```
B2(id = 0): created
B2(id = 0): created
B2(id = 0): created
B2(id = 1): created
B2(id = 2): created
1. B2(id = 0)
BI3BI1
2. B2(id = 0)
BI3BI1
3. B2(id = 0)
BI3BI1
4. B2(id = 1)
BI3BI1
5. B2(id = 2)
BI3BI1
```

Polazimo od glavne funkcije:
B2 b1 = new B2(-2);
B2 b2 = new B2(-1);
B2 b3 = **new** B2(0);

```
public B2(int id) {
    if (id > 0 || new Random().nextDouble() > 1)
        this.id = id;
    else
        id = count;
    System.out.println(this + ": created");
}
```

id = count; se odnosi na **id – argument Constructor-a**, promjenjiva koja je argument funkcije znači, dok se **this.id** ne mijenja a kako se isti nigdje nije eksplicitno inicijalizovao, Java kompajler podrazumijeva da je njegova vrijednost = 0!!

B2 b4 = **new** B2(1);

B2 b5 = **new** B2(2);

```
public B2(int id) {
    if (id > 0 || new Random().nextDouble() > 1)
        this.id = id;
    else
        id = count;
    System.out.println(this + ": created");
}
```

```
int i = 1;
for (B2 b : Arrays.asList(b1, b2, b3, b4, b5)) {
    System.out.println(b.reverse(i + ". " + b.toString()));
    System.out.println(b.concat(BI1.x, BI3.x));
    i++;
}
```

b.reverse(..

```
class B2 implements BI1 {
    static int count;
    {
        count++;
    }
    int id;

    public B2(int id) {
        if (id > 0 || new Random().nextDouble() > 1)
            this.id = id;
        else
            id = count;
        System.out.println(this + ": created");
    }
    public String toString() {
        return "B2(id = " + id + ")";
    }
    public String reverse(String str) {
        return str;
    }
}
```

jer b - direktno referencira objekat klase B2, u toj klasi imamo jasno redefinisanu metodu reverse()!!

i + ". " + b.toString())

što ispiše **i.B2(id=X**), znači vrijednost od i. pa poziv metode **toString()** B2 klase!!

Klasa B2, implementira interfejs BI1 u kojem imamo jasno redefinisanu metodu concat()

```
class B2 implements BI1 {
    static int count;
    {
        count++;
    }
    int id;

    public B2(int id) {
        if (id > 0 || new Random().nextDouble() > 1)
            this.id = id;
        else
            id = count;
        System.out.println(this + ": created");
    }
    public String toString() {
        return "B2(id = " + id + ")";
    }
    public String reverse(String str) {
        return str;
    }
}

interface BI1 extends BI2, BI3 {
    String x = "BI1";
    default String concat(String str1, String str2) {
        return str2 + str1;
    }
    String reverse(String str);
}
```

c)

```
C1() {
    System.out.println("C1()");
}

public static void main(String[] args) {
    C1 c1 = new C1();
```

Pozvaće Constructor i ispisaće C1() ..

```
try {
    c1.metoda();
    System.out.println("main 1");
} catch (CE2 e) {
```

vodi me u

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

Prethodno što će ispisati C2() zbog ConsturctorCall, a dalje me vodi u c2.metoda():

(Elem, obrati pažnju na throws Throwable –što znači da je metoda spremna na bacanje bilo kakve vrste izuzetaka!!)

```
class C2 {
    C2 () {
        System.out.println("C2()");
    }
    void metoda() throws CE1 {
        C3 c3 = new C3();
        System.out.println("C2: metoda()");
        c3.metoda();
    }
}
```

I tako se te tri linije code-a izvršavaju redom..došli smo do

```
class C3 {
    C3 () {
        System.out.println("C3()");
    }
    protected void metoda() throws CE1 {
        System.out.println("C3: metoda()");
        throw new CE2("CE2");
    }
}
```

Dalje idemo u (Jer je bačen CE2!!):

```
class CE1 extends Throwable {
    CE1(String s) {
        super(s);
        System.out.println("CE1: " + s);
    }
}

class CE2 extends CE1 {
    CE2(String s) {
        super(s);
        System.out.println("CE2: " + s);
    }
}
```

To nam ispisuje:
CE1: CE2
CE2: CE2

E SAD, BAČEN JE CE2 KOJI SE PROPAGIRA!!
Izuzetak propada, sa Stack-a se skidaju prethodno pozvane metode i dolazimo do

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

pri čemu se izvršava finally ali CE2 nastavlja da pada!!

```
public static void main(String[] args) {
    C1 c1 = new C1();
    try {
        c1.metoda();
        System.out.println("main 1");
    } catch (CE2 e) {
        System.out.println("main 2: " + e);
    } catch (CE1 e) {
        System.out.println("main 3: " + e);
    } catch (Throwable e) {
        System.out.println("main 4: " + e);
    }
}
```

UHVAĆEN JE!!

main 2: CE2 je naš poslednji ispis!!!

d)

🧠 Ukratko:

Kod se ne može kompajlirati jer:

> Interfejs `DI3` pokušava da da `default` implementaciju metode `metoda1()`, koja je već označena kao `abstract` u nadinterfejsu `DI2`. Klasa `D1`, koja implementira `DI3`, ne obezbeđuje **svoju** implementaciju `metoda1()`, pa Java kompajler ne zna **koju verziju da koristi**.

✅ 1. Konflikt između `abstract` i `default` metode

❌ Kao u tvojem primjeru `b)`:

java Copy Edit

```java
interface DI2 extends DI1 {
    void metoda1(); // apstraktna
}

interface DI3 extends DI2 {
    default void metoda1() {  // ❌ ne može redefinisati apstraktnu metodu kao default
        System.out.println("DI3.metoda1()");
    }
}
```

🔥 Greška:

> *"class D1 is not abstract and does not override abstract method metoda1() in DI2"*

📌 Zašto?

Ako **nadređeni interfejs** zahtijeva da metoda bude `abstract`, **podinterfejs ne smije da je implementira kao** `default`, osim ako se očekuje da klasa koja to implementira **sama obezbijedi konačnu verziju**.

✅ 2. Višestruko nasleđivanje interfejsa sa istim default metodama

java Copy Edit

```java
interface A {
    default void metoda() {}
}

interface B {
    default void metoda() {}
}

interface C extends A, B {} // ❌ Greška!
```

🔥 Greška:

> *"Duplicate default methods named metoda with the parameters () and () are inherited from types A and B"*

📌 Zašto?

Kompajler ne zna **koju verziju da koristi**, jer obje interfejs metode imaju default implementaciju.

✅ 3. Korištenje `default` u klasi, ne interfejsu

java Copy Edit

```java
class Test {
    public default void metoda() {} // ❌ Ne može!
}
```

🔥 Greška:

> *"modifier default not allowed here"*

📌 Zašto?

`default` metode su **dozvoljene samo unutar interfejsa.**

✅ 4. Statička metoda u interfejsu ne može se nasleđivati

java Copy Edit

```java
interface A {
    static void metoda() {}
}

class B implements A {
    void test() {
        metoda(); // ❌
    }
}
```

🔥 Greška:

> *Cannot find symbol: metoda()*

📌 **Zašto?**

Statičke metode iz interfejsa **nisu nasleđene** – pozivaju se **putem imena interfejsa**:

java Copy Edit

```java
A.metoda(); // ✅
```

✅ 5. Neimplementirana apstraktna metoda iz interfejsa

java Copy Edit

```java
interface A {
    void metoda();
}

class B implements A {} // ❌
```

🔥 Greška:

> *"class B is not abstract and does not override abstract method metoda() in A"*

📌 **Rešenje**: Implementiraj metodu `metoda()` u klasi `B`, ili označi `B` kao `abstract`.

✅ 6. **Nepodudaranje povratnog tipa**

java Copy Edit

```java
interface A {
    int metoda();
}

class B implements A {
    public void metoda() {} // ❌
}
```

🔥 Greška:

> *method does not override or implement a method from a supertype*

📌 Zašto?

Potpis metode mora **tačno odgovarati** onom iz interfejsa – povratni tip, ime i parametri.

✅ 7. Nepravilna upotreba modifikatora u interfejsu

java Copy Edit

```java
interface A {
    private void metoda(); // ❌ ako nije u Java 9+ i ako se ne koristi pravilno
}
```

📌 **Samo od Jave 9+** su dozvoljene `private` metode u interfejsima, i to samo ako se koriste kao pomoćne metode u okviru `default` ili `static` metoda.

✅ Ukratko – da bi interfejs izazvao grešku pri kompajlaciji:

Problem	Objašnjenje
`default` redefiniše `abstract`	Kao u tvom slučaju – ne dozvoljava se
Dvostruki `default` nasleđeni	Moraš eksplicitno rešiti konflikt
Statička metoda pozvana kao obična	Statičke metode iz interfejsa se ne nasleđuju
`default` metoda u klasi	Dozvoljeno samo u interfejsu
Apstraktna metoda neimplementirana	Klasa mora da je implementira ili bude `abstract`
Pogrešan potpis metode	Povratni tip i parametri moraju biti isti

e)

```
Runnable[] niz = {new E1(), new E1(),
    new Thread(), new E1(), new Thread()};
```

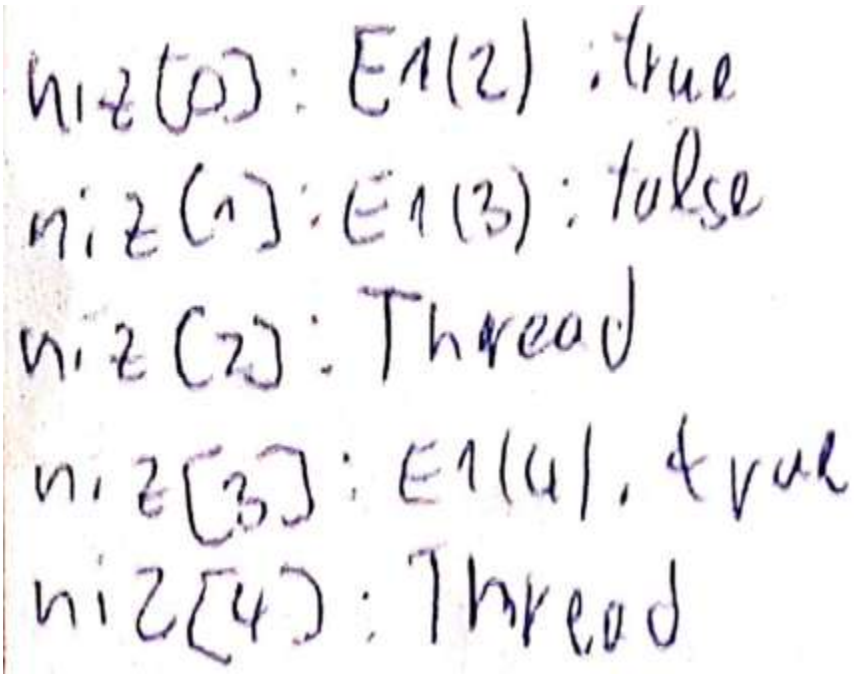
To nas vodi u:

```
static int c = 1;
{
    c++;
}
int id;

public E1() {
    id = c;
    if (id % 2 == 0)
        setDaemon(true);
    else
        setDaemon(false);
}
```

Svaki put kada uđeš u klasu, IZVRŠIĆE SE TAJ NON-STATIC-BLOCK!!

Ovakav imaš rezultat:



Dalje:

```
for (Runnable r : niz) {
    System.out.println("Checking...");
    if (r instanceof Thread) {
        Thread t = (Thread) r;
        if (t.isDaemon()) {
            System.out.println("Starting background thread...");
            new Thread(r);
        } else {
            if (t instanceof EI)
                ((E1) t).run("arg1");
            else
                t.run();
        }
    } else {
        new Thread(r).start();
    }
}
System.out.println("Last line");
```

niz(0):

```
if (r instanceof Thread) {
    Thread t = (Thread) r;
    if (t.isDaemon()) {
        System.out.println("Starting background thread...");
        new Thread(r);
```

Idemo onda na niz(1):

```
} else {
    if (t instanceof EI)
        ((E1) t).run("arg1");
    else
        t.run();
```

što nas dalje vodi u:

```
interface EI extends Runnable {
    default void run(String... args) {
        if (args.length > 0) {
            System.out.println(args[0]);
            new Thread(this).start();
        }
    }
}
```

To se redom izvršava pa idemo u niz(2):

```
for (Runnable r : niz) {
    System.out.println("Checking...");
    if (r instanceof Thread) {
        Thread t = (Thread) r;
        if (t.isDaemon()) {
            System.out.println("Starting background thread...");
            new Thread(r);
        } else {
            if (t instanceof E1)
                ((E1) t).run("arg1");
            else
                t.run();
        }
    } else {
        new Thread(r).start();
    }
}
```

Nakon izvršavanja, idemo u niz(3):

```
public static void main(String argv[]) throws Exception {
    Runnable[] niz = {new E1(), new E1(),
        new Thread(), new E1(), new Thread()};
    System.out.println("First line");
    for (Runnable r : niz) {
        System.out.println("Checking...");
        if (r instanceof Thread) {
            Thread t = (Thread) r;
            if (t.isDaemon()) {
                System.out.println("Starting background thread...");
                new Thread(r);
            } else {
                if (t instanceof E1)
                    ((E1) t).run("arg1");
                else
                    t.run();
            }
        } else {
            new Thread(r).start();
        }
    }
    System.out.println("Last line");
}
```

Nakon toga za niz(4) kao i za niz(2) da bi na kraju imali još:

```
System.out.println("Last line");
```