HATÉKONY JAVA PROGRAMOZÁS

1. forduló

MSCI

A kategória támogatója: MSCI

Ismertető a feladathoz

Kérjük, hogy a feladatlap indítása előtt mindenképp olvasd el az alábbi útmutatót:

- MINDEN kérdésre **van helyes válasz**.
- Olyan kérdés **NINCS**, amire az összes válasz helyes, ha mégis az összes választ bejelölöd, arra a feladatra automatikusan 0 pont jár.
- A **radio button-os kérdésekre egy helyes válasz van**.
- **Ha lejár a feladatlap ideje, a rendszer AUTOMATIKUSAN** beküldi azt az addig megjelölt válaszokkal.
- Azokat a feladatlapokat, amelyekhez **csatolmány** tartozik, javasoljuk **NEM mobilon** elindítani, erre az érintett feladatlapok előtt külön felhívjuk a figyelmet.
- Az **adatbekérős feladatokra NEM jár részpontszám**, csak a feleletválasztósakra.
- **Helyezéseket a 4. forduló után mutatunk**, százalékos formában: adott kategóriában a TOP 20-40-60%-hoz tartozol.
- **Badge-ket** szintén a 4.forduló után kapsz majd először.
- Ha egyszerre több böngészőből, több ablakban vagy több eszközről megnyitod ugyanazt a feladatlapot, **nem tudjuk vállalni** az adatmentéssel kapcsolatban esetlegesen felmerülő anomáliákért a felelősséget!
- A hét forduló során az egyes kategóriákban (de nem feltétlenül mindegyikben) **könnyű-közepes-nehéz kérdésekkel** egyaránt találkozhatsz majd.

Jó versenyzést kívánunk!

Felhasznált idő: 00:00/10:00 **Elért pontszám:** 0/10

1. feladat 0/1 pont

Adott az alábbi Ticker osztály. Melyik kódrészlet ad **helyes** megvalósítást a hashcode és equals metódusokra?

```
class Ticker extends StockPriceProvider {
  String symbol;
  BigDecimal tradedPrice;
}
```

Válaszok

- [ ] A:

```
@Override
public int hashCode() {
  final int prime = 31;
  int result = super.hashCode();
  result = prime * result + Objects.hash(symbol, tradedPrice);
  return result;
}

@Override
public boolean equals(Object obj) {
  if(this == obj)
    return true;
  if(!super.equals(obj))
    return false;
  if(!(obj instanceof Ticker))
    return false;
  Ticker other = (Ticker) obj;
  return Objects.equals(symbol, other.symbol) && Objects.equals(tradedPrice, other.tradedPrice);
}
```

B:

```
@Override
public int hashCode() {
  final int prime = 31;
  int result = super.hashCode();
  result = prime * result + Objects.hash(symbol);
  return result;
}

@Override
public boolean equals(Object obj) {
  if(this == obj)
    return true;
  if(!super.equals(obj))
    return false;
  if(getClass() != obj.getClass())
    return false;
  Ticker other = (Ticker) obj;
  return Objects.equals(symbol, other.symbol) && Objects.equals(tradedPrice, other.tradedPrice);
}
```

Ez a válasz helyes, de nem jelölted meg.

C:

```
@Override
public int hashCode() {
  final int prime = 31;
  int result = super.hashCode();
  result = prime * result + Objects.hash(symbol, tradedPrice);
  return result;
}

@Override
public boolean equals(Object obj) {
  if(this == obj)
    return true;
  if(!super.equals(obj))
    return false;
  if(getClass() != obj.getClass())
    return false;
```

```
  Ticker other = (Ticker) obj;
  return Objects.equals(symbol, other.symbol);
}
```

D:

```
@Override
public int hashCode() {
  final int prime = 31;
  int result = super.hashCode();
  result = prime * result + Objects.hash(symbol, tradedPrice);
  return result;
}

@Override
public boolean equals(Object obj) {
  if(this == obj)
    return true;
  if(obj == null || getClass() != obj.getClass())
    return false;
  Ticker other = (Ticker) obj;
  return Objects.equals(symbol, other.symbol) && Objects.equals(tradedPrice, other.tradedPrice);
}
```

E:

```
@Override
public int hashCode() {
  final int prime = 31;
  int result = super.hashCode();
  result = prime * result + (tradedPrice != null ? tradedPrice.hashCode() : 0) + (symbol != null ? symbo
  return result;
}

@Override
public boolean equals(Object obj) {
  if(this == obj)
    return true;
  if(!super.equals(obj))
    return false;
  if(getClass() != obj.getClass())
    return false;
  Ticker other = (Ticker) obj;
  return Objects.equals(symbol, other.symbol) && Objects.equals(tradedPrice, other.tradedPrice);
}
```

Ez a válasz helyes, de nem jelölted meg.

Magyarázat

A: Helytelen megvalosítás: **instanceof** használata nem ajánlott, az equals hívások szimmetriája sérülhet. Ha "A" osztály "a" példány "B" extends "A" osztály "b" példány esetén a.equals b és b.equals a eredménye megtörténhet, hogy nem egyezik meg.

B: Helyes megvalósítás. Nem probléma, hogy a **tradedPrice** hiányzik a hashcode-ból.

C: Helytelen megvalósítás. **tradedPrice** csak a hashcode-ban szerepel, az equals hashcode contract sérülhet.

D: Helytelen megvalosítás. a hashcode meghívja a super megvalosítását, de az equals nem.

E: Helyes megvalósítás.

2. feladat 0/1 pont

Melyik függvényhívásnál dobódik **NullPointerException**?

```
public class NullPrintout {
 public static void main(String[] args) {
   Integer test = null;

   System.out.println("The value is " + test);
   System.out.println(String.format("The value is %s", test));
   System.out.println(test + " is the value");

   int nonNull = 3;
   System.out.println(test + nonNull);
  }
}
```

Válasz

A:

```
System.out.println("The value is " + test);
```

B:

```
System.out.println(String.format("The value is %s", test));
```

C:

```
System.out.println(test + " is the value");
```

D:

```
System.out.println(test + nonNull);
```

Ez a válasz helyes, de nem jelölted meg.

Magyarázat

A **System.out.println(test + nonNull)** esetben az unboxing miatt a null referencia kivételt okoz.

Minden más esetben nincs gond.

3. feladat 0/1 pont

Mit ír ki az alábbi kódrészlet, ha meghívjuk a **main** függvényt?

```
public class Lister {

  public static void main(String[] args) {
    List<Integer> list = new ArrayList<>();
    list.add(0);
    list.add(1);
    list.add(2);
    list.add(3);

    list.remove(0);
    list.remove(1);

    System.out.println(list);
  }
}
```

Válaszok

- [x] a program nem fordul
  Ez a válasz helyes, de nem jelölted meg.
- [x] [1, 3]
  Ez a válasz helyes, de nem jelölted meg.
- [ ] [2, 3]
- [ ] A program fordul, de kivételt dob futás közben.

Magyarázat

A remove(int) esetében az argumentum az indexre vonatkozik, nem az értékre. Tehat az első (0) és a második (2) értéket vesszük ki a listából.

4. feladat 0/3 pont

Válaszd ki azokat az állításokat, amelyek egyéb körülményektől függetlenül biztosan igaz értékkel térnek vissza Java 11-ben!

Válaszok

- [ ] Integer.valueOf(255) == Integer.valueOf(255)
- [x] Integer.valueOf(8) == Integer.valueOf(8)
  Ez a válasz helyes, de nem jelölted meg.
- [ ] new Integer(127) == new Integer(127)
- [ ] new Long(103) == new Long(103)
- [x] Byte.valueOf((byte)97) == Byte.valueOf((byte)97)
  Ez a válasz helyes, de nem jelölted meg.
- [x] Byte.valueOf("2") == Byte.valueOf("1") + 1
  Ez a válasz helyes, de nem jelölted meg.
- [x] Integer.valueOf(200) == Integer.valueOf(199) + 1
  Ez a válasz helyes, de nem jelölted meg.

- [ ] "200" == new String("2") + "0" + "0"
- [ ] new Object().equals(new Object())

Magyarázat

Wrapper osztályok esetében mint a Byte, Integer, Long, Boolean, Short a valueOf metódus egy cache-ből adja vissza a példányokat. Integer/Long/Short esetében alapértelmezetten csak a -128-tól 127-ig tartó értéktartomány van a cache-ben.

Ha direkt a konstruktort hívjuk, akkor mindig új példány keletkezik.

Egy additív operátor (+ és -) primitívvé konvertálja az argumentumokat, ezért a **Integer.valueOf(200) == Integer.valueOf(199) + 1** primitív értékeket hasonlít össze.

Tehát a helyes megoldások:

Integer.valueOf(8) == Integer.valueOf(8)

Byte.valueOf((byte)97) == Byte.valueOf((byte)97)

Byte.valueOf("2") == Byte.valueOf("1") + 1

Integer.valueOf(200) == Integer.valueOf(199) + 1

5. feladat 0/4 pont

Az alábbiak közül melyek a JIT (HotSpot) fordító által végrehajthatott optimalizációk?

Válaszok

- [x] escape analysis - a JIT compiler megvizsgálja hogy egy lokális objektum "megszökhet-e" a metódusból ahol létrehozták. Ennek függvényében a lockolás vagy a memória lefoglalás változhat.
  Ez a válasz helyes, de nem jelölted meg.
- [x] loop unrolling - ciklusok esetén az utasítások megismétlésével helyettesíti a ciklus ismételt végrehajtását
  Ez a válasz helyes, de nem jelölted meg.
- [ ] constant inlining - static final valtozók értékének behelyettesítése
- [x] method inlining - behelyettesíti a metódus tartalmát a metódushívás helyett
  Ez a válasz helyes, de nem jelölted meg.
- [ ] String compression - +CompressStringBytes opcióval a megadott méretű vagy annál nagyobb String objektumokat a heap-en tömörítjük deflate algoritmussal, ha ritkán olvassa vagy írja a program.
- [ ] mindegyik korábbi válasz helyes

Magyarázat

Helyes válaszok:

escape analysis

loop unrolling

method inlining

A constant inlining-et már a javac elvégzi a JIT előtt, a string compression ebben a formában, ilyen kapcsolóval és deflate tömörítéssel nem létezik. A Java 9-ben bevezetett "Compact String" nem így működik és nincs kapcsolója.

KÉSZÍTETTE cone

Megjelenés

Világos