for (String jmeno : seznam) {
         System.out.println(jmeno);
      }
   }
}
```

**ArrayList** je dynamický seznam, který umožňuje snadné přidávání, mazání a přístup k prvkům.

- **HashSet** (neuchovává duplicitní hodnoty):

```
import java.util.HashSet;


public class Main {
```

```
  public static void main(String[] args) {
    HashSet<String> set = new HashSet<>();
    set.add("Petr");
    set.add("Eva");
    set.add("Petr");  // Tento prvek nebude přidán, protože je duplikát

    for (String jmeno : set) {
      System.out.println(jmeno);  // Výstup: Petr, Eva
    }
  }
}
```

- **HashMap** (uchovává páry klíč-hodnota):

```
import java.util.HashMap;

public class Main {
  public static void main(String[] args) {
    HashMap<String, Integer> map = new HashMap<>();
    map.put("Petr", 25);
    map.put("Eva", 30);

    System.out.println("Věk Petra: " + map.get("Petr"));  // Výstup: Věk Petra: 25
  }
}
```

Pokračujme tedy dál. Teď se podíváme na **další důležité** koncepty v Javě, které se hodí při **maturitních úlohách** a při hlubším porozumění jazyku:

**6. Práce s textovými řetězci (Strings)**

Manipulace s textovými řetězci je v Javě jedním z nejběžnějších úkolů, ať už jde o **porovnávání**, **hledání podřetězců**, **nahrazování** nebo **rozdělování**.

- **Konstrukce textového řetězce**:

```
String jmeno = "Petr";  // Vytvoření řetězce

String prijmeni = "Novak";

String celeJmeno = jmeno + " " + prijmeni;  // Spojení řetězců

System.out.println(celeJmeno);  // Výstup: Petr Novak
```

- **Porovnání řetězců**:

```
String str1 = "ahoj";

String str2 = "ahoj";

String str3 = "Ahoj";


System.out.println(str1.equals(str2));  // Výstup: true (porovnává hodnoty)

System.out.println(str1.equalsIgnoreCase(str3));  // Výstup: true (ignoruje velikost písmen)
```

- **Získání délky řetězce**:

```
String text = "Hello";

System.out.println(text.length());  // Výstup: 5
```

- **Vyhledávání podřetězce**:

```
String text = "Java je skvělá!";

System.out.println(text.contains("skvělá"));  // Výstup: true

System.out.println(text.indexOf("je"));  // Výstup: 5 (index první pozice výskytu)
```

- **Nahrazování částí textu**:

```
String text = "Jsem student";

String novyText = text.replace("student", "programátor");

System.out.println(novyText);  // Výstup: Jsem programátor
```

- **Rozdělení řetězce**:

```
String text = "jablko,banán,hruška";

String[] ovoce = text.split(",");
```

```
for (String ovoce : ovoce) {
   System.out.println(ovoce);
}
// Výstup:
// jablko
// banán
// hruška
```

---

**7. Práce s kolekcemi (Collections Framework)**

V Javě máme různorodé kolekce pro práci s daty. Tato část je extrémně důležitá, protože je součástí standardní knihovny pro práci s **seznamy**, **sety** a **mapami**.

- **ArrayList**:

```
import java.util.ArrayList;


public class Main {
   public static void main(String[] args) {
      ArrayList<String> seznam = new ArrayList<>();
      seznam.add("Petr");
      seznam.add("Eva");
      seznam.add("Jan");

      seznam.remove(1);  // Odstraní položku na indexu 1 (Eva)
      seznam.set(1, "Martin");  // Změní hodnotu na indexu 1 na "Martin"

      for (String jmeno : seznam) {
         System.out.println(jmeno);  // Výstup: Petr, Martin, Jan
      }
   }
```

```
}
```

- **LinkedList** (seznam, který je efektivnější pro vložení/odebrání na začátku seznamu):

```
import java.util.LinkedList;


public class Main {
   public static void main(String[] args) {
      LinkedList<String> linkedList = new LinkedList<>();
      linkedList.add("Petr");
      linkedList.add("Eva");

      linkedList.addFirst("Martin");  // Přidá na začátek
      linkedList.addLast("Jan");  // Přidá na konec

      for (String jmeno : linkedList) {
         System.out.println(jmeno);  // Výstup: Martin, Petr, Eva, Jan
      }
   }
}
```

- **HashSet** (kolekce, která neumožňuje duplicity):

```
import java.util.HashSet;


public class Main {
   public static void main(String[] args) {
      HashSet<String> set = new HashSet<>();
      set.add("Petr");
      set.add("Eva");
      set.add("Petr");  // Duplicitní hodnota nebude přidána
```

```
        for (String jmeno : set) {
            System.out.println(jmeno);  // Výstup: Petr, Eva
        }
    }
}
```

- **HashMap** (kolekce pro uchovávání dvojic klíč-hodnota):

```
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Petr", 25);
        map.put("Eva", 30);
        map.put("Jan", 35);

        System.out.println(map.get("Petr"));  // Výstup: 25
    }
}
```

---

**8. Práce s daty a soubory**

Java umožňuje práci se soubory a základními vstupy/výstupy (I/O).

- **Čtení ze souboru**:

```
import java.io.*;

public class Main {
    public static void main(String[] args) {
        try {
```

```
        FileReader fr = new FileReader("soubor.txt");
        BufferedReader br = new BufferedReader(fr);
        String radek;
        while ((radek = br.readLine()) != null) {
            System.out.println(radek);
        }
        br.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

- **Zápis do souboru**:

```
import java.io.*;


public class Main {
  public static void main(String[] args) {
    try {
      FileWriter fw = new FileWriter("soubor.txt");
      BufferedWriter bw = new BufferedWriter(fw);
      bw.write("Ahoj světe!");
      bw.newLine();
      bw.write("Toto je soubor.");
      bw.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
```

```
}
```

---

**9. Základní principy práce s vlákny (Threads)**

Vlákna umožňují běh více procesů zároveň, což je užitečné pro provádění úloh na pozadí.

- **Vytvoření a spuštění vlákna**:

```
class MojeVlakenko extends Thread {
    @Override
    public void run() {
        System.out.println("Vlákno běží.");
    }
}

public class Main {
    public static void main(String[] args) {
        MojeVlakenko mojeVlakenko = new MojeVlakenko();
        mojeVlakenko.start();  // Spustí nové vlákno
    }
}
```

- **Synchronized**: Pomocí tohoto klíče můžete **zajistit, že jeden proces na vlákno nebude kolidovat s jiným**.

```
class Banka {
    private int stav = 100;

    synchronized void vyber(int castka) {
        if (stav >= castka) {
            stav -= castka;
            System.out.println("Vybráno: " + castka);
        } else {
```

```
            System.out.println("Nedostatek prostředků.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Banka banka = new Banka();

        Thread t1 = new Thread(() -> banka.vyber(50));
        Thread t2 = new Thread(() -> banka.vyber(70));

        t1.start();
        t2.start();
    }
}
```

---