Elméleti ZH. kiegészítés

1. Java kód fordításának lépései

1. **Írás:** A Java kódot .java kiterjesztésű fájlban írjuk meg.
2. **Fordítás:** A `javac` parancs segítségével lefordítjuk a kódot bájtkódra (.class fájl).
3. **Futtatás:** A `java` parancs segítségével a JVM (Java Virtual Machine) végrehajtja a .class fájlt.

---

2. JVM: mi az, miért fontos?

- **Mi az?** A Java Virtual Machine egy futtatókörnyezet, amely a Java programokat bájtkódból futtatható kóddá alakítja a gépen.
- A Java platform alapja, különböző hardveralapú platformokra is átültették.
- Elkülöníti a programot a hardvertől
- **Miért fontos?**
  - Platformfüggetlenséget biztosít, mivel a bájtkód minden JVM-en ugyanúgy fut.
  - Automatizált memóriakezelést (Garbage Collection) végez.

---

3. JRE, JDK, IDE: mire lehet használni?

- **JRE (Java Runtime Environment):** A Java programok futtatásához szükséges környezet. Tartalmazza a JVM-et és a Java könyvtárakat.
- **JDK (Java Development Kit):** A Java programok fejlesztéséhez és futtatásához szükséges eszközkészlet. Tartalmazza a JRE-t, fordítót, és más fejlesztési eszközöket.
- **IDE (Integrated Development Environment):** Olyan eszköz, mint az IntelliJ IDEA vagy Eclipse, amely segíti a Java programok fejlesztését kódszerkesztő, hibakereső, és más funkciók biztosításával.

---

4. main metódus: hogy néz ki, miért fontos, mi a szerepe?

- **Hogy néz ki?**

```
public static void main(String[] args) {
    // A program belépési pontja
}
```

- Java-ban minden applikációnak tartalmaznia kell egy main metódust
- **Miért fontos?** Ez a program belépési pontja, minden más a program által szükséges metódus innen lesz meghívva
- **Szerepe:** A program futásának elindítása, a kezdő logika meghatározása.
- A main metódus egyetlen argumentumot fogad el: egy String tömböt, ezen keresztül kap információt a program a futtató rendszertől

---

5. Primitív adattípusok és wrapper osztályaik

Primitív típus	Wrapper osztály	Méret	Tartomány
`byte`	`Byte`	8 bit	-128 és 127 között
`short`	`Short`	16 bit	-32,768 és 32,767 között
`int`	`Integer`	32 bit	-2^{31} és $2^{31}-1$ között
`long`	`Long`	64 bit	-2^{63} és $2^{63}-1$ között

`float`	`Float`	32 bit	1.4E-45 és 3.4E+38 között
`double`	`Double`	64 bit	4.9E-324 és 1.7E+308 között
`char`	`Character`	16 bit	Unicode karakterek (0–65535)
`boolean`	`Boolean`	1 byte	`true` vagy `false`

6. Az objektum és osztály definíciója

- **Objektum:** Egy osztály példánya, amely tartalmazza az adott osztály attribútumainak (állapot) és metódusainak (viselkedés) megvalósítását. Például: egy „Autó” osztályból létrehozott konkrét „piros autó”.
- **Osztály:** Egy sablon vagy minta, amely meghatározza az objektumok attribútumait (adattagok) és viselkedését (metódusok).

7. Absztrakció fogalma

- Az absztrakció az a folyamat, amikor egy rendszer bonyolultságát csökkentjük azáltal, hogy csak a **lényeges tulajdonságokat emeljük ki**, és elrejtjük a részleteket.
- **Példa:** Egy autó vezetéséhez nem kell tudni, hogyan működik a motor, csak azt, hogyan kell használni a kormányt és a pedálokat.

8. Mit jelent az egységbezárás (encapsulation)?

- Az egységbezárás az objektumok adatainak és metódusainak védelme oly módon, hogy azok csak az osztály által meghatározott módokon legyenek elérhetők. Az osztály vagy objektum felhasználója nem ismeri a metódusok implementációját.
- **Java-ban megvalósítás:**
  - Adattagokat `private` kulcsszóval zárunk el.
  - Getterekkel és setterekkel biztosítjuk a hozzáférést.

9. Mi a különbség az objektum állapota és viselkedése között?

- **Állapot:** Az objektum adattagjai által tárolt értékek. Példa: egy „Autó” objektum színe, sebessége, gyártási éve.
- **Viselkedés:** Az objektum által végrehajtott műveletek (metódusok). Példa: egy „Autó” gyorsítása vagy fékezése.

10. Mi az osztályattribútum és osztálymetódus?

- **Osztályattribútum:** Olyan adattag, amely az osztályhoz kötődik, és minden példány számára közös, static kulcsszóval jelöljük.
  - **Példa:** `public static int counter;`
- **Osztálymetódus:** Olyan metódus, amely az osztályhoz tartozik, nem egy adott példányhoz, static kulcsszóval jelöljük.
  - **Példa:** `public static void printCounter();`

11. Mi az a getter és setter?

- **Adatelrejtés:** legtöbb esetben nem engedjük, hogy egy másik osztály közvetlenül módosítsa az adott osztály attribútumait. Emiatt getter-t és setter-r használunk, amelyek ellenőrzött módon érik el az objektum adatait.
- **Getter:** Visszaadja egy privát adattag értékét. (másik neve: **accessor**)
- **Setter:** Beállítja egy privát adattag értékét. (másik neve: **mutator**)
- **Szerepük:** Az egységbezárás megvalósítása és az adattagok biztonságos elérése.

12. Mi az öröklés (általánosan)?

- Két osztály közötti **kapcsolat**
- Az öröklődés egy olyan kapcsolattípus, amely lehetővé teszi egy osztály (leszármazott osztály) számára, hogy **örökölje** egy másik osztály (szülő osztály) **tulajdonságait** és **metódusait**.
- Az öröklődés segítségével létrehozhatunk egy **hierarchiát** az osztályok között, ahol a leszármazott osztályok az ősosztály tulajdonságait és viselkedését megöröklik.
- Az öröklődés lehetővé teszi az osztályok közötti újrafelhasználhatóságot és a kód strukturáltabbá tételét.

---

13. Mi az aggregáció? Mi a kompozíció? Mi az asszociáció?

- **Asszociáció:** Osztályok közötti kapcsolatot jelöl, amikor két osztály között van valamilyen kapcsolat vagy kapcsolódás. Az asszociáció általában kétirányú kapcsolat, és segít az osztályok közötti információk megosztásában és együttműködésben.
  - **Példa:** Egy „Tanár" tanít „Diákokat".
- **Aggregáció: E**gy speciális típusú asszociáció, amikor egy osztály tartalmazza egy másik osztály objektumait.
  - **Példa:** Egy „Osztály" tartalmaz „Diákokat", de a diákok önállóan is létezhetnek.
- **Kompozíció:** Erős kapcsolat, ahol egy osztály példányosítja és tartalmazza egy másik osztály objektumait.
  - **Példa:** Egy „Név" nem kell, hogy létezzen „Személy" nélkül.
- A kompozíció és az aggregáció asszociáció, de asszociáció lehet más kapcsolat is

---

14. Mi az absztrakt osztály?

- Egy olyan osztály, amelyet nem lehet példányosítani, csak leszármaztatni.
- Tartalmazhat absztrakt (implementáció nélküli) és konkrét metódusokat is.

```
abstract class Shape {
    abstract void draw(); // absztrakt metódus
}
```

---

15. Népszerű objektumorientált programozási nyelvek

1. Java
2. C++
3. Python
4. C#

---

16. Névkonvenciók Java-ban

- **Osztályok:** PascalCase formátum, pl. `MyClass`.
- **Adattagok és metódusok:** camelCase formátum, pl. `myVariable`, `myMethod()`.
- **Konstansok:** Nagybetűk és aláhúzásjelek, pl. `MAX_VALUE`.
- **Paraméterek:** camelCase formátum, pl. `parameterName`.

---

17. Mi az a konstruktor? Mi történik, ha nem adunk meg konstruktort?

- **Konstruktor:** Olyan speciális metódus, amelyet egy osztály példányosításakor hív meg a JVM.
- A konstruktor hozza létre az objektumokat.
- Lehet több konstruktor is → Konstruktor túlterhelés.

- **Szintaxis:** Az osztály nevét viseli, nincs visszatérési típusa.

```
public MyClass() {
    // Inicializálás
}
```

- **Ha nincs megadva:** A Java fordító egy alapértelmezett üres konstruktort generál.

---

18. Hogyan példányosítunk Java-ban egy osztályt?

```
MyClass obj = new MyClass(); // Paraméter nélküli konstruktor
Circle myCircle = new myCircle(5); // Ha paramétere van a konstruktornak
```

---

19. Java Garbage Collector mit csinál?

- A Garbage Collector (GC) automatikusan felszabadítja azokat az objektumokat a memóriából, amelyekre már nincs hivatkozás.
- Automatikusan és időszakosan működik.
- **Fontossága:** Megakadályozza a memória túlcsordulását és optimalizálja az erőforrás-kezelést.

---

20. Osztály tagjainak és metódusainak láthatósági módosítói

- **public:** Az osztály eleme mindenki számára látható.
- **private:** Csak az osztályon belül látható.
- **módosító nélkül (package-private):** Csak az osztályon belül, és a csomagon belül látható.
- **protected:** Az osztály, a csomag, és az alosztályok látják.

---

21. A `static` kulcsszó használata Java-ban

- **Adattagoknál:** Az osztály szintjén osztozik minden példány.
- **Metódusoknál:** Az osztályhoz tartoznak, és nem egy-egy példányhoz. Akkor is meghívható, ha nincs példánya az osztálynak.

```
myClass.attribute;
myClass.method();
```

- **Konstansoknál:** Konstansok deklarálására is használható:

```
static final double PI = 3.1415;
```

  - A final kulcsszó jelentése: A mező nem változtathatja az értékét.
  - A konstansok neve a konvenció szerint végig nagybetűs, ha több szóból áll, akkor aláhúzás jellel (_) választjuk el a szavakat.

---

22. Hogyan deklarálunk konstanst? Névkonvenció?

- Konstansok deklarálása a **static** és a **final** kulcsszóval történik:
  - A **final** kulcsszó jelentése: A mező nem változtathatja az értékét.
  - A konstansok neve a **konvenció** szerint végig **nagybetűs**, ha több szóból áll, akkor **aláhúzás** jellel (_) választjuk el a szavakat.

```
public static final int MAX_VALUE = 100;
```

23. Mit csinál a `final` kulcsszó?

- **Osztályok:** Megakadályozza az osztály öröklését, nem lehet alosztálya.
- **Metódusok:** Nem lehet felülírni. Ha egy metódus implementációja nem szabad változzon és kritikus fontosságú, akkor előnyös final kulcsszóval megjelölni azt.
- **Adattagok:** Nem lehet újra inicializálni.

24. Tömb (array) deklarációja és használata

- **Deklaráció:**

```
int[] numbers = new int[5];
int[] numbers2 = {11, 22, 33, 44, 55};
```

- **Használat:**

```
numbers[0] = 10; // Értékadás
System.out.println(numbers[0]); // Érték kiíratása
System.out.println(numbers2.length) // Méret kiíratása
```

25. Mi az a Java csomag? Hogyan adunk neki nevet?

- **Csomag:** Olyan névtér, amely osztályokat és interfészeket csoportosít.
- A neve az Internet domain név fordítottja.
- Példa:

```
package hu.unideb.inf.mypackage;
```

26. Csomag elemeinek láthatósági módosítói

- **public:** Bárhonnan elérhető.
- **Alapértelmezett (package-private):** Csak ugyanabban a csomagban elérhető.

27. Hogyan lehet használni a csomag elemeit?

- **Teljes név használata:**

```
mypackage.MyClass obj = new mypackage.MyClass();
```

- **Csomagelem importálása:**

```
import mypackage.MyClass;
```

- **Egész csomag importálása:**

```
import mypackage.*;
```

28. Népszerű Java API csomagok

1. `java.util`
2. `java.io`
3. `java.net`
4. `java.sql`

---

29. Mi az az annotáció?

- Kiegészítő információt biztosító metaadatok a kódban.
- Nincsenek közvetlenül kihatással a kódra, amit annotálnak.
- Formátum: @Entity
- A @ karakter jelzi a compiler-nek, hogy annotáció következik.

```
@Override
public void mySuperMethod() {}
```

---

30. Mit csinál az `@Override` annotáció?

- Jelzi a compiler számára, hogy a metódus felülír egy szülőosztályban lévő metódust.

---

31. Mit csinál a `this` kulcsszó?

- Az aktuális objektumra hivatkozik.
- Az aktuális objektum bármely tagjára hivatkozhatunk a segítségével.
- Egy ugyanabban az osztályban lévő, másik konstruktor meghívására is használható, ezt explicit konstruktorhívásnak nevezik.

---

32. Mit csinál a `super` kulcsszó?

- A szülőosztályra hivatkozik.
- Ha egy metódus felülír egy szülőosztálybeli metódust, akkor a super kulcsszóval meghívhatjuk a szülőosztály metódusát.
- A szülőosztály konstruktorát is hívhatjuk vele.
- Leggyakrabban arra használt, hogy egyértelműsítsük az alosztályban és szülőosztályban lévő azonos nevű metódusok közül melyiket szeretnénk használni.
- Rejtett adattagok elérésére is használható.

---

33. Miért fontos a Java Object osztálya?

- Az osztályhierarchia gyökere.
- Minden osztálynak szuperosztálya az Object osztály.
- Minden objektum (beleértve a tömböket is) implementálja az Object osztály metóduait.
- Alapmetódusokat biztosít, pl. `toString()`, `equals()`, `hashCode()`.

---

34. Mi a baj a String osztállyal? Mit és hogyan használunk helyette?

- **String osztály problémája:**
  - A `String` **immutábilis (immutable)**, ami azt jelenti, hogy minden módosítás új objektumot hoz létre. Ez memória- és teljesítményproblémákat okozhat.
- **Alternatíva:**

- `StringBuilder`: Nem szálbiztos, de gyorsabb.
- `StringBuffer`: Szálbiztos, de lassabb.

- **Használat `StringBuilder`rel:**

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb.toString());
```

---

35. Numerikus típusok közötti konverzió Java-ban

- **Automatikus típuskonverzió (widening):**

  Kisebb típus automatikusan konvertálódik nagyobbra.

```
int i = 10;
double d = i; // Automatikus konverzió
```

- **Kényszerített típuskonverzió (casting):**

  Nagyobb típusból kisebbbe kényszerítéssel.

```
double d = 10.5;
int i = (int) d; // Kényszerített konverzió
```

---

36. Hogyan konvertálunk számot Stringgé és vissza?

- **Számból String:**

```
int num = 123;
String str = String.valueOf(num);
```

- **Stringből szám:**

```
String str = "123";
int num = Integer.parseInt(str);
```

---

37. Java paraméterátadás módja

- **Érték szerint:** A Java mindig érték szerint adja át a paramétereket.
  - **Primitív típusok:** A paraméter értéke másolódik.
  - **Objektumok:** Az objektum referencia másolódik, de az objektum állapota módosítható, ha van megfelelő hozzáférési szint.
- Az érték szerinti paraméterátadás esetén a **formális paramétereknek van címkomponensük** a hívott alprogram területén.
- A formális paraméter kap egy kezdőértéket, és az alprogram ezzel az értékkel dolgozik a saját területén.

---

38. Hogyan lehet tetszőleges számú paramétert átadni egy metódusnak?

- **Varargs (változó hosszúságú paraméterlista)** használatával lehetséges.
- Hasznos akkor, amikor nem tudhatjuk előre, hogy hány bizonyos típusú argumentum lesz átadva a metódusnak.
- **Használat:**

```
public void printNumbers(int... numbers) {
    for (int num : numbers) {
        System.out.println(num);
    }
}
```

- **Hívás:**

```
printNumbers(1, 2, 3, 4, 5);
```

---

39. Mit csinál Java-ban a `return` utasítás?

- A return utasítással visszatér a metódus a kódhoz, ami meghívta, és visszaadja az értéket a hívónak.
- Minden metódus, ami nem void típusú, tartalmazzon kell return utasítást.
- **Használat:**

```
public int add(int a, int b) {
    return a + b;
}
```

---

40. Inicializáló mező és blokk Java-ban

- **Inicializáló mező:** Adattagok deklaráláskor történő inicializálása.

```
int x = 10;
```

- **Inicializáló blokk:** Az összes példányosításkor lefutó kód.

```
{
    System.out.println("Inicializáló blokk fut");
}
```

---

41. Java Enum típus

- Speciális adattípus, ami lehetővé teszi egy változó számára, hogy egy előre definiált konstansoknak a halmaza legyen.
- Az értéke egyike kell legyen az előre definiált konstansoknak.

```
public enum Day {
    MONDAY, TUESDAY, WEDNESDAY
}
```

- **Használat:**

```
Day today = Day.MONDAY;
System.out.println(today);
```

---

42. Java osztályok közötti öröklés

- Az öröklés az objektumorientált programozás egyik alapelve, amely lehetővé teszi, hogy egy osztály (`gyerekosztály`) örökölje egy másik osztály (`szülőosztály`) attribútumait és metódusait.

- Az Object osztályon kívül minden osztálynak csak egy közvetlen szülőosztálya van, ha nem adunk meg explicit módon szülőosztályt, akkor az Object lesz a közvetlen szülőosztálya.
- Az alosztály/gyerekosztály örökli a szülőosztály minden tagját (adattagok, metódusok és beágyazott osztályok).
- A konstruktorok nem öröklődnek, de meghívható a szülőosztály konstruktora.
- Az alosztály nem örökli a szülőosztály privát tagjait, de ha a szülőosztályban van public vagy protected metódus a privát adattagok elérésére, akkor az alosztály által is használhatja ezeket.
- **Célja:** Kód újrafelhasználás, strukturált programozás.
- **Öröklés szintaxisa:**

```
public class Parent {
    // Szülőosztály tartalma
}

public class Child extends Parent {
    // Gyermekosztály tartalma
}
```

43. Konstruktorok öröklődése öröklés esetén

- A konstruktorok nem öröklődnek.
- A szülőosztály konstruktorának hívása kell legyen az első sor a gyerekosztály konstruktorában.
- Szülő konstruktor hívásának szintaxisa:
  - super();
  - super(paraméter lista);
- Ha a gyerekosztály nem hívja meg explicit módon a szülő konstruktorát, akkor a Java compiler automatikusan meghívja a szülőosztály argumentumok nélküli konstruktorát.

44. Metódus felülírás öröklés esetén

- A gyermekosztály újradefiniálhatja a szülőosztály metódusát.
- A @Override annotáció jelzi a compiler-nek, hogy felül szeretnénk írni egy metódust
- **Szabályok:**
  - Azonos method signature.
  - Nem lehet szigorúbb láthatóságú.

```
@Override
public void parentMethod() {
    // Új implementáció
}
```

45. Ugyanolyan nevű statikus metódus a szuperosztályban és alosztályban

- Ha azonos method signature-rel rendelkező statikus metódus van az alosztályban és a szuperosztályban is, akkor az alosztály elrejti a szuperosztályban lévőt.
- Meghívható a szuperosztály és az alosztály statikus metódusa is.

46. Ugyanolyan nevű adattag a szuper- és alosztályban

- Ha egy alosztályban szerepel egy ugyanolyan nevű adattag, mint a szuperosztályban, akkor az alosztály elrejti a szuperosztályban lévőt, még akkor is, ha különböző típusúak.
- Szuperosztály adattagjának elérése a super kulcsszóval történik:

```
super.attributeName;
```

---

47. Konstruktorok lánca (Chain of Constructors)

- Ha egy konstruktor explicit vagy implicit módon meghívja a szuperosztály konstruktorát, konstuktorok láncáról beszélünk, ez visszamegy egészen az Object osztályig.
- Figyelembe kell venni, főleg ha hosszú az öröklési lista.

---

48. Láthatósági módosítók változása metódus felülíráskor

- Szigorúbb láthatóság nem megengedett, de kevésbé szigorú igen.
- Példa: A szülőosztály `protected` metódusa csak `protected` vagy `public` lehet az alosztályban.

---

49. Polimorfizmus Java-ban

- Egy osztály alosztályai definiálhatnak saját viselkedést, de mindeközben osztozhatnak a szülőosztály funkcionalitásának részein.
- Egy metódushívás (ugyanaz a név, és paraméterlista) más-más osztályokon **másképp viselkedik**.
- Pl. öröklésnél: Kacsa, Kecske, és Kutya osztályok mint az Állat osztály leszármazottjai, és mind a hárman másképpen valósítják meg a beszél metódust
- Minden alosztályban **másképp van implementálva**.

---

50. Absztrakt osztály és absztrakt metódus

- **Absztrakt osztály:** Nem példányosítható, tartalmazhat absztrakt és konkrét metódusokat.
- **Absztrakt metódus:** Olyan metódus, amelyet implementáció nélkül deklarálunk.

```
public abstract class Shape {
    public abstract void draw();
}
```

---

51. Az alkalmazásfejlesztés életciklusának lépései

1. Vízió
2. Követelmények feltárása
3. Elemzés
4. Architekturális tervezés
5. Tervezés
6. Implementálás
7. Tesztelés
8. Üzembe helyezés
9. Üzemeltetés
10. Karbantartás
11. Evolúció

12. Üzemen kívül helyezés

52. Mi az az UML?

- Unified Modeling Language (Egységesített Modellező Nyelv).
- Az objektum-orientált szemléletre épülő elemzés és tervezés eszköze.
- Szabványos jelölésrendszer.
- Alapvetően grafikus nyelv.

53. Hogy néz ki az UML osztálydiagram?

- **Osztályok:** Téglalapok, amelyek tartalmazzák az osztály nevét, attribútumait és metódusait.
  - **Statikus tagok és metódusok:** Aláhúzással jelölve
  - **Láthatósági attribútumok:**
    - - private
    - + public
    - # protected
    - ~ package (az attribútum ugyanazon csomag minden eleme számára látható)
    - default → semmi
- Tartalmazhat még más dobozokat is:
  - Responsibilities (felelősségek) (az osztály célja)
  - Exceptions (kivételek)
  - Signals (jelzések) (olyan üzenetek, amelyeket az osztály a normál működése közben is tud fogadni)
- **Kapcsolatok:**
  - Asszosziáció: Vonal két osztály között, vagy rombusz több osztály között
  - Aggregáció: Üres rombuszban végződő vonal
  - Kompozíció: Tele rombuszban végződő vonal

54. Az UML hogyan jelöli az osztályok és interfészek közötti öröklést?

- Az **osztály és interfész közötti öröklés** egy üres háromszögben végződő szaggatott vonallal van jelölve.
- Egy osztály vagy interfész több interfészt is örökölhet.

55. Az UML hogyan jelöli az absztrakt osztályt és interfészt?

- **Absztrakt osztály:** Az osztály neve dőlt betűvel szerepel.
- **Interfész:** Téglalapban a <<interface>> kulcsszóval van megjelölve.

56. Mi az a Java interfész? Mit tartalmazhat?

- Egy referencia típus, amely hasonlít az osztályokhoz, és tartalmazhat:
  - Absztrakt metódusokat,
  - Default metódusokat,
  - Statikus metódusokat,
  - Konstansokat, és
  - Beágyazott típusokat tartalmazhat.

- Interfészeket nem lehet példányosítani, csak implementálhatja egy másik osztály, vagy örökölheti egy másik interfész.

---

57. Mire szolgál a Java interfész default metódusa?

- Lehetővé teszi, hogy az interfész új metódusokat vezessen be anélkül, hogy a meglévő implementáló osztályokat módosítani kellene.

---

58. Java interfészek közötti öröklés, interfész és osztály közötti öröklés

- **Interfészek közötti öröklés:** Egy interfész egy vagy több másik interfészt örökölhet.
- Az extends kulcsszó használatával történik.

```
public interface A { }
public interface B extends A { }
```

- **Interfész és osztály közötti öröklés:** Egy osztály implementálhat egy vagy több interfészt.
- Az implements kulcsszóval történik.

```
public class MyClass implements A, B { }
```

---

59. Hogyan lehet használni egy Java interfészt?

- Amikor egy új interfészt definiálunk, akkor egy új referencia adattípust hozunk létre.
- Az interfész neveit bárhol használhatjuk, ahol bármilyen más adattípus nevét is használhatnánk.
- Ha egy referencia változót definiálunk, amelynek típusa egy interfész, akkor az ahhoz rendelt objektumnak egy olyan osztály példányának kell legyen, amely implementálja az interfészt.
- Egy osztálynak implementálnia kell az interfész összes absztrakt metódusát.
- Szintaxis:

```
public interface A {
    void method();
}

public class MyClass implements A {
    public void method() {
        System.out.println("Implemented method");
    }
}
```

---

60. Default metódusok konfliktusának feloldása interfészek öröklése során

- Ha két interfész azonos nevű default metódust tartalmaz, az implementáló osztályban konfliktusba kerülnek a default metódusok.
- Ilyenkor a Java compiler egy compiler error-t ad.
- Explicit módon felül kell írni a szupertípus metódusait.

---

61. Comparable interfész

- A Comparable interfész a compareTo metódust tartalmazza.
- A `compareTo` metódus összehasonlítja a hívó objektumot a megadott objektummal, és egy negatív egész számot, 0-t vagy egy pozitív egész számot ad vissza attól függően, hogy a hívó objektum kisebb, egyenlő

vagy nagyobb a megadott objektumnál.

- Ha a megadott objektum nem hasonlítható össze a hívó objektummal, a metódus `ClassCastException` kivételt dob.

```
public class MyClass implements Comparable<MyClass> {
    public int compareTo(MyClass other) {
        return this.value - other.value;
    }
}
```

62. Comparator interfész

- A Comparator interfész a compare metódust tartalmazza.
- A `compare` metódus összehasonlítja a két argumentumát, és egy negatív egész számot, 0-t, vagy egy pozitív egész számot ad vissza attól függően, hogy az első argumentum kisebb, egyenlő vagy nagyobb a másodiknál.

```
import java.util.Comparator;

public class MyComparator implements Comparator<MyClass> {
    public int compare(MyClass o1, MyClass o2) {
        return o1.value - o2.value;
    }
}
```

63. Mi az a Java generikus? Hogyan definiálhatunk generikusokat?

- A generikus típusok lehetővé teszik, hogy típusok (osztályok és interfészek) paraméterek legyenek, amikor osztályokat, interfészeket és metódusokat definiálunk.
- Hasonlóan a metódus deklarációkban használt formális paraméterekhez, a típusparaméterek lehetőséget adnak arra, hogy ugyanazt a kódot különböző bemenetekkel használjuk újra.
- A különbség az, hogy a formális paraméterek bemenetei értékek, míg a típusparaméterek bemenetei típusok.
- Egy generikus osztályt a következő formátumban definiálunk:
  - class Név<T1, T2, ..., Tn> { /* ... */ }
- Névkonvenció szerint a típusparaméterek neve egyetlen nagybetű.
- A leggyakrabban használt típusparaméter nevek:
  - E - Elem (a Java Collections Framework által széles körben használva)
  - K - Kulcs
  - N - Szám
  - T - Típus
  - V - Érték
  - S, U, V stb. - 2., 3., 4. típusok

```
public class GenericClass<T> {
    private T data;
    public T getData() { return data; }
    public void setData(T data) { this.data = data; }
}
```

64. Java generikus metódus, statikus metódus, és hívásuk

- A generikus metódusok olyan metódusok, amelyek saját típusparamétereket vezetnek be.
- Ez hasonló a generikus típus deklarálásához, de a típusparaméter hatóköre korlátozódik arra a metódusra, ahol deklarálják.
- Engedélyezettek statikus és nem statikus generikus metódusok, valamint generikus osztály konstruktorok is.
- A generikus metódus szintaxisa tartalmaz egy típusparaméter listát angle bracket-ek (<, >) között, amely a metódus visszatérési típusa előtt jelenik meg.
- Statikus generikus metódusok esetén a típusparaméter szakasznak a metódus visszatérési típusa előtt kell megjelennie.
- **Generikus metódus:**

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

Hívás:

```
String[] strArray = {"A", "B", "C"};
printArray(strArray);
```

- **Statikus generikus metódus:**

```
public static <T> T getFirstElement(T[] array) {
    return array[0];
}
```

---

65. Java generikus, bounded type

- Előfordulhat, hogy korlátozni szeretnénk azokat a típusokat, amelyeket típusargumentumként használhatunk egy paraméterezett típusban.
- Például egy metódus, amely számokkal dolgozik, csak a Number vagy annak leszármazott osztályainak példányait szeretné elfogadni.
- Erre szolgálnak a bounded type paraméterek.
- Egy bounded type paraméter deklarálásához soroljuk fel a típusparaméter nevét, majd az `extends` kulcsszót, majd annak felső határát, ami ebben az esetben a Number.
- Fontos megjegyezni, hogy ebben a kontextusban az `extends` általános értelemben használatos, jelentheti akár az "extends" (osztályok esetén), akár az "implements" (interfészek esetén) szót.

---

66. Java generikus, wildcards

- A generikus kódban a kérdőjel (?), amelyet wildcard-nak nevezünk, egy ismeretlen típust képvisel.
- A wildcard különféle helyzetekben használható: paraméter, mező vagy lokális változó típusaként; néha visszatérési típusaként is (bár jobb, ha konkrétabbak vagyunk).
- A wildcard-ot soha nem használják típusargumentumként generikus metódushívás, generikus osztály példányosítás vagy szuperosztály esetén.

---

67. Java generikus, type erasure

- Type erasure biztosítja, hogy nem jönnek létre új osztályok a paraméterezett típusokhoz, így a generikusok nem okoznak futási időbeli többletköltséget.
- A type erasure folyamat során a Java fordító törli az összes típusparamétert, és helyettesíti őket az első bound-dal, ha a típusparaméter bounded, vagy Object-tel, ha a típusparaméter unbounded.

---

68. Mi az a kollekció? Deklaráljon és példányosítson egyet.

- A collection egy olyan objektum, amely több elemet csoportosít egyetlen egységbe.
- A collection-öket arra használják, hogy adatokat tároljanak, lekérdezzenek, manipuláljanak és aggregált adatokat kommunikáljanak.
- Jellemzően olyan adatokat képviselnek, amelyek természetes csoportot alkotnak, mint például egy póker kéz (kártyák gyűjteménye), egy levelező mappa (levelek gyűjteménye), vagy egy telefonkönyv (nevek és telefonszámok leképezése).
- **Példa:**

```
List<String> list = new ArrayList<>();
list.add("Elem1");
list.add("Elem2");
```

---

69. A kollekció elemeinek a rendezését hogyan lehet megvalósítani?

- Collections.sort(list), Comparable interfész
- A `Collections.sort(list);` metódus használatakor, ha a lista elemei nem implementálják a `Comparable` interfészt, a `Collections.sort(list)` `ClassCastException` kivételt fog dobni.
- Collections.sort(list, comparator), Comparator interfész,
- Hasonlóképpen, ha a `Collections.sort(list, comparator)` metódust használjuk, és olyan lista elemeit próbáljuk rendezni, amelyek nem hasonlíthatók össze egymással a megadott comparator segítségével, akkor `ClassCastException` kivételt fog dobni.
- Azokat az elemeket, amelyek összehasonlíthatók egymással, kölcsönösen összehasonlíthatóknak nevezik.
- Bár különböző típusú elemek is lehetnek kölcsönösen összehasonlíthatók, az itt felsorolt osztályok egyike sem engedélyezi az osztályok közötti összehasonlítást.

---

70. Collections osztály

- Ez az osztály kizárólag statikus metódusokat tartalmaz, amelyek collection-ökön végeznek műveleteket vagy collection-öket adnak vissza.
- Tartalmaz polimorf algoritmusokat, amelyek collection-ökön működnek, "wrapper"-öket, amelyek egy új collection-t adnak vissza, amely egy meghatározott collection által van támogatva.
- Segédmetódusokat biztosít kollekciók kezelésére, például:
  - `sort(List list)` - rendezés
  - `reverse(List list)` - lista fordítása
  - `shuffle(List list)` - elemek keverése
  - `min(Collection coll)` - legkisebb elem

---

71. Rajzolja le a Java kollekciók interfészeinek és osztályainak öröklési rendszerét!

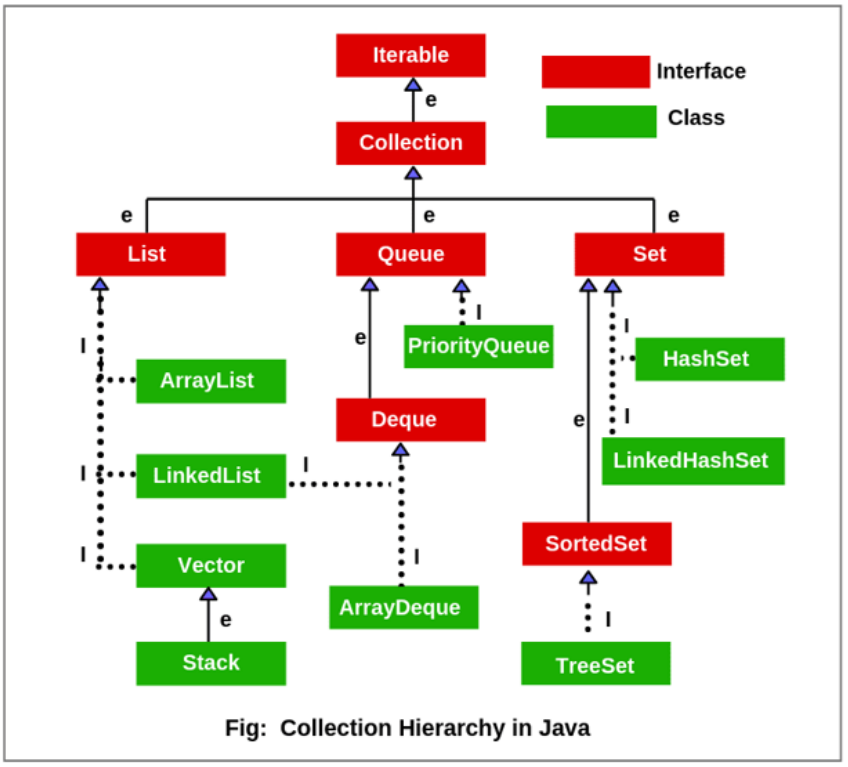

Fig: Collection Hierarchy in Java

e→ extends, I→ implements

---

72. Hogyan különbözteti meg a HashSet az elemeket egymástól?

- A `hashCode()` és `equals()` metódusok segítségével ellenőrzi az elemek egyediségét.
- Nem ad garanciát a set sorrendjére; különösen nem garantálja, hogy a sorrend idővel változatlan marad.

---

73. Hogyan különbözteti meg a TreeSet az elemeket egymástól?

- Az elemeket természetes sorrendjük vagy egyedi **Comparator** alapján rendezi.

---

74. Java beágyazott osztály

- Az egyik osztály tartalmazhat egy másik osztályt (beágyazott), azaz a beágyazott osztály egy másik osztályban van definiálva.
- A beágyazott osztályokat két kategóriába sorolják: nem statikus és statikus.
  - A nem statikus beágyazott osztályokat **inner class**-nak nevezik.
  - A statikusként deklarált beágyazott osztályokat **static nested class**-nak nevezik.

---

75. Statikus beágyazott osztály

- **Static nested class**
- A statikus beágyazott osztályok nem férnek hozzá a külső osztály egyéb tagjaihoz.
- Ahogyan az osztály metódusai és változói, egy statikus belső osztály is a külső osztályához van társítva.
- És hasonlóan a statikus osztálymetódusokhoz, a statikus belső osztály sem hivatkozhat közvetlenül a környező osztály adattagjaira vagy metódusaira: csak egy objektum referencia segítségével férhet hozzá azokhoz.

```
public class Outer {
    static class Nested {
        public void display() {
            System.out.println("Static nested class");
        }
    }
}
```

76. Java local class

- A lokális osztályok olyan osztályok, amelyeket egy blokkban definiálnak, amely egy vagy több állításból áll, zárójelekkel körülvéve. A lokális osztályok jellemzően egy metódus törzsében találhatók.
- A lokális osztály hozzáférhet a környező osztály tagjaihoz.
- A lokális osztály hozzáférhet a lokális változókhoz is, de csak azokhoz, amelyek final-ként vannak deklarálva.

77. Java anonymous class

- Az anonim osztályok lehetővé teszik, hogy a kódunk tömörebbé váljon.
- Lehetővé teszik, hogy egyszerre deklaráljuk és példányosítsuk az osztályt.
- Hasonlóak a lokális osztályokhoz, de nincs nevük.
- Az anonim osztályokat akkor használjuk, ha egy lokális osztályra csak egyszer van szükségünk.
- Az anonim osztályok kifejezések, ami azt jelenti, hogy az osztályt egy másik kifejezésen belül definiáljuk.
- Az anonim osztály kifejezés szintaxisa hasonlít egy konstruktor hívására, de az osztálydefiníció egy kódban lévő blokkban található.

78. Java Funkcionális interfész

- A funkcionális interfész olyan interfész, amely csak egy absztrakt metódust tartalmaz.
- A funkcionális interfész tartalmazhat egy vagy több `default` metódust vagy statikus metódust is.
- Mivel egy funkcionális interfész csak egy absztrakt metódust tartalmaz, elhagyhatjuk annak a metódusnak a nevét, amikor implementáljuk.
- Ehhez anonim osztály kifejezés helyett egy lambda kifejezést használunk.

79. Soroljon fel 5 beépített Java funkcionális interfészt

1. `Function<T, R>`
2. `Consumer<T>`
3. `Supplier<T>`
4. `Predicate<T>`
5. `BiFunction<T, U, R>`

80. Java lambda kifejezés

- A lambda kifejezések lehetővé teszik, hogy az egy metódusú osztályok példányait tömörebben fejezzük ki.

```
List<String> list = Arrays.asList("A", "B", "C");
list.forEach(item -> System.out.println(item));
```

81. Java metódus referencia

- Metódus referencia: Lambda kifejezéseket anonim metódusok létrehozására használjuk.
- Néha azonban egy lambda kifejezés nem csinál mást, mint hogy meghív egy létező metódust.
- Ilyen esetekben gyakran világosabb, ha közvetlenül a létező metódus nevét használjuk.
- A metódus referenciák lehetővé teszik ezt; kompakt, könnyen olvasható lambda kifejezések, amelyek már meglévő nevű metódusokat használnak.
- **Típusai:**
  1. **Statikus metódus referencia:**

```
Consumer<String> printer = System.out::println;
printer.accept("Hello, World!");
```

  2. **Példány metódus referencia:**

```
String str = "example";
Supplier<Integer> length = str::length;
System.out.println(length.get());
```

  3. **Konstruktor referencia:**

```
Supplier<List<String>> listSupplier = ArrayList::new;
List<String> list = listSupplier.get();
```

82. Mi az a kivétel?

- A kivétel egy olyan esemény, amely egy program futása közben következik be, és megszakítja az utasítások normál folyamatát.
- Amikor hiba történik egy metódusban, a metódus létrehoz egy objektumot, és átadja azt a futtató rendszernek.
- Ez az objektum, amelyet kivétel objektumnak (exception object) nevezünk, tartalmazza a hiba típusára és a program állapotára vonatkozó információkat abban a pillanatban, amikor a hiba bekövetkezett.
- Kivétel objektum létrehozása és átadása a futtató rendszernek kivétel dobása (throwing an exception) néven ismert.

83. Hogyan lehet kezelni a kivételt?

- A helyes Java kódnak tiszteletben kell tartania a „Catch or Specify” követelményt.
- Ez azt jelenti, hogy az olyan kódot, amely bizonyos kivételeket dobhat, az alábbiak egyikével kell körülvenni:
  - Egy `try` utasítás, amely elkapja a kivételt. A `try`nak biztosítania kell egy kezelőt a kivételhez.
  - Egy metódus, amely megadja, hogy dobhat kivételt. A metódusnak rendelkeznie kell egy `throws` záradékkal, amely felsorolja a kivételt.
- Kivételek kezelése a `try-catch` blokkal:

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Hiba: " + e.getMessage());
} finally {
    System.out.println("Ez mindig lefut.");
}
```

84. A kivételeknek mi a 3 alapvető kategóriája?

1. **Checked Exceptions:** Kötelezően kezelendők, például: `IOException`.
    a. Ezek olyan kivételes körülmények, amelyeket egy jól megírt alkalmazásnak előre kell látnia és kezelnie kell.
2. **Unchecked Exceptions:** Nem kötelezően kezelendők, például: `NullPointerException`.
    a. Ezek olyan kivételes körülmények, amelyek kívülről érkeznek az alkalmazásba, és amelyeket az alkalmazás általában nem tud előre látni vagy kezelni.
3. **Error:** Súlyos hibák, amelyeket nem lehet kezelni, például: `OutOfMemoryError`.
    a. Ezek olyan kivételes körülmények, amelyek belsőleg az alkalmazáson belül történnek, és amelyeket az alkalmazás általában nem tud előre látni vagy kezelni.

---

85. Hogy néz ki a try-catch-final utasítás, és melyik része mit csinál?

- `try` **blokk:** Kód, amely hibát okozhat.
- `catch` **blokk:** A kivétel kezelése.
- `finally` **blokk:** Mindig lefut, akár történt kivétel, akár nem.

```
try {
    int num = Integer.parseInt("abc");
} catch (NumberFormatException e) {
    System.out.println("Hiba történt!");
} finally {
    System.out.println("Ez mindig lefut.");
}
```

---

86. Mit csinál a try-with-resource utasítás?

- A try-with-resources utasítás automatikusan felszabadítja a rendszer erőforrásait, amikor már nincs rájuk szükség.
- A try-with-resources utasítás egy try utasítás, amely deklarál egy vagy több erőforrást.
    - Egy erőforrás olyan objektum, amelyet le kell zárni, miután a program befejezte a használatát.
- A try-with-resources utasítás biztosítja, hogy minden erőforrás az utasítás végén le legyen zárva.
- Bármely objektum, amely implementálja a java.lang.AutoCloseable-t (beleértve az összes java.io.Closeable-t is), használható erőforrásként.

---

87. Hogyan lehet kivételt dobni?

- Mielőtt elkapnánk egy kivételt, valahol valamilyen kódnak el kell dobnia azt.
- A kivételt mindig a `throw` utasítással dobjuk.
- A `throw` utasítás egyetlen argumentumot igényel: egy kivételt dobó objektumot.
- A kivételobjektumok a `Throwable` osztály bármelyik alosztályának példányai.
- Íme egy példa a `throw` utasításra:

```
throw someThrowableObject;
```

---

88. Hogyan hozhatunk létre Java kivétel osztályokat?

- A jól olvasható kód érdekében jó gyakorlat, hogy minden olyan osztály nevéhez, amely közvetlenül vagy közvetve öröklődik az `Exception` osztályból, hozzáfűzzük az `Exception` szót.

- Egyedi kivételosztály létrehozása:

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

89. Mi az I/O stream? Mi lehet a forrása és célhelye az I/O streameknek?

- Az I/O Stream bemeneti forrást vagy kimeneti célt képvisel.
- Forrása és célhelye lehet:
  - lemezfájlok
  - eszközök
  - más programok
  - memória tömbök.
- A streamek sokféle különböző adatot támogatnak, beleértve az egyszerű bájtokat, primitív adat típusokat, lokalizált karaktereket és objektumokat.
- Néhány stream egyszerűen továbbítja az adatokat; mások manipulálják és átalakítják az adatokat hasznos módokon.

90. Mi az a byte stream?

- A programok byte stream-eket használnak a 8-bites bájtok bemenetére és kimenetére.
- Számos byte stream osztály létezik, és mindegyik az `InputStream` és `OutputStream` osztályokból származik.
- A byte stream-eket csak a legegyszerűbb I/O műveletekhez érdemes használni.

91. Mi az a character stream?

- A Java platform a karakter értékeket Unicode szabvány szerint tárolja.
- A karakter stream I/O automatikusan lefordítja ezt a belső formátumot a helyi karakterkészletre és onnan vissza.
- Nyugati helyi beállításoknál a helyi karakterkészlet jellemzően az ASCII 8-bites kiterjesztése.
- A legtöbb alkalmazás esetén a karakter stream-ekkel végzett I/O nem bonyolultabb, mint a byte stream-ekkel végzett I/O.

92. Mi az a buffered stream?

- Buffered input stream-ek adatokat olvasnak egy buffer nevű memória területről; az alapértelmezett bemeneti API csak akkor kerül meghívásra, amikor a buffer üres.
- Hasonlóképpen, a buffered output stream-ek adatokat írnak a bufferbe, és az alapértelmezett kimeneti API csak akkor kerül meghívásra, amikor a buffer tele van.

93. Scanning and Formatting

- I/O programozása gyakran magában foglalja az adatok átalakítását az emberek által olvashatóbb formátumra. Ennek megkönnyítésére a Java platform két API-t biztosít.
- A Scanner API az inputot egyes tokenekre bontja, amelyek adatdarabokhoz vannak rendelve.
- A Formatting API az adatokat szépen formázott, emberi szem számára olvasható formába állítja össze.

- A Scanner típusú objektumok hasznosak a formázott bemenetek tokenekre bontásában és az egyes tokenek adat típusuknak megfelelő átalakításában.
- Két szintű formázás érhető el:
  - **print** és **println** formázza az egyes értékeket egy szabványos módon.
  - **format** szinte bármilyen számú értéket formáz a formátum string alapján, számos lehetőséggel a pontos formázásra.
- **Scanning**

```
Scanner sc = new Scanner(System.in);
int num = sc.nextInt();
```

- **Formatting**

```
System.out.printf("Szám: %d%n", num);
```

---

94. Mi a 3 standard stream a Java-ban? Melyik mire való?

- A Java platform három standard stream-et támogat:
  - **Standard Input**, elérhető a **System.in** segítségével (bemenet)
  - **Standard Output**, elérhető a **System.out** segítségével (kimenet)
  - **Standard Error**, elérhető a **System.err** segítségével (hiba)

---

95. Mi a data stream?

- A data stream-ek támogatják a primitív adat típusok (boolean, char, byte, short, int, long, float és double) valamint a String értékek bináris I/O-ját.
- Minden data stream implementálja a **DataInput** vagy a **DataOutput** interfészt.
- A legszélesebb körben használt implementációk ezen interfészekre a **DataInputStream** és a **DataOutputStream**.

---

96. Mi az object stream?

- Az **Object stream-ek** támogatják az objektumok I/O-ját.
- A legtöbb, de nem mindegyik standard osztály támogatja az objektumok szerializálását.
- Azok az osztályok, amelyek támogatják, implementálják a **Serializable** jelölő interfészt.

---

97. Mi a tesztelés? Miért fontos?

- A programozás során a "tesztelés" olyan folyamatot jelent, amely során ellenőrizzük egy adott szoftver vagy program működését.
- A tesztelés során a kód működését különböző bemeneti értékeken próbáljuk ki, és az eredményt összehasonlítjuk a várt eredménnyel.
- A tesztelés segít meggyőződni arról, hogy a kód helyesen működik, és minimalizálja a hibák kockázatát a valódi használat során.
- Néhány fő ok, amiért fontos a tesztelés:
  - **Hibák Felfedezése**: A tesztek segítenek felfedezni és kijavítani a kódban lévő hibákat, mielőtt a szoftvert éles környezetben használnák.
  - **Megbízhatóság**: A tesztelt kód megbízhatóbb, mivel kiszűri a hibákat és megerősíti a helyes működést.
  - **Kód Minőségének Javítása**: A tesztelés ösztönzi a jól strukturált és karbantartható kód írását.

- **Dokumentáció**: A tesztek dokumentálják a kód működését és elvárásait, ami segíti a kód továbbfejlesztését és karbantartását.
- **Tesztvezérelt Fejlesztés (TDD)**: A tesztelés segíthet a TDD módszer alkalmazásában, ahol először a tesztek íródnak, majd a kód, és ezzel fokozza a tervezés pontosságát.

---

98. Milyen teszttípusokról volt szó előadáson? Melyik mit csinál?

- **Unit Tesztek**: Az egyes kódrészleteket, például függvényeket vagy metódusokat tesztelik, hogy ellenőrizzék a helyes működésüket.
- **Integrációs Tesztek**: Tesztek, amelyek az alkalmazás különböző részeinek együttműködését vizsgálják.
- **Elfogadási Tesztek**: A teljes alkalmazás működését ellenőrzik, és azt, hogy megfelel-e az ügyfél elvárásainak.
- **Stressz Tesztek**: Az alkalmazás teljesítményét és skálázhatóságát vizsgálják terhelés alatt.
- **Regressziós Tesztek**: A korábban kijavított hibák újbóli megjelenését megelőzik.

---

99. Mire való a JUnit?

- A JUnit egy népszerű keretrendszer a Java nyelvhez, amely lehetővé teszi a Unit tesztek könnyű írását és végrehajtását.
- Használata hatékony módja a Unit tesztek készítésének és a kód minőségének javításának.

---

100. Soroljon fel a JUnit eszközben használatos annotációkból ötöt és magyarázza mire valók!

1. `@Test`: Egy metódust tesztként jelöl.
2. `@BeforeEach`: A tesztek előtt futó inicializáló kód.
3. `@AfterEach`: A tesztek után futó tisztítási kód.
4. `@BeforeAll`: Az összes teszt előtt egyszer lefutó kód.
5. `@AfterAll`: Az összes teszt után egyszer lefutó kód.

---

101. A JUnit eszköz assertation-jai mire valók? Soroljon fel belőlük 10-et és magyarázza őket!

Az assert-ek a tesztelés során elvárt eredmények ellenőrzésére szolgálnak:

1. `assertEquals(expected, actual)`: Két érték egyenlőségét ellenőrzi.
2. `assertNotEquals(unexpected, actual)`: Ellenőrzi, hogy két érték nem egyenlő.
3. `assertTrue(condition)`: Ellenőrzi, hogy a feltétel igaz.
4. `assertFalse(condition)`: Ellenőrzi, hogy a feltétel hamis.
5. `assertNull(object)`: Ellenőrzi, hogy az objektum null.
6. `assertNotNull(object)`: Ellenőrzi, hogy az objektum nem null.
7. `assertSame(expected, actual)`: Ellenőrzi, hogy két objektum ugyanaz.
8. `assertNotSame(unexpected, actual)`: Ellenőrzi, hogy két objektum különböző.
9. `assertThrows(Exception.class, () -> { ... })`: Ellenőrzi, hogy egy kódrészlet kivételt dob.
10. `assertTimeout(Duration.ofSeconds(1), () -> { ... })`: Ellenőrzi, hogy egy művelet adott idő alatt lefut.

---

102. Javadoc mit csinál, mire való, miért használjuk?

- A Javadoc automatikusan dokumentációt generál a kódhoz speciális kommentekből.
- Fontos a kód érthetőségének és fenntarthatóságának növeléséhez.

- A Java hivatalosan ezt az eszközt használja saját könyvtári API dokumentációjának létrehozásához.

---

103. Javadoc eszköz használata esetén a fő leírást hova írja? Milyen elemekhez adhat meg dokumentációt?

- A dokumentációs kommentben a fő leírás az a tartalom, amely a komment elejétől az első blokk címkéig terjed, ha van ilyen, vagy a komment végéig, ha nincs.
- A dokumentációs kommentek csak akkor ismerhetők fel, ha közvetlenül egy modul, csomag, osztály, interfész, konstruktor, metódus, enum tag vagy mező deklarációja előtt helyezkednek el.

---

104. A metódus dokumentációjának (javadoc) mik a kötelező elemei? Magyarázza az egyes elemeket.

1. `@param`: A metódus paramétereinek leírása.
2. `@return`: A visszatérési érték leírása.
3. `@throws`: A metódus által dobott kivételek leírása.

---

105. A metódus dokumentációjának kötelező elemein kívül soroljon fel és magyarázzon 5 standard tag-et (javadoc)!

1. `@author`: A kód írója.
2. `@version`: A kód verziója.
3. `@see`: Hivatkozás más osztályra vagy metódusra.
4. `@since`: A funkció bevezetésének verziója.
5. `@deprecated`: Jelzi, hogy az elem elavult.

---

106. A JAR eszköz mire való, milyen műveleteket lehet vele elvégezni?

- A Java™ Archive (JAR) fájlformátum lehetővé teszi, hogy több fájlt egyetlen archív fájlba csomagoljunk.
- Jellemzően egy JAR fájl tartalmazza az osztályfájlokat és egyéb segédletet biztosító erőforrásokat.
- A JAR fájlformátum számos előnyt kínál:
  - Biztonság
  - Csökkentett letöltési idő
  - Több fájl tömörítése
  - Kiterjesztések csomagolása
  - Csomag lezárás
  - Csomag verziókezelés
  - Hordozhatóság
- Gyakori JAR fájl műveletek:
  - JAR fájl létrehozása:
    `jar cf jar-file input-file(s)`
  - JAR fájl tartalmának megtekintése:
    `jar tf jar-file`
  - JAR fájl tartalmának kinyerése:
    `jar xf jar-file`
  - Specifikus fájlok kinyerése egy JAR fájlból:
    `jar xf jar-file archived-file(s)`

- Alkalmazás futtatása, amely JAR fájlban van csomagolva (Main-class manifest fejléc szükséges): `java -jar app.jar`

---

107. A JAR file „manifest”-jét mutassa be!

- Az alkalmazás belépési pontjának meghatározásához hozzá kell adni a **Main-Class** fejlécet a JAR fájl manifestjéhez. A fejléc formája a következő:

```
Manifest-Version: 1.0
Main-Class: com.example.Main
```

- Amikor JAR fájlt hozunk létre, automatikusan egy alapértelmezett manifest fájlt kap. Egy archívumban csak egy manifest fájl lehet, és annak elérési útja mindig a következő:

```
META-INF/MANIFEST.MF
```

- Amikor JAR fájlt hozunk létre, az alapértelmezett manifest fájl egyszerűen a következő tartalommal rendelkezik:

```
Manifest-Version: 1.0
```

---

108. Mi az a Java modul?

- A modularitás egy magasabb szintű összesítést ad a csomagok felett. Az új kulcsfontosságú nyelvi elem a modul – egy egyedileg elnevezett, újrahasznosítható, kapcsolódó csomagok csoportja, valamint olyan erőforrások (például képek és XML fájlok), és egy modul leíró, amely meghatározza:
  - a modul nevét
  - a modul függőségeit (azaz más modulok, amelyekre ez a modul épít)
  - a csomagokat, amelyeket kifejezetten elérhetővé tesz más modulok számára (minden egyéb csomag implicit módon nem elérhető más modulok számára)
  - a szolgáltatásokat, amelyeket kínál
  - a szolgáltatásokat, amelyeket fogyaszt
  - hogy mely más moduloknak engedélyezi a reflektálást

---

109. A Java modul module-info.java állományában mit jelentenek az exports, exports ... to, requires, uses direktívák?

1. `exports`: Csomagok láthatóvá tétele más modulok számára.
2. `exports ... to`: Csomagok exportálása konkrét moduloknak.
3. `requires`: Függőség egy másik modulra.
4. `uses`: Szolgáltatás használatát jelzi.

---

110. A kollekciók aggregáló műveletei esetén a pipeline-nak milyen részei vannak? Melyik mire való?

- **Forrás**: Ez lehet egy kollekció, egy tömb, egy generátor függvény vagy egy I/O csatorna.
- **Nulla vagy több közbenső művelet**: Egy közbenső művelet, mint például a *filter*, új stream-et hoz létre.
- **Terminális művelet**: Egy terminális művelet, mint például a *forEach*, nem-stream eredményt generál, mint például egy primitív értéket (például egy double értéket), egy gyűjteményt, vagy a *forEach* esetén nem ad vissza semmilyen értéket.

---

111. Hogyan működik a reduce művelet a kollekciók aggregáló műveletei esetén?

- A **Stream.reduce** módszer egy általános célú redukciós művelet.
- Két argumentumot vesz:
  - **identity**: Az identitás elem egyaránt az összegzés kezdeti értéke, és az alapértelmezett eredmény, ha nincsenek elemek a stream-ben
  - **accumulator**: Az akkumulátor függvény két paramétert vesz: a redukció részleges eredményét és a stream következő elemét
- A **reduce** művelet mindig egy új értéket ad vissza. Azonban az akkumulátor függvény is mindig egy új értéket ad vissza, amikor feldolgoz egy elemet a stream-ből.

```
int sum = list.stream().reduce(0, Integer::sum);
```

---

112. Hogyan működik a collect művelet a kollekciók aggregáló műveletei esetén?

- A **collect** művelet módosítja, vagyis mutálja a meglévő értéket.
- A **collect** művelet három argumentumot vesz:
  - **supplier**: A supplier egy gyári függvény; új példányokat hoz létre. A collect művelet számára az eredmény tárolóinak példányait hozza létre.
  - **accumulator**: Az accumulator függvény beilleszti egy stream elemét az eredmény tárolóba.
  - **combiner**: A combiner függvény két eredmény tárolót fogad el, és összevonja a tartalmukat.

---

113. Mutassa be a kollekciók aggregáló műveleteiből a groupingBy és reducing műveleteket!

- A **groupingBy** művelet egy olyan **map**-et ad vissza, amelynek kulcsai azok az értékek, amelyek az alkalmazott lambda kifejezés alapján keletkeznek (ezt a kifejezést nevezik **classification function**-nak).
  - A **groupingBy** művelet két paramétert vesz át: egy **classification function**-t és egy **Collector** példányt.
  - A **Collector** paraméter a **downstream collector**, amit a Java futtatókörnyezet alkalmaz az egyik collector által létrehozott eredményekre. Ennek következtében a **groupingBy** művelet lehetővé teszi, hogy egy **collect** metódust alkalmazzunk a **List** értékekre, amelyeket a **groupingBy** operátor hoz létre.
- A **reducing** művelet három paramétert vesz át:
  - **identity**: Mint a **Stream.reduce** műveletnél, az identity elem mind az összevonás kezdőértéke, mind a stream üres állapota esetén a default érték.
  - **mapper**: A **reducing** művelet ezt a **mapper** függvényt alkalmazza az összes stream elemre.
  - **operation**: Az **operation** függvény az összegzett értékek csökkentésére szolgál.

2. rész

1. **Mi az a Project Lombok?**

   A **Project Lombok** egy Java könyvtár, amely célja, hogy csökkentse a verbózus (nagyon részletes, de nem feltétlenül szükséges) kód mennyiségét, amelyet a fejlesztők nap mint nap írnak. Lombok segítségével automatikusan generálhatók olyan gyakran használt metódusok, mint például getterek, setterek, `toString`, `equals`, `hashCode`, stb., így a fejlesztők nem kell manuálisan megírni őket. Lombok egyedi annotációkat biztosít, melyekkel ezek a metódusok generálhatók a fordítás ideje alatt.

2. **Project Lombok-kal hogyan ad meg gettert, settert?**

   Lombok segítségével a getterek és setterek generálásához a következő annotációkat használhatjuk:

   - **@Getter**: Automatikusan generálja a getter metódust az osztály minden adattagjához.

- **@Setter**: Automatikusan generálja a setter metódust az osztály minden adattagjához.

Példa:

```
@Getter @Setter
public class Person {
    private String name;
    private int age;
}
```

Ebben az esetben Lombok automatikusan generálja a `getName()`, `setName(String name)`, `getAge()`, és `setAge(int age)` metódusokat.

3. **Project Lombok-kal hogyan ad meg toString-et?**

   Lombok segítségével a `toString` metódus automatikusan generálható a **@ToString** annotációval, amely az összes adattagot felhasználja a `toString` implementálásához.

   Példa:

```
@ToString
public class Person {
    private String name;
    private int age;
}
```

   Ebben az esetben Lombok automatikusan létrehozza a `toString()` metódust, amely a `name` és `age` adattagokat tartalmazza.

4. **Project Lombok-kal hogyan ad meg equals-t és HashCode-ot?**

   Lombok a `@EqualsAndHashCode` annotációval automatikusan generálja az `equals()` és `hashCode()` metódusokat. Az alapértelmezett implementáció figyelembe veszi az osztály összes adattagját.

   Példa:

```
@EqualsAndHashCode
public class Person {
    private String name;
    private int age;
}
```

   Lombok ebben az esetben generálja az `equals()` és `hashCode()` metódusokat, amelyeket az osztály példányainak összehasonlítására és hashekelésére használhatunk.

5. **A Jackson könyvtár segítségével milyen eszközök segítségével tud Stringbe illetve állományba írni JSON-t?**

   A Jackson könyvtár segítségével JSON-t a következő eszközökkel írhatunk String-be vagy állományba:

   - **ObjectMapper.writeValueAsString()**: JSON string formátumba történő írás.
   - **ObjectMapper.writeValue()**: JSON fájlba történő írás.

   Példa JSON string írása:

```
ObjectMapper objectMapper = new ObjectMapper();
String jsonString = objectMapper.writeValueAsString(person);
```

   Példa JSON fájlba írása:

```
objectMapper.writeValue(new File("person.json"), person);
```

6. **A Jackson könyvtár segítségével milyen eszközök segítségével tud Stringből illetve állományból JSON-t olvasni?**

   Jackson könyvtár segítségével JSON-t olvashatunk String-ből vagy állományból a következő eszközökkel:

   - **ObjectMapper.readValue(String, Class)**: JSON string-ből való olvasás.
   - **ObjectMapper.readValue(File, Class)**: JSON fájlból való olvasás.

   Példa JSON string-ből történő olvasás:

   ```
   String jsonString = "{\"name\":\"John\", \"age\":30}";
   Person person = objectMapper.readValue(jsonString, Person.class);
   ```

   Példa JSON fájlból történő olvasás:

   ```
   File file = new File("person.json");
   Person person = objectMapper.readValue(file, Person.class);
   ```

7. **Soroljon fel és magyarázzon 3 Jackson annotációt!**

   - **@JsonProperty**: Lehetővé teszi, hogy az adattag nevei eltérjenek a JSON kulcsaitól. Az annotációval megadható a JSON kulcs, amelyhez az osztály adattagja kapcsolódik.

     ```
     @JsonProperty("full_name")
     private String name;
     ```

   - **@JsonIgnore**: Ezt az annotációt alkalmazva egy adattag nem kerül be a JSON-ba, sem íráskor, sem olvasáskor.

     ```
     @JsonIgnore
     private int age;
     ```

   - **@JsonFormat**: Lehetővé teszi a dátumok formázását JSON-ba történő íráskor és olvasáskor. Például meghatározhatjuk a dátum formátumát.

     ```
     @JsonFormat(pattern = "yyyy-MM-dd")
     private LocalDate birthDate;
     ```

8. **Az előadásanyag a Java Annotáció 3 használatát említette, melyek voltak ezek?**

   - **Információ a fordító számára** — Az annotációk használhatók a fordító által hibák észlelésére vagy figyelmeztetések elnyomására.
   - **Fordítási idő és telepítési idő feldolgozása** — Szoftvereszközök feldolgozhatják az annotációs információkat kód, XML fájlok stb. generálására.
   - **Futásidejű feldolgozás** — Egyes annotációk futásidőben is elérhetők és vizsgálhatók.

9. **Mi az az annotáció interfész és mire lehet használni?**

   - Egyfajta interfész.
   - Az annotáció interfész deklarációjának megkülönböztetése érdekében a `@` szimbólum előzi meg az `interface` kulcsszót.
   - Az összes szabály, amely a normál interfész deklarációkra vonatkozik, alkalmazható az annotációs interfész deklarációkra is.
   - Az annotációs interfész deklarációja megadhat egy felső szintű interfészt vagy egy tag interfészt, de nem egy lokális interfészt.
   - Az annotációs interfész közvetlen szuperinterfész típusa mindig `java.lang.annotation.Annotation`.

10. **Pythonban hogy néz ki egy osztály konstruktora?**

Pythonban az osztály konstruktorát az `__init__` metódus jelenti, amelyet az osztály példányosításakor automatikusan meghív a rendszer.

Példa:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

11. **Pythonban hogy lehet egy osztály példányszintű és osztályszintű adattagjait definiálni?**

- **Példányszintű adattag**: Ezek az osztály példányához tartoznak, és az `__init__` konstruktorban definiálhatók.
- **Osztályszintű adattag**: Ezek az osztályhoz tartoznak, és minden példány számára közösek. Az osztályszintű adattagokat közvetlenül az osztályban lehet definiálni.

Példa:

```
class Person:
    species = "Homo sapiens"  # osztályszintű adattag

    def __init__(self, name, age):
        self.name = name  # példányszintű adattag
        self.age = age  # példányszintű adattag
```

12. **Python hogy néz ki az öröklés osztályok és interfészek között?**

Pythonban az öröklés a következőképpen néz ki, ahol az osztályok egymástól örökölhetnek:

- **Osztály öröklés**: Egy osztály örökölhet egy másik osztályt.
- **Interfészek**: Pythonban nincs dedikált interfész fogalom, de egy osztály több osztályt is örökölhet, így több interfészhez hasonló struktúrát is képezhet.

  ```
  class AlapOsztaly:
      def kiir(self):
          return "Alap osztály"

  class SzarmaztatottOsztaly(AlapOsztaly):
      def kiir(self):
          return "Származtatott osztály"
  ```