1. b)

B1 b1 = new B1();

```
public B1() {
    super();
    System.out.println("B1()");
}
```

B1 b2 = new B2();

```
class B2 extends B1 {

    static BI1 b1 = new B1();
    String str = "";
```

Zatim se pozivaju redom super() -B1, a nakon toga B2()!!
Znači, zbog "b1" imamo B1() prije super unutar B2 Constructor-a imamo B1() te onda imamo još B2() zbog B2 Constructor-a!!
b1 ima priroritet nad super() jer on okarakterisan sa static!!

BI1 b3 = **new** B3("b3");
Poziva se super() unutar B2 u kojem se poziva super() od B1 tako da je ispis redom:
B1() B2() ali pazi, "b1" UNUTAR B2 JE "static" TAKO DA SE ON KREIRAO JEDNOM I TO JE TO..

BI1 b3 = **new** B3("b3");
Poziva se super() unutar B2 u kojem se poziva super() od B1 tako da je ispis redom:
B1() B2() ali pazi, "b1" UNUTAR B2 JE "static" TAKO DA SE ON KREIRAO JEDNOM I TO JE TO..

b1.print(b1.j);

```
public void print(String str) {
    System.out.println(str.isEmpty() ? "" : str.charAt(0));
}
```

b2.print("av");

```
public void print(String str) {
     str = str + 'a';
     System.out.println(str);
}
```

ZBOG POLIMORFIZMA!

U klasi B2, imamo implementiranu metodu imena “print”!!

Iako je referenciramo iz njoj roditeljske klase!!

b2.close();

```
public void close() {
     System.out.println("B2 closed...");
}
```

b4.print("b3");

```
public void print(String str) {
     str = str + 'a';
     System.out.println(str);
}
```

((BI1) b3).close();

```
public void close() {
     System.out.println("B2 closed...");
}
```

jer B3 extends B2, a metoda close() u B2 je public, tako da se desio inheritance!!

```
try (B3 b5 = new B3("a")) {
    b5.print("Zadnja linija?");
}
```

Zbog Constructor Call imamo redom B1() te B2() onda

```
public void print(String x) {
    StringBuilder builder = new StringBuilder(x + str);
    System.out.println(builder.reverse());
}
```

ali PAZI, ZBOG AUTOCLOSABLE!! IMAMO:

```
public void close() {
    System.out.println("B2 closed...");
}
```

Jer

Kada se `close()` poziva automatski:

Metoda `close()` na resursu unutar `try-with-resources` bloka poziva se automatski u sljedećim situacijama:

1. **Normalan završetak bloka:** Kada se izvršavanje `try` bloka završi normalno (bez izuzetaka), `close()` metoda se poziva na svim resursima deklarisanim u zagradama `try-with-resources`.
2. **Izuzetak unutar bloka:** Ako se izuzetak dogodi unutar `try` bloka, `close()` metoda se i dalje poziva na svim resursima deklarisanim u zagradama `try-with-resources` **prije nego što se izuzetak propagira dalje ili uhvati u** `catch` **bloku.**

Java

```
try (B3 b5 = new B3("a")) {
    b5.print("Zadnja linija?");
}
```

Ovaj `try` blok je sintaksa poznata kao `try-with-resources` (uvedena u Javi 7).

Evo zašto se izvršava bez eksplicitnog `catch` bloka:

1. `try-with-resources` **ne zahtijeva** `catch` **blok:**
   - Primarna svrha `try-with-resources` bloka je da osigura da se resursi (objekti koji implementiraju `java.lang.AutoCloseable` ili `java.io.Closeable`) automatski zatvore kada se `try` blok završi, bilo normalno ili zbog izuzetka.
   - Za razliku od tradicionalnog `try-catch-finally` bloka, `try-with-resources` **ne zahtijeva** da odmah nakon `try` bloka slijedi `catch` ili `finally` blok.
   - **Važno:** Ako se izuzetak dogodi unutar `try-with-resources` bloka i nema `catch` bloka koji bi ga uhvatio, taj izuzetak će se **propagirati dalje** uz stack poziva, baš kao što bi se desilo da se izuzetak dogodi izvan bilo kakvog `try-catch` bloka. Program bi se tada srušio (terminirao) ako se izuzetak ne uhvati negdje više u pozivnom steku.

c)

```
void metoda() throws Throwable {
    C2 c2 = new C2();
    try {
        c2.metoda();
        System.out.println("C1: metoda()");
    } finally {
        System.out.println("finally");
    }
}
```

"throws Throwable" u potpisu metode, poput ovog u vašem priloženom kodu (`void metoda() throws Throwable { ... }`), znači da **ova metoda može baciti (throw) bilo koju vrstu izuzetka ili greške (Throwable)**.

1. Šta je `Throwable` ?

- `Throwable` je **bazna klasa svih grešaka i izuzetaka u Javi**. To je klasa iz koje direktno ili indirektno nasljeđuju sve ostale klase koje se mogu baciti (throw).
- Ima dvije glavne podklase:
  - `Exception` : Predstavlja izuzetne uslove koje aplikacija može željeti da uhvati i obradi. Većina izuzetaka spada u ovu kategoriju. `Exception` se dalje dijeli na:
    - **Provjerene izuzetke (Checked Exceptions)**: Moraju biti eksplicitno uhvaćeni (`try-catch`) ili deklarisani (`throws SomeException`) u potpisu metode. Kompajler će vas natjerati da ih obradite. Primjeri: `IOException` , `SQLException` , `ClassNotFoundException` .
    - **Neprovjerene izuzetke (Unchecked Exceptions)** ili `RuntimeException` : Ne moraju biti eksplicitno uhvaćeni ili deklarisani. Obično ukazuju na programske greške. Primjeri: `NullPointerException` , `ArrayIndexOutOfBoundsException` , `ArithmeticException` .
  - `Error` : Predstavlja ozbiljne probleme koji obično ukazuju na nešto izvan kontrole aplikacije i koji se obično ne mogu oporaviti. Trebali biste ih izbjegavati hvatati (osim u vrlo specifičnim slučajevima, npr. radi logovanja). Primjeri: `OutOfMemoryError` , `StackOverflowError` , `VirtualMachineError` .

3. Šta znači `throws Throwable`?

- Kada vidite `throws Throwable` u potpisu metode, to je vrlo široka deklaracija. To znači da metoda može potencijalno baciti **bilo koji** `Exception` (provjereni ili neprovjereni) ili **bilo koji** `Error`.
- **Implikacije:**
  - **Pretjerano široka deklaracija:** Obično se smatra lošom praksom da se u potpisu metode deklariše `throws Throwable`. Razlog je što to skriva specifičnost grešaka koje metoda može baciti. Programer koji poziva tu metodu ne zna tačno kakve vrste izuzetaka može očekivati i mora biti spreman da uhvati sve.
  - **Svi provjereni izuzeci su pokriveni:** Ako metoda `throws Throwable`, onda ne morate eksplicitno navesti svaki provjereni izuzetak koji metoda može baciti. `Throwable` ih sve pokriva. Međutim, ovo je i razlog zašto se to ne preporučuje, jer gubite jasnost i kompajler vam ne može pomoći da identifikujete specifične obrade izuzetaka.
  - **Primjer iz vašeg koda:** U metodi `metoda()` u klasi `C1`, poziva se `c2.metoda()`. Ako `c2.metoda()` može baciti provjereni izuzetak (kao što je `CE1` u vašem potpunom kodu), onda bi `C1.metoda()` morala ili uhvatiti taj izuzetak ili ga proglasiti. Pošto je deklaraisano `throws Throwable`, kompajler će to dozvoliti jer `Throwable` pokriva `CE1`.

Kada se `throws Throwable` može vidjeti (i kada je opravdano)?

1. **Glavna (`main`) metoda:** Ponekad, za jednostavne testne programe ili primjere, možete vidjeti `main` metodu deklarisanu kao `public static void main(String[] args) throws Throwable`. Ovo omogućava da bilo koji izuzetak koji se javi unutar `main` metode bude prosljeđen JVM-u, koji će ga ispisati na konzolu i terminirati program. U produkcionom kodu, to se izbjegava.
2. **Okviri (Frameworks) ili Testne metode:** U nekim framework-ovima (npr. JUnit testovima) ili vrlo specifičnim infrastrukturnim kodovima, može se koristiti `throws Throwable` kada metoda služi kao generički ulaz za kod koji može bacati nepredvidive izuzetke, a okvir je odgovoran za njihovu opću obradu.
3. **Refleksija (Reflection):** Kod koji intenzivno koristi refleksiju (npr. pozivanje metoda dinamički) često mora raditi sa `InvocationTargetException` i drugim izuzecima koji su omotani, a `throws Throwable` može biti način da se sve to proglasi. Ipak, i tu je bolja praksa uhvatiti specifične izuzetke i re-throwati ih kao `RuntimeException` ili specifičnije provjerene izuzetke.

Zaključak

`throws Throwable` u potpisu metode je legalno u Javi i znači da metoda može baciti bilo koji izuzetak ili grešku. Međutim, generalno se **ne preporučuje** u običnom aplikativnom kodu jer:

- Smanjuje čitljivost koda.
- Otežava programerima koji pozivaju metodu da znaju koje izuzetke trebaju specifično obraditi.
- Može prikriti stvarne probleme jer dozvoljava da provjereni izuzeci prođu "ispod radara".

Bolja praksa je deklarisati **najspecifičnije izuzetke** koje metoda zaista može baciti, ili, ako se očekuju mnoge vrste provjerenih izuzetaka, uhvatiti ih i re-baciti kao specifični `RuntimeException` ili prilagođeni provjereni izuzetak ako je to u skladu sa logikom aplikacije.

d)

D1 d1 = D1.getInstance();

```
public static D1 getInstance() {
    if (instance == null)
        instance = new D1();
    return instance;
}
```

UVIJEK OBRATI PAŽNJU NA KLJUČNU RIJEČ “static”!!
Elem, na hep-u imamo objekat klase D1, u klasi D1, statička referenca upućuje na taj objekat..

D1 d2 = D1.getInstance();
Sad i d1 i d2 upućuju na taj isti objekat..

ZNAČI!! -instace je referenca koja pripada klasi D1 i ona ukazuje na objekat klase D1 koji se nalazi na heap-u!!

d1.rijec = "hello";
d2.rijec = new String("HELLO");

```
System.out.println(d1.rijec == d2.rijec);
System.out.println(d1.rijec.equalsIgnoreCase(d2.rijec));
System.out.println(d1.compareTo(d2));
System.out.println(d1.rijec + " ? " + d2.rijec);
```

true –jer d1.rijec i d2.rijec referenciraju isti objekat!!
Jer d1 i d2 također referenciraju isti objekat!!

true -jer “Hello”.equalsIgnoreCase(“Hello”)
0 – jer rezultat poređenja je 0, identičan je sadržaj..
HELLO ? HELLO

3. Zadatak..

a)

```
public class Test1 {
    public static void main(String... a) {
        Klasa1 k1 = new Klasa1() {
            public void add(int a, int b) {
                return a + b;
            }
            public void sub(int a, int b) {
                return a - b;
            }
        };
        System.out.println(k1.add(10,20));
        System.out.println(k1.add(5,10));
    };;;
    ;

    static abstract class Klasa1 {
        public abstract add(int x, int y);
        public abstract sub(int x, int y);
    }
}
```

Pravila za `abstract` klase i metode u Javi:

1. `abstract` **metoda mora biti eksplicitno deklarisana kao** `abstract`**.**
   - Nijedna metoda u Java klasi (bilo apstraktnoj ili konkretnoj) nije `abstract` po defaultu. Morate eksplicitno koristiti ključnu riječ `abstract` u njenoj deklaraciji.
   - Ako metoda nema tijelo i nije označena kao `abstract`, to je greška u kompilaciji (osim ako je u interfejsu prije Jave 8).
2. `abstract` **metoda može imati različite modifikatore pristupa.**
   - Nijedna metoda u Javi nije `public` po defaultu. Podrazumijevani (default) modifikator pristupa, ako nije naveden, je `package-private` (tj. metoda je vidljiva samo unutar istog paketa).
   - `abstract` metode mogu biti `public`, `protected` ili `package-private`. **Ne mogu biti** `private`, jer ih moraju implementirati podklase.
   - U vašem primjeru, metode `add` i `sub` su `public` jer ste to **eksplicitno naveli** ključnom riječi `public`. Nisu `public` po defaultu.
3. `abstract` **klasa ne mora imati samo** `abstract` **metode.**
   - Apstraktna klasa može sadržavati kombinaciju apstraktnih i konkretnih (sa implementacijom) metoda. Nije nužno da su sve njene metode apstraktne.
   - Međutim, ako klasa sadrži **barem jednu** apstraktnu metodu, ona sama mora biti deklarisana kao `abstract`.

b)
Uvijek obrati pažnju:
-potpis main metode
-privatni članovi/metode klase
-statički članovi/metode klase

Ovde konkretno, uočiš privatni-x, i vidiš da se on poziva unutar statičke metode!!

c) Moraš znati šta se nalazi na heap/stack/data-egment!!