

# A Java SE/JDK új lehetőségei

Jeszenszky Péter

2024.02.17.

# Tartalom (1)

## Tárgyalt új lehetőségek:

- Lokális változó típus kikövetkeztetés (Java SE 10, 2018)
- HTTP kliens (Java SE 11, 2018)
- Lokális változó szintaxis lambda paraméterekhez (Java SE 11, 2018)
- `switch` kifejezések (Java SE 14, 2020)
- Szövegblokkok (Java SE 15, 2020)
- Mintaillesztés az `instanceof`-hoz (Java SE 16, 2021)
- Rekordok (Java SE 16, 2021)
- Lezárt osztályok (Java SE 17, 2021)
- Mintaillesztés a `switch`-hez (Java SE 21, 2023)
- TODO: Rendezett kollekciók (Java SE 21, 2023)
- TODO: Rekord minták (Java SE 21, 2023)

# Tartalom (2)

Tárgyalt új előzetes lehetőségek:

- Sztring sablonok (Java SE 21, 2023)
- Névtelen minták és változók (Java SE 21, 2023)
- Névtelen osztályok és példány `main` metódusok (Java SE 21, 2023)

# További információk

- *Java Platform, Standard Edition, Java Language Updates, Release 21.* September 2023.  
<https://docs.oracle.com/en/java/javase/21/language/>
- *Java SE Specifications* <https://docs.oracle.com/javase/specs/>
- *OpenJDK: Project Amber.* <https://openjdk.java.net/projects/amber/>

# Programozási nyelvek bőbeszédűsége

- Más programozási nyelvekhez képest a Java-t gyakran bőbeszédűnek tartják.
  - Egy empirikus mód a programozási nyelvek kifejezőerejének és bőbeszédűségének összehasonlításához: *Rosetta Code*  
<https://rosettacode.org/>
    - Példa: a  $\pi$  kiszámítása [https://rosettacode.org/wiki/Arithmetic-geometric\\_mean/Calculate\\_Pi](https://rosettacode.org/wiki/Arithmetic-geometric_mean/Calculate_Pi)
- Számos bemutatott lehetőség célja az, hogy tömörebb szintaxist biztosítsanak egy bizonyos cél eléréséhez.
- Így tehát a *boilerplate* kód írásának szükségességét csökkentik.

# Boilerplate kód (1)

Egy szellemes nem hivatalos definíció Adam Vandenberg-től a *Stack Overflow*-n:

*A "boilerplate kód" olyan láthatólag gyakran ismétlődő kód, mely újra és újra felbukkan valamilyen eredmény elérése céljából, és amelyről látszik, hogy sokkal egyszerűbbnek kellene lennie.*

Forrás: [What is boilerplate code?](#)

- Más néven: ceremoniális/szertartásos kód (*ceremony code, ceremonial code*)

## Boilerplate kód (2)

*Boilerplate* kód példa (HTML):

```
<html>
  <head>
    <meta charset="utf-8">
    <title>
      <!-- A weboldal címe -->
    </title>
  </head>
  <body>
    <!-- Tartalom -->
  </body>
</html>
```

## Boilerplate kód (3)

*Boilerplate* kód példa (Python):

```
def main():  
    print("Hello, World!")  
  
if __name__ == "__main__":  
    main()
```



## Boilerplate kód (4)

*Boilerplate* kód példa (Java):

```
public class Name {  
  
    String firstName;  
    String lastName;  
  
    public Name(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    public String getFirstName() { return firstName; }  
  
    public void setFirstName(String firstName) { this.firstName = firstName; }  
  
    public String getLastName() { return lastName; }  
  
    public void setLastName(String lastName) { this.lastName = lastName; }  
  
}
```

## Boilerplate kód (5)

A *boilerplate* kód mennyiségének csökkentése Java-ban:

- Lombok:

```
@lombok.Data
public class Name {
    String firstName;
    String lastName;
}
```

- Rekordok (Java SE 16):

```
public record Name(String firstName, String lastName) {}
```

# Előzetes lehetőségek (1)

- Egy előzetes lehetőség (*preview feature*) a Java nyelv, a virtuális gép vagy a Java SE API egy új lehetősége, mely pontosan meghatározott, teljesen implementált, de még nem végleges.
- Azért elérhető egy JDK fő kiadásban, hogy a valós használatból kapcsolatos visszajelzéseket generáljon a fejlesztőktől, mely egy jövőbeli Java SE platformban történő állandósulásához vezethet.
- Lásd:
  - *JEP 12: Preview Features* <https://openjdk.java.net/jeps/12>

## Előzetes lehetőségek (2)

- Az előzetes lehetőségek három fajtája: előzetes nyelvi lehetőségek, előzetes VM lehetőségek, előzetes API-k.
- Egy előzetes lehetőség soha nem kísérleti, kockázatos, hiányos vagy instabil.

## Előzetes lehetőségek (3)

- A JDK parancssori eszközöknek (például `java`, `javac`, `jshell`) a `--enable-preview` parancssori opciót kell megadni az előzetes lehetőségek engedélyezéséhez.
- Lásd:
  - *Java Language Updates – Preview Features*  
<https://docs.oracle.com/en/java/javase/21/language/preview-language-and-vm-features.html>

# Előzetes lehetőségek (4)

## Apache Maven:

- Fordításhoz az alábbi módon lehet megadni a `--enable-preview` parancssori opciót a **Maven Compiler Plugin** számára:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.12.1</version>
  <configuration>
    <compilerArgs>--enable-preview</compilerArgs>
  </configuration>
</plugin>
```

- Programok az **Exec Maven Plugin** segítségével történő végrehajtásához a `.mvn/jvm.config` állományban lehet megadni a `--enable-preview` parancssori opciót.

# Előzetes lehetőségek (5)

IntelliJ IDEA:

- Lásd: *Project structure settings – Preview features policy*.  
<https://www.jetbrains.com/help/idea/project-settings-and-structure.html#preview-policy>

# Java SE 10/JDK 10

- Weboldal: <https://openjdk.java.net/projects/jdk/10/>
- Kiadás dátuma: 2018. március 20.
- Főbb új lehetőségek:
  - *JEP 286: Local-Variable Type Inference*  
<https://openjdk.java.net/jeps/286>



## Java SE 10: lokális változó típus kikövetkeztetés (1)

- A nem `null` kezdőértékű lokális változók típus megadása nélkül deklarálhatók a `var` azonosítóval.
- A Java fordító a változó típusát fordítási időben következteti ki.
- A kikövetkeztetési folyamat lényegében az inicializáló kifejezés típusát adja a változónak.
- A `var` azonosító nem kulcsszó, hanem egy fenntartott típusnév. Ez azt jelenti, hogy a `var` használható változó, metódus vagy csomag neveként.

## Java SE 10: lokális változó típus kikövetkeztetés (2)

A `var` fenntartott típusnév az alábbiakhoz használható:

- Inicializált lokális változók
- A *for-each* ciklus indexei
- Hagyományos `for` ciklusban deklarált változók
- A *try-with-resources* utasításban deklarált lokális változók

## Java SE 10: lokális változó típus kikövetkeztetés (3)

- Inicializált lokális változók:

```
var i = 0;
var numberOfItems = 0L;
var epsilon = 1e-10;
var sum = 0.0f;
var failOnError = false;
var greeting = "Hello, World!\n";
var tags = new HashSet<String>();
var lines = Files.readAllLines(Path.of("file.txt"));
```

## Java SE 10: lokális változó típus kikövetkeztetés (4)

- A *for-each* ciklus indexei:

```
for (var name : List.of("Eric", "Kenny", "Kyle", "Stan")) {  
    // ...  
}
```

```
for (var env : System.getenv().keySet()) {  
    // ...  
}
```

```
for (var month : Month.values()) {  
    // ...  
}
```

## Java SE 10: lokális változó típus kikövetkeztetés (5)

- Hagyományos for ciklusban deklarált változók:

```
for (var i = 0; i < 10; i++) {  
    // ...  
}
```

- A *try-with-resources* utasításban deklarált lokális változók:

```
try (var input = new FileInputStream("file.json")) {  
    // ...  
}
```

## Java SE 10: lokális változó típus kikövetkeztetés (6)

A var fordítási hibát okozó nem legális használatai:

```
var a = 1, b = 2;  
var[] c = new int[5];  
var d = {1, 2, 3};  
var e = (int x, int y) -> x + y;  
var f = System.currentTimeMillis;  
var g = System.out::println;
```

# Java SE 10: lokális változó típus kikövetkeztetés (7)

További információk:

- Brian Goetz, Stuart Marks. *Local Variable Type Inference: Frequently Asked Questions*.  
<https://openjdk.java.net/projects/amber/LVTIFAQ.html>
- Stuart W. Marks. *Local Variable Type Inference: Style Guidelines*.  
<https://openjdk.java.net/projects/amber/LVTIstyle.html>

# Java SE 11/JDK 11

- Weboldal: <https://openjdk.java.net/projects/jdk/11/>
- Kiadás dátuma: 2018. szeptember 25.
- Főbb új lehetőségek:
  - *JEP 321: HTTP Client* <https://openjdk.java.net/jeps/321>
  - *JEP 323: Local-Variable Syntax for Lambda Parameters*  
<https://openjdk.java.net/jeps/323>



# Java SE 11: HTTP kliens (1)

- Egy új, a HTTP/2-t és a WebSocket-et implementáló HTTP kliens API, mely helyettesíteni tudja a régi `java.net.HttpURLConnection` API-t.
- Az API modul- és csomagneve `java.net.http`.
- Lásd: <https://docs.oracle.com/en/java/javase/21/docs/api/java.net.http/module-summary.html>

## Java SE 11: HTTP kliens (2)

Példa: időjárási adatok lekérése a [wttr.in](https://wttr.in) szolgáltatótól

```
import java.net.URI;
import java.net.http.*;

var request = HttpRequest.newBuilder()
    .uri(URI.create("https://wttr.in/Debrecen?OAT"))
    .header("Accept-Language", "hu")
    .GET()
    .build();

var response = HttpClient.newHttpClient()
    .send(request, HttpResponse.BodyHandlers.ofString());

int statusCode = response.statusCode();
var contentType = response.headers().map().get("Content-Type");
String body = response.body();
```

## Java SE 11: HTTP kliens (3)

Példa: állománymegosztás a [transfer.sh](#) szolgáltatással

```
import java.net.URI;
import java.net.http.*;
import java.nio.file.Paths;

var request = HttpRequest.newBuilder()
    .uri(URI.create("https://transfer.sh/file.txt"))
    .header("Max-Downloads", "1")
    .PUT(HttpRequest.BodyPublishers.ofFile(Paths.get("file.txt")))
    .build();

var response = HttpClient.newHttpClient()
    .send(request, HttpResponse.BodyHandlers.ofString());

int statusCode = response.statusCode(); // 200
String uri = response.body(); // "https://transfer.sh/1Lf83/file.txt"
```

## Java SE 11: lokális változó szintaxis lambda paraméterekhez

- A `var` fenntartott típusnév használatának megengedése implicit módon típusos lambda kifejezések paramétereinek deklarálásához, mint például

```
(var x, var y) -> x.process(y)
```

- Ez lehetővé teszi annotációk és módosítók használatát lambda paramétereken, mint például

```
(@NonNull var x, @Nullable var y) -> x.process(y)
```

- Egy implicit módon típusos lambda kifejezés minden paraméteréhez kötelező a `var` vagy tilos bármelyikhez is.

# Java SE 12/JDK 12

- Weboldal: <https://openjdk.java.net/projects/jdk/12/>
- Kiadás dátuma: 2019. március 19.
- Főbb új lehetőségek:
  - *JEP 325: Switch Expressions (Preview)*  
<https://openjdk.java.net/jeps/325>

# Java SE 13/JDK 13

- Weboldal: <https://openjdk.java.net/projects/jdk/13/>
- Kiadás dátuma: 2019. szeptember 17.
- Főbb új lehetőségek:
  - *JEP 354: Switch Expressions (Second Preview)*  
<https://openjdk.java.net/jeps/354>
  - *JEP 355: Text Blocks (Preview)* <https://openjdk.java.net/jeps/355>

# Java SE 14/JDK 14

- Weboldal: <https://openjdk.java.net/projects/jdk/14/>
- Kiadás dátuma: 2020. március 17.
- Főbb új lehetőségek:
  - *JEP 305: Pattern Matching for instanceof (Preview)*  
<https://openjdk.java.net/jeps/305>
  - *JEP 359: Records (Preview)* <https://openjdk.java.net/jeps/359>
  - *JEP 361: Switch Expressions* <https://openjdk.java.net/jeps/361>
  - *JEP 368: Text Blocks (Second Preview)*  
<https://openjdk.java.net/jeps/368>

## Java SE 14: switch kifejezések (1)

- A `switch` utasításként és kifejezésként is használható, mindkét formánál használhatók hagyományos `case` címkék (“áteséssel”) vagy az új `case` címkék (“átesés” nélkül). Rendelkezésre áll egy további új utasítás (`yield`) is, mellyel egy érték adható vissza a `switch` kifejezésből.
- Egy `switch` kifejezésnek vagy normális módon egy értékkel kell befejeződnie vagy váratlanul egy kivétel dobásával.



## Java SE 14: switch kifejezések (2)

- A `switch` címke egy új formája (`case ... ->`) került bevezetésre, mely esetenként több vesszővel elválasztott konstanst is megenged.
- A nyíl jobb oldalán csak egy kifejezés, blokk vagy `throw` utasítás megengedett.
- Ha egy címke illeszkedik, akkor csak a nyíl jobb oldalán lévő kifejezés vagy utasítás kerül végrehajtásra, nincs "átesés".
- Példa:

```
var s = switch (items.size()) {  
    case 0 -> "no items";  
    case 1 -> "only one item";  
    default -> String.format("%d items");  
}
```

## Java SE 14: switch kifejezések (3)

- Egy `switch` kifejezés címkéit kimerítő módon kell felsorolni, azaz minden lehetséges esethez kell, hogy legyen egy illeszkedő címke.
- A gyakorlatban ez azt jelenti, hogy egy default záradék szükséges.
- Nem kötelező azonban a default záradék az összes enum konstanst lefedő enum `switch` kifejezéseknél.

## Java SE 14: switch kifejezések (4)

Példa:

```
enum Season {  
    SPRING,  
    SUMMER,  
    AUTUMN,  
    WINTER;  
  
    public static Season of(java.time.Month month) {  
        return switch (month) {  
            case MARCH, APRIL, MAY           -> SPRING;  
            case JUNE, JULY, AUGUST          -> SUMMER;  
            case SEPTEMBER, OCTOBER, NOVEMBER -> AUTUMN;  
            case DECEMBER, JANUARY, FEBRUARY -> WINTER;  
        };  
    }  
}  
  
Season season = Season.of(Month.JUNE); // Season.SUMMER
```

## Java SE 14: switch kifejezések (5)

Példa:

```
import java.time.Duration;
import java.util.concurrent.TimeUnit;

public long durationToTimeUnit(Duration duration, TimeUnit timeUnit) {
    return switch (timeUnit) {
        case DAYS -> duration.toDays();
        case HOURS -> duration.toHours();
        case MICROSECONDS -> duration.toMillis() * 1000;
        case MILLISECONDS -> duration.toMillis();
        case MINUTES -> duration.toMinutes();
        case NANOSECONDS -> duration.toNanos();
        case SECONDS -> duration.toSeconds();
    };
}
```

## Java SE 14: switch kifejezések (6)

- Egy új utasítás (`yield`) került bevezetésre egy érték egy `switch` blokkból történő visszaadására, mely a közrefogó `switch` kifejezés értéke lehet.
- Példa:

```
int j = switch (day) {  
    case MONDAY  -> 0;  
    case TUESDAY -> 1;  
    default      -> {  
        int k = day.toString().length();  
        int result = f(k);  
        yield result;  
    }  
};
```

- A `yield` nem kulcsszó, hanem a `var`-hoz hasonlóan egy korlátozott azonosító.

## Java SE 14: switch kifejezések (7)

- Az alábbi kód is legális:

```
int value = 1;
var result = switch (value) {
    case 1 -> 42;
    case 2 -> Math.PI;
    case 3 -> "Hello, World!\n";
    default -> throw new IllegalArgumentException();
};
```

- A result változó kikövetkeztetett típusa `java.lang.Object`.

# Java SE 15/JDK 15

- Weboldal: <https://openjdk.java.net/projects/jdk/15/>
- Kiadás dátuma: 2020. szeptember 15.
- Főbb új lehetőségek:
  - *JEP 360: Sealed Classes (Preview)* <https://openjdk.java.net/jeps/360>
  - *JEP 375: Pattern Matching for instanceof (Second Preview)*  
<https://openjdk.java.net/jeps/375>
  - *JEP 378: Text Blocks* <https://openjdk.java.net/jeps/378>
  - *JEP 384: Records (Second Preview)* <https://openjdk.java.net/jeps/384>

# Java SE 15: szövegblokkok (1)

- Egy szövegblokk egy többsoros sztring literál, mely bárhol használható, ahol egy közönséges sztring literál.
- További információk:
  - Jim Laskey, Stuart Marks. *Programmer's Guide to Text Blocks*. September 15, 2020.  
<https://docs.oracle.com/en/java/javase/17/text-blocks/>



## Java SE 15: szövegblokkok (2)

Régi stílusú inicializálás:

```
String html = "<!DOCTYPE html>\n" +  
    "<html lang=\"en\">\n" +  
    "  <head>\n" +  
    "    <meta charset=\"utf-8\">\n" +  
    "    <title>Hello, World!</title>\n" +  
    "  </head>\n" +  
    "  <body>\n" +  
    "    <p>Hello, World!</p>\n" +  
    "  </body>\n" +  
    "</html>\n";
```

## Java SE 15: szövegblokkok (3)

Új stílusú inicializálás szövegblokk segítségével:

```
var html = """
    <!DOCTYPE html>
    <html lang="en">
        <head>
            <meta charset="utf-8">
            <title>Hello, World!</title>
        </head>
        <body>
            <p>Hello, World!</p>
        </body>
    </html>
    """;
```

## Java SE 15: szövegblokkok (4)

- A

```
"""
```

```
line 1
```

```
line 2
```

```
line 3
```

```
"""
```

szövegblokk ekvivalens a "line 1\nline 2\nline 3\n" sztring literállal.

- A

```
"""java
```

```
line 1
```

```
line 2
```

```
line 3"""
```

szövegblokk ekvivalens a "line 1\nline 2\nline 3" sztring literállal.

# Java SE 16/JDK 16

- Weboldal: <https://openjdk.java.net/projects/jdk/16/>
- Kiadás dátuma: 2021. március 16.
- Főbb új lehetőségek:
  - *JEP 394: Pattern Matching for instanceof*  
<https://openjdk.java.net/jeps/394>
  - *JEP 395: Records* <https://openjdk.java.net/jeps/395>
  - *JEP 397: Sealed Classes (Second Preview)*  
<https://openjdk.java.net/jeps/397>

# Java SE 16: mintaillesztés az instanceof operátorhoz (1)

- A mintaillesztés lehetővé teszi egy programban komponensek objektumokból történő feltételes kinyerésének tömörebb és biztonságosabb kifejezését.
- Egy minta az alábbiak kombinációja:
  - 1 Egy célra alkalmazható predikátum vagy teszt.
  - 2 Minta változóknak nevezett változók egy halmaza, melyek csak akkor kerülnek kinyerésre a célból, ha a predikátum sikeresen teljesül rá.
- A minta-érzékeny konstrukciók változókat vezethetnek be egy kifejezés kellős közepén.

## Java SE 16: mintaillesztés az instanceof operátorhoz (2)

Mintaillesztés más programozási nyelvekben:

- **C#:** *Pattern Matching* <https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/functional/pattern-matching>
- **Python:**  
[https://docs.python.org/3.11/reference/compound\\_stmts.html#the-match-statement](https://docs.python.org/3.11/reference/compound_stmts.html#the-match-statement)
  - *PEP 634: Structural Pattern Matching: Specification*  
<https://www.python.org/dev/peps/pep-0634/>
  - *PEP 636: Structural Pattern Matching: Tutorial*  
<https://www.python.org/dev/peps/pep-0636/>
- **Scala:** *Scala Documentation – Tour of Scala – Pattern Matching*  
<https://docs.scala-lang.org/tour/pattern-matching.html>
- **Swift:** *Patterns*  
<https://docs.swift.org/swift-book/ReferenceManual/Patterns.html>

## Java SE 16: mintaillesztés az instanceof operátorhoz (3)

- Egy típus minta egy típust meghatározó predikátumból és egyetlen minta változóból áll.
- Az instanceof operátor úgy lett kiterjesztve, hogy csupán egy típus helyet egy típus mintát kapjon.
- Az *instanceof-and-cast* idióma:

```
if (obj instanceof String) {  
    String s = (String) obj;  
    // ...  
}
```

- Az alábbi módon írható mintaillesztéssel:

```
if (obj instanceof String s) {  
    // ...  
}
```

## Java SE 16: mintaillesztés az instanceof operátorhoz (4)

Az equals() metódus hagyományos implementálása:

```
@Override
public boolean equals(Object o) {
    if (! (o instanceof Point)) {
        return false;
    }
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```



## Java SE 16: mintaillesztés az instanceof operátorhoz (5)

Az equals() metódus implementálása mintaillesztéssel:

```
@Override
public boolean equals(Object o) {
    if (o instanceof Point other) {
        return x == other.x && y == other.y;
    }
    return false;
}
```

```
@Override
public boolean equals(Object o) {
    return (o instanceof Point other) && x == other.x && y == other.y;
}
```

## Java SE 16: mintaillesztés az instanceof operátorhoz (6)

Hatáskörkezelés (*flow scoping*):

- Egy minta változó hatásköre a program azon pontjaira terjed ki, ahol a mintaillesztés sikeres lesz és a minta változó értéket kap.
- Példa:

```
if (obj instanceof String s) {  
    // s is in scope here  
} else {  
    // s is not in scope here  
}
```

Az alábbi kód is megengedett:

```
return obj instanceof String s && s.length() > 5;
```

# Java SE 16: mintaillesztés az instanceof operátorhoz (6)

Hatáskörkezelés (*flow scoping*):

- Nem lehetséges és fordítási hibát eredményez egy minta változóra való hivatkozás ott, ahol nem szavatolható a sikeres mintaillesztés.
  - Példa:

```
return obj instanceof String s || s.length() > 5;
```

## Java SE 16: rekordok (1)

- A rekord osztályok nem módosítható adatokat becsomagoló újfajta osztályok.
- A rekord példányok rekord komponenseknek nevezett rögzített értékek egy halmazát ábrázolják.
- A rekord osztályok a `java.lang.Record` osztály alosztályai.
- Egy rekord osztálynak minden egyes komponenséhez van egy implicit módon deklarált lekérdező metódusa. Van implicit módon deklarált konstruktora, `equals()`, `hashCode()` és `toString()` metódusa is.
- A Java SE 21 további nyelvi támogatást vezet be a rekordokhoz, lásd a rekord mintákat (JEP 440).

## Java SE 16: rekordok (2)

Példa:

```
import java.time.Year;
```

```
record LegoSet(String number, Year year, int pieces) {}
```

```
var legoSet = new LegoSet("75211", Year.of(2018), 519);
```

```
System.out.println(legoSet.number());    // 75211
```

```
System.out.println(legoSet.year());      // 2018
```

```
System.out.println(legoSet.pieces());    // 519
```

```
System.out.println(legoSet);
```

```
// "LegoSet[number=75211, year=2018, pieces=519]"
```

## Java SE 16: rekordok (3)

Példa:

```
import java.time.Year;

record LegoSet(String number, Year year, int pieces) {

    LegoSet { // Tömör kanonikus konstruktor
        if (! number.matches("\\d{3,7}")) {
            throw new IllegalArgumentException("invalid number: " + number);
        }
        if (pieces < 0) {
            throw new IllegalArgumentException("pieces must be non-negative");
        }
    }

}
```

# Java SE 17/JDK 17

- Weboldal: <https://openjdk.java.net/projects/jdk/17/>
- Kiadás dátuma: 2021. szeptember 14.
- Főbb új lehetőségek:
  - *JEP 406: Pattern Matching for `switch` (Preview)*  
<https://openjdk.java.net/jeps/406>
  - *JEP 409: Sealed Classes* <https://openjdk.java.net/jeps/409>

# Java SE 17: lezárt osztályok (1)

- A lezárt osztályok és interfészek megszabhatják, hogy mely osztályok vagy interfészek terjeszthetők ki vagy implementálhatják őket.
- Lehetővé teszik olyan osztályhierarchia deklarálását, mely nem nyitott a tetszőleges osztályok általi kiterjesztésre.
- Egy lezárt osztályt vagy interfészt csak azok az osztályok terjeszthetnek ki vagy implementálhatnak, melyek számára ez megengedett.
- Egy osztály a `sealed` módosító a deklarációjára történő alkalmazásával zárható le. Az `extends` és `implements` záradék után a `permits` záradék adja meg azokat az osztályokat, melyek számára megengedett a lezárt osztály kiterjesztése.



## Java SE 17: lezárt osztályok (2)

Példa:

```
public abstract sealed class Shape permits Circle, Rectangle, Square {  
    // ...  
}
```

```
public final class Circle extends Shape {  
    // ...  
}
```

```
public sealed class Rectangle extends Shape  
    permits FilledRectangle, TransparentRectangle {  
    // ...  
}
```

```
public non-sealed class Square extends Shape {  
    // ...  
}
```

## Java SE 17: lezárt osztályok (3)

Egy lezárt osztály három megszorítást szab a megengedett alosztályokra:

- A lezárt osztály és megengedett alosztályai ugyanahhoz a modulhoz kell, hogy tartozzanak, és ha egy névtelen modulban kerülnek deklarálásra, akkor ugyanahhoz a csomaghoz.
- Minden megengedett osztály közvetlenül kell, hogy kiterjessze a lezárt osztályt.
- Minden megengedett alosztályhoz meg kell adni egy azt leíró módosítót, hogy a lezárás hogyan adódik tovább:
  - `final`: Nem terjeszthető ki.
  - `sealed`: Csak a megengedett alosztályok terjeszthetik ki.
  - `non-sealed`: Kiterjeszthetik nem ismert alosztályok, egy lezárt osztály nem akadályozhatja meg ebben a megengedett alosztályait.

## Java SE 17: lezárt osztályok (4)

- Az osztályokhoz hasonlóan egy interfész is a `sealed` módosító az interfészre való alkalmazásával zárható le.
- A szuper interfészeket megadó `extends` záradék után a `permits` záradékkal kerülnek megadásra az implementáló osztályok és az alinterfészek.

## Java SE 17: lezárt osztályok (5)

Példa:

```
sealed interface Celestial permits Planet, Star, Comet {  
    // ...  
}  
  
final class Planet implements Celestial {  
    // ...  
}  
  
final class Star implements Celestial {  
    // ...  
}  
  
final class Comet implements Celestial {  
    // ...  
}
```

# Java SE 18/JDK 18

- Weboldal: <https://openjdk.java.net/projects/jdk/18/>
- Kiadás dátuma: 2022. március 22.
- Főbb új lehetőségek:
  - *JEP 420: Pattern Matching for switch (Second Preview)*  
<https://openjdk.java.net/jeps/420>

# Java SE 19/JDK 19

- Weboldal: <https://openjdk.java.net/projects/jdk/19/>
- Kiadás dátuma: 2022. szeptember 20.
- Főbb új lehetőségek:
  - *JEP 405: Record Patterns (Preview)* <https://openjdk.org/jeps/405>
  - *JEP 427: Pattern Matching for switch (Third Preview)*  
<https://openjdk.java.net/jeps/427>

# Java SE 20/JDK 20

- Weboldal: <https://openjdk.org/projects/jdk/20/>
- Kiadás dátuma: 2023. március 3.
- Főbb új lehetőségek:
  - *JEP 432: Record Patterns (Second Preview)*  
<https://openjdk.org/jeps/432>
  - *JEP 433: Pattern Matching for `switch` (Fourth Preview)*  
<https://openjdk.org/jeps/433>

# Java SE 21/JDK 21

- Weboldal: <https://openjdk.java.net/projects/jdk/21/>
- Kiadás dátuma: 2023. szeptember 19.
- Főbb új lehetőségek:
  - JEP 430: *String Templates (Preview)* <https://openjdk.java.net/jeps/430>
  - JEP 431: *Sequenced Collections* <https://openjdk.java.net/jeps/431>
  - JEP 440: *Record Patterns* <https://openjdk.java.net/jeps/440>
  - JEP 441: *Pattern Matching for switch*  
<https://openjdk.java.net/jeps/441>
  - JEP 443: *Unnamed Patterns and Variables (Preview)*  
<https://openjdk.org/jeps/443>
  - JEP 444: *Virtual Threads* <https://openjdk.org/jeps/444>
  - JEP 445: *Unnamed Classes and Instance Main Methods (Preview)*  
<https://openjdk.org/jeps/445>
  - JEP 446: *Scoped Values (Preview)* <https://openjdk.org/jeps/446>



## Java SE 21: mintaillesztés a switch-hez (1)

- A `switch` utasítások és kifejezések továbbfejlesztése a `case` címkéknél a konstansokon túl lehetővé téve minták és `null` használatát.
- Egy `switch` lényege változatlan: a szelektor kifejezés értéke összehasonlításra kerül a `switch` címkékkel, kiválasztásra kerül a címkék egyike és végrehajtásra kerül a hozzá tartozó kód.
- Azonban a mintás `case` címkéknél a kiválasztást mintaillesztés határozza meg egyenlőség vizsgálat helyett.

## Java SE 21: mintaillesztés a switch-hez (2)

Példa:

```
import com.google.gson.*;

public static int getJsonDepth(JsonElement json) {
    return switch (json) {
        case JsonArray a -> 1 + iteratorToStream(a.iterator())
            .mapToInt(e -> getJsonDepth(e))
            .max()
            .orElse(0);
        case JsonObject o -> 1 + o.entrySet()
            .stream()
            .map(Map.Entry::getValue)
            .mapToInt(e -> getJsonDepth(e))
            .max()
            .orElse(0);
        case JsonPrimitive p -> 0;
        case JsonNull n -> 0;
        default -> throw new AssertionError();
    };
}
```

## Java SE 21: mintaillesztés a switch-hez (3)

Örzött minta case címék:

- Tekintsük az alábbi switch utasítást:

```
static void test(Object o) {  
    switch (o) {  
        case String s -> {  
            if (s.length() == 0) {  
                // Üres sztring kezelése  
            } else {  
                // Nem üres sztring kezelése  
            }  
        }  
        // ...  
    }  
}
```

## Java SE 21: mintaillesztés a switch-hez (4)

Őrzött minta case címkék:

- Az előbbi switch egy érthetőbb és tömörebb formába írható, mely egy őrzött minta case címkét használ:

```
static void test(Object o) {  
    switch (o) {  
        case String s when s.length() == 0 -> // Üres sztring kezelése  
        case String s                        -> // Nem üres sztring kezelése  
        // ...  
    }  
}
```

- Egy őrzött minta case címke (*guarded pattern case label*) `p when e` formájú, ahol `p` egy minta, `e` pedig egy őrfeltételnek nevezett logikai kifejezés.
  - Egy érték illeszkedik a `p when e` őrzött minta case címkére, ha illeszkedik a `p` mintára és az `e` kifejezés értéke igaz.
  - Ha az érték nem illeszkedik `p`-re, akkor az `e` kifejezés nem kerül kiértékelésre.

## Java SE 21: mintaillesztés a switch-hez (5)

Örzött minta címkék:

```
import java.lang.reflect.Array;

static boolean isEmpty(Object object) {
    return switch (object) {
        case null                -> true;
        case CharSequence cs     -> cs.length() == 0;
        case Collection c        -> c.isEmpty();
        case Map m                -> m.isEmpty();
        case Optional o          -> o.isEmpty();
        case OptionalDouble od   -> od.isEmpty();
        case OptionalInt oi      -> oi.isEmpty();
        case OptionalLong ol     -> ol.isEmpty();
        case Object o when o.getClass().isArray() -> Array.getLength(o) == 0;
        default                  -> false;
    };
}
```

## Java SE 21: mintaillesztés a `switch`-hez (6)

A fő tervezési kérdések, amikor a `case` címkék lehetnek minták:

- Kibővített típusellenőrzés
- A `switch` kifejezések és utasítások teljessége
- Minta változó deklarációk hatásköre
- A `null` kezelése
- Hibák

## Java SE 21: mintaillesztés a switch-hez (7)

Kibővített típusellenőrzés:

- A szelektor kifejezés típusa hagyományosan a következők valamelyike kell, hogy legyen egy közöséges switch-nél:
  - egy egész primitív típus (`char`, `byte`, `short` vagy `int`) (a `long` kivételével) vagy a megfelelő csomagoló típusok (`Character`, `Byte`, `Short` vagy `Integer`),
  - `String`,
  - egy `enum` típus.
- Mostantól kezdve a szelektor kifejezés típusa vagy egy egész primitív típus (kivéve `long`) vagy pedig egy referencia típus kell, hogy legyen.

## Java SE 21: mintaillesztés a switch-hez (8)

Kibővített típusellenőrzés: case címkék dominanciája

- Korábban legfeljebb egy case címke vonatkozhatott a szelektor kifejezés egy adott értékére.
- A minta case címkék támogatása azt jelenti, hogy lehetséges az, hogy egynél több case címke vonatkozik a szelektor kifejezés egy adott értékére.



## Java SE 21: mintaillesztés a switch-hez (9)

- Az alábbi kód nem fordul le, mivel az első minta case címke (case CharSequence cs) dominálja a második minta case címkét (case String s), azaz a második minta címke típusa altípusa az első mintáénak:

```
static void error(Object o) {  
    switch (o) {  
        case CharSequence cs ->  
            System.out.println("A sequence of length " + cs.length());  
        case String s ->  
            System.out.println("A string: " + s);  
        default ->  
            System.out.println("Something else");  
    }  
}
```

- A String s case címe elérhetetlen kódnak tekinthető, ezért hibaként van kezelve.

## Java SE 21: mintaillesztés a switch-hez (10)

Kibővített típusellenőrzés: minta címkék dominanciája

- Egy nem őrzött minta case címke dominál egy olyan őrzött minta case címkét, melynek ugyanaz a mintája.
  - Például a `case String s` minta case címe dominálja a `case String s && s.length() > 0` őrzött minta case címkét, mivel az őrzött mintára illeszkedő valamennyi érték illeszkedik a `String s` mintára is.

## Java SE 21: mintaillesztés a switch-hez (11)

Kibővített típusellenőrzés: minta címkék dominanciája

- Egy minta case címke dominálhat egy konstans case címkét.
  - Például a `case Integer i` minta case címke dominálja a `case 42` konstans case címét, a `case E e` minta case címke pedig dominálja a `case A` konstans case címkét, ha `A` az `E` enum típus konstansa.
- Egy őrzött minta case címke dominál egy konstans case címkét, ha dominálja ugyanaz a minta case címke az őrfeltétel nélkül (azaz az őrfeltétel nem kerül ellenőrzésre).
  - Tehát a `case String s when s.length() > 1` őrzött minta case címke dominálja a `case "hello"` minta case címkét, de a `case Integer i when i > 0` is dominálja a `case 0` case címkét.

## Java SE 21: mintaillesztés a switch-hez (12)

Kibővített típusellenőrzés: minta címkék dominanciája

- Ez a case címkék egy egyszerű, kiszámítható és olvasható sorrendjét eredményezi, ahol először a konstans case címkék kell, hogy megjelenjenek az őrzött minta case címkék előtt, melyek pedig a nem őrzött minta case címkék előtt kell, hogy szerepeljenek.
- Példa:

```
switch(o) {  
    case -1, 1                -> ... // Speciális esetek  
    case Integer i when i > 0 -> ... // Pozitív egészek  
    case Integer i           -> ... // Az összes többi egész  
    default                  -> ...  
}
```

## Java SE 21: mintaillesztés a switch-hez (13)

Kibővített típusellenőrzés: minta címkék dominanciája

- Fordítási hiba, ha egy `switch` blokkban egy `case` címkét egy korábbi `case` címke dominál. Ez biztosítja azt, hogy amennyiben egy `switch` blokk csak minta címkéket tartalmaz, akkor azok altípus sorrendben jelennek meg.
  - Lásd például a 73. fólián látható kódot.

## Java SE 21: mintaillesztés a switch-hez (14)

Kibővített típusellenőrzés: minta címkék dominanciája

- Az is fordítási hiba, ha egy `switch` blokkban egynél több olyan címke van, melyre a szelektor kifejezés minden értéke illeszkedik.
  - Ilyenek a `default` és azok a minta `case` címkék, ahol a szelektor kifejezés feltétel nélkül illeszkedik a mintára.
    - Például a `String s` típus mintára feltétel nélkül illeszkedik egy `String` típusú szelektor kifejezés.
- Példa:

```
static void matchAll(String s) {  
    switch(s) {  
        case Object o:  
            System.out.println("An Object");  
            break;  
        default:  
            System.out.println("Something else"); // Fordítási hiba  
    }  
}
```

## Java SE 21: mintaillesztés a switch-hez (15)

A `switch` kifejezések és utasítások teljessége:

- Egy `switch` kifejezésnek kimerítőnek kell lennie, azaz a szelektor kifejezés minden lehetséges értékét kezelni kell a `switch` blokkban.
  - Ez biztosítja azt, hogy egy `switch` kifejezés bármely sikeres kiértékelése egy értéket eredményez.
- A közönséges `switch` kifejezéseknél ezt a `switch` blokkra vonatkozó meglehetősen egyszerű extra feltételek biztosítják.
- A minta `switch` kifejezéseknél ezt a `switch` címkék típus lefedettségének definiálásával érjük el.

## Java SE 21: mintaillesztés a switch-hez (16)

A switch kifejezések és utasítások teljessége:

- Példa (nem fordul le):

```
static int coverage(Object o) {  
    return switch (o) {  
        case String s -> s.length();  
    };  
}
```

- A `case String s` címkére a szelektor kifejezés bármely olyan értéke illeszkedik, melynek típusa `String` vagy annak egy altípusa. Ezért azt mondjuk, hogy ennek a switch címkének a típus lefedettsége a `String` típus.
- Ez a minta switch kifejezés nem teljes, mivel a switch blokkjának típus lefedettsége nem tartalmazza a szelektor kifejezés típusát.



## Java SE 21: mintaillesztés a switch-hez (17)

A switch kifejezések és utasítások teljessége:

- Példa (nem fordul le):

```
static int coverage(Object o) {  
    return switch (o) {  
        case String s -> s.length();  
        case Integer i -> i;  
    };  
}
```

- A switch blokk típus lefedettsége a két switch címke lefedettségének uniója, azaz a String típus altípusainak és az Integer típus altípusainak uniója.
- A típus lefedettség még mindig nem tartalmazza a szelektor kifejezés típusát, így ez a minta switch kifejezés sem teljes és fordítási hibát okoz.

## Java SE 21: mintaillesztés a switch-hez (18)

A switch kifejezések és utasítások teljessége:

- Példa:

```
static int coverage(Object o) {  
    return switch (o) {  
        case String s   -> s.length();  
        case Integer i  -> i;  
        default         -> 0;  
    };  
}
```

- A default típus lefedettsége az összes típus, így ez a példa legális.

## Java SE 21: mintaillesztés a switch-hez (19)

A `switch` kifejezések és utasítások teljessége:

- A teljesség a minta `switch` kifejezésekre és a minta `switch` utasításokra is vonatkozik.
- Visszafelé kompatibilitási okokból minden létező `switch` utasítás változatlanul fordul le.
- Ha azonban egy `switch` utasítás itt részletezett új lehetőségeket használ, akkor a fordító ellenőrizni fogja, hogy teljes-e.
- A teljesség fogalmát bonyolítják a JEP 432-ben bevezetett rekord minták, mivel ezek lehetővé teszik a minták egymásba ágyazását.

## Java SE 21: mintaillesztés a switch-hez (20)

Minta változó deklarációk hatásköre:

- A (JEP 394 által bevezetett) minta változók minták által deklarált lokális változók.
- A minta változó deklarációk a hatáskörkezelés tekintetében szokatlanok (*flow scoping*).

## Java SE 21: mintaillesztés a switch-hez (21)

Minta változó deklarációk hatásköre:

- Egy őrzött case címke mintájában szereplő minta változó deklaráció hatáskörébe tartozik az őrfeltétel, azaz a `when` kifejezés.
- Egy `switch` szabály case címkéjében előforduló minta változó deklaráció hatáskörébe tartozik a nyíl jobb oldalán megjelenő kifejezés, blokk vagy `throw` utasítás.
- Egy `switch`-ben egy utasításcsoport case címkéjében előforduló minta változó deklaráció hatáskörébe tartoznak az utasításcsoport utasításai, feltéve, hogy nem lehetséges “átesés” minta változót deklaráló case címkén.
  - Fordítási hibaként kell kizárni egy minta változót deklaráló case címkén való “átesés” lehetőségét.
  - Tehát például `case Character c: case Integer i: ...` vagy `case Character c, Integer i -> ...` sem megengedett.

## Java SE 21: mintaillesztés a switch-hez (22)

Minta változó deklarációk hatásköre:

- Például az alábbi kód nem fordítható le az “átesés” lehetősége miatt:

```
static void test(Object o) {  
    switch (o) {  
        case Character c:  
            System.out.println("A character " + c);  
        case Integer i: // Error: illegal fall-through to a pattern  
            System.out.println("An integer " + i);  
            break;  
        default:  
            System.out.println("Something else");  
    }  
}
```

## Java SE 21: mintaillesztés a switch-hez (23)

Minta változó deklarációk hatásköre:

- Minta változót nem deklaráló címkén történő átesés megengedett:

```
void test(Object obj) {  
    switch (obj) {  
        case String s:  
            System.out.println("A string");  
        default:  
            System.out.println("Done");  
    }  
}
```

## Java SE 21: mintaillesztés a switch-hez (24)

null kezelése:

- A switch hagyományosan `NullPointerException`-t dob, ha a szelektor kifejezés értéke `null`.
- A `null` case címke került bevezetésre annak az esetnek a kezeléséhez, amikor a szelektor kifejezés értéke `null`.
- A `null`-t kezelő case címke hiányában `NullPointerException` kerül dobásra, ha a szelektor kifejezés értéke `null`, mint korábban is.
- A switch hagyományos szemantikájával való kompatibilitás fenttartása végett a `default` címkére nem illeszkedik a `null` szelektor.



## Java SE 21: mintaillesztés a switch-hez (25)

Példa: null kezelése

```
static String format(Object o) {  
    return switch (o) {  
        case null          -> "null";  
        case Integer i     -> String.format("int %d", i);  
        case Long l        -> String.format("long %d", l);  
        case Double d      -> String.format("double %f", d);  
        default            -> o.toString();  
    };  
}
```

## Java SE 21: mintaillesztés a switch-hez (26)

Példa: null kezelése

```
static int intValue(Object obj) {  
    return switch (obj) {  
        case Boolean b      -> b ? 1 : 0;  
        case Integer i      -> i;  
        case String s       -> Integer.parseInt(s);  
        case null, default -> 0;  
    };  
}
```

# Java SE 21: sztring sablonok (előzetes) (1)

- Számos programozási nyelv (például a C#, JavaScript, Kotlin, Python, Scala) teszi lehetővé a sztring interpolációt a sztring összefűzés alternatívájaként.
- Az interpoláció kifejezések sztring literálokba történő beágyazását jelenti, melyek automatikusan kiértékelésre és az értékükkel való helyettesítésre kerülnek.
- Az interpoláció veszélyes lehet, mivel ki van téve a befecskendezéses támadásoknak (például SQL befecskendezés).

## Java SE 21: sztring sablonok (előzetes) (2)

Példa SQL befecskendezésre:

```
private updatePassword(String username, String newPassword) {  
    final String sql = String.format(  
        "UPDATE users SET password = '%s' WHERE username = '%s'",  
        newPassword,  
        username  
    );  
    // SQL utasítás végrehajtása  
}
```

Egy "jabba" nevű rosszindulatú támadó az összes felhasználó jelszavát átállíthatja a felhasználói nevét "jabba" OR username <> "jabba" módon megadva.

## Java SE 21: sztring sablonok (előzetes) (3)

- A sablon kifejezések (*template expressions*) egy biztonságos mechanizmust nyújtanak egy eredmény létrehozásához literális szöveg és kifejezések értékeinek kombinálása révén.
  - Az eredmény általában egy sztring, mely sztring interpolációt jelent.
- Egy sablon egy sztring sablon vagy pedig egy szövegblokk sablon.
  - Egy sztring sablon (szövegblokk sablon) egy sztring literálhoz (szövegblokkhoz) hasonlít, azonban egy vagy több beágyazott kifejezést tartalmaz.
- A szöveg és a kifejezések értékeinek kombinálása egy sablon feldolgozóhoz (*template processor*) van delegálva.
  - A sablon feldolgozók nincsenek sztringek létrehozására korlátozva, visszaadhatnak sztringektől különböző értékeket.

# Java SE 21: sztring sablonok (előzetes) (4)

- A Java platformban definiált szabványos sablon feldolgozók:
  - STR: `java.lang.StringTemplate.STR`
  - FMT: `java.util.FormatProcessor.FMT`
  - RAW: `java.lang.StringTemplate.RAW`
- További sablon feldolgozók hozhatók létre a `java.lang.StringTemplate.Processor` funkcionális interfész implementálásával.

## Java SE 21: sztring sablonok (előzetes) (5)

Példa:

```
var width = 1024;  
var height = 768;  
var s = STR."\{width} * \{height} = \{width * height}";  
System.out.println(s);
```

## Java SE 21: sztring sablonok (előzetes) (6)

Többsoros sablon kifejezések:

```
var now = LocalDate.now();
var s = STR."""
{
    "meaningOfLife": \{
        2 * 3 * 7 // Douglas Adams szerint
    },
    "java": {
        "vendor": "\{System.getProperty("java.vendor")}",
        "version": "\{System.getProperty("java.version")}"
    },
    "date": "\{now}",
    "isLeapYear", \{now.isLeapYear()}
}
""";
System.out.println(s);
```



## Java SE 21: sztring sablonok (előzetes) (7)

Többsoros sablon kifejezések:

```
import static java.util.Formatter.FMT;

var r = 3.0;
var s = FMT."""
{
    "circle": {
        "radius": %.4f\{r},
        "area": %.4f\{r * r * Math.PI},
        "perimeter": %.4f\{2 * r * Math.PI}
    }
}
""";

System.out.println(s);
```

# Java SE 21: sztring sablonok (előzetes) (8)

További információk:

- Jim Laskey, Brian Goetz. *String Tapas Redux: Beyond Mere String Interpolation*. September 2021.  
<https://openjdk.org/projects/amber/design-notes/templated-strings>

# Java SE 21: névtelen változók (előzetes) (1)

- Egy aláhúzójel egy névtelen változót jölöl, mely csak inicializálható, de nem használható.
- A névtelen változók a kód karbantarthatóságát javítják.
- Olyan esetekben hasznosak, melyekben fontosabb egy utasítás mellékhata, mint maga az eredmény.
- Egy aláhúzójel egy lokális változó, kivétel paraméter vagy lambda paraméter deklarálására használható.
- Egy névtelen változó soha nem fed el másik változót, mivel nincs neve, így tehát több névtelen változó is deklarálható ugyanabban a blokkban.

## Java SE 21: névtelen változók (előzetes) (2)

Példák:

- ```
long count = 0;
for (var _ : items) {
    count++;
}
```

## Java SE 21: névtelen változók (előzetes) (3)

Példák:

- ```
Queue q = ...;
while (q.size() >= 3) {
    var x = q.remove();
    var y = q.remove();
    var _ = q.remove();
    // ... new Point(x, y) ...
}
```
- ```
Queue q = ...;
while (q.size() >= 3) {
    var x = q.remove();
    var _ = q.remove();
    var _ = q.remove();
    // ... new Point(x, 0) ...
}
```

## Java SE 21: névtelen változók (előzetes) (4)

Példák:

- ```
String s = ...;
try {
    int i = Integer.parseInt(s);
    // ...
} catch (NumberFormatException _) {
    System.out.println("Invalid number: " + s);
}
```
- ```
var map = stream.collect(Collectors.toMap(
    Function.identity(),
    _ -> 0 // <= névtelen lambda paraméter
));
```

# Java SE 21: névtelen osztályok és példány main metódusok (1)

A JEP fő célja, hogy lehetővé tegye a diákok számára az első Java programjaik megírását úgy, hogy ahhoz nem szükséges érteniük a nagy programokhoz tervezett nyelvi lehetőségeket (“programming in the small”).

# Java SE 21: névtelen osztályok és példány `main` metódusok (2)

Példány `main` metódusok:

- A Java programok elindítására szolgáló protokoll továbbfejlesztése példány `main` metódusokkal.
- A példány `main` metódusok nem statikusak, nem szükséges nyilvánosaknak lenniük, és az sem szükséges, hogy egy `String[]` típusú paraméterük legyen.



## Java SE 21: névtelen osztályok és példány main metódusok (3)

Példány main metódus példa:

```
class Hello {  
    void main() {  
        System.out.println("Hello, World!");  
    }  
}
```

# Java SE 21: névtelen osztályok és példány main metódusok (4)

Példány main metódus példa:

```
class Hello {  
  
    String greeting;  
  
    Hello() {  
        greeting = STR."Hello, \{System.getProperty("user.name")}\!";  
    }  
  
    void main() {  
        System.out.println(greeting);  
    }  
}
```

## Java SE 21: névtelen osztályok és példány main metódusok (5)

Egy osztály elindításakor az indítási protokoll az alábbi metódusok közül az elsőt választja meghívásra:

- 1 Az elindított osztályban deklarált nem `private` (azaz `public`, `protected` vagy csomagszintű) hozzáférésű `static void main(String[] args)` metódus.
- 2 Az elindított osztályban deklarált nem `private` hozzáférésű `static void main()` metódus.
- 3 Az elindított osztályban deklarált vagy egy ősosztálytól örökölt nem `private` hozzáférésű `void main(String[] args)` példánymetódus.
- 4 Az elindított osztályban deklarált vagy egy ősosztálytól örökölt nem `private` hozzáférésű `void main()` példánymetódus.

# Java SE 21: névtelen osztályok és példány `main` metódusok (6)

## Névtelen osztályok:

- A névtelen osztályok a névtelen csomagokhoz és modulokhoz hasonlóak.
- Amikor a Java fordító egy forrásállományban olyan metódusokat vagy mezőket talál, melyek nem egy osztálydeklaráció részei, akkor azokról implicit módon azt feltételezi, hogy egy felső szintű névtelen osztály tagjai.
- Egy névtelen osztály a névtelen csomaghoz tartozik. A névtelen csomagban több névtelen osztály is lehet.
- Egy névtelen osztály majdnem pontosan olyan, mint egy explicit módon deklarált osztály. A tagjainak például ugyanazok a módosítói lehetnek (például `private` és `static`).

# Java SE 21: névtelen osztályok és példány `main` metódusok (7)

## Névtelen osztályok:

- Különbségek az explicit módon deklarált osztályoktól:
  - Egy névtelen osztály nem kiterjeszthető, nem implementálhat semmilyen interfészt és az `Object` osztályon kívül nem terjeszthet ki semmilyen osztályt.
  - Kód nem hivatkozhat egy névtelen osztályra a nevével, tehát az nem példányosítható közvetlenül.
  - Egy névtelen osztálynak van paraméter nélküli alapértelmezett konstruktora, azonban nem lehet más konstruktora.
- Egy olyan osztály csak egy különálló programként vagy egy program belépési pontjaként hasznos. Tehát kell, hogy legyen `main` metódusa.

## Java SE 21: névtelen osztályok és példány main metódusok (8)

Példa névtelen osztályra (Hello.java):

```
void main() {  
    System.out.println("Hello, World!");  
}
```

## Java SE 21: névtelen osztályok és példány main metódusok (9)

Példa névtelen osztályra (Hello.java):

```
String greeting = "Hello, World!";
```

```
void main() {  
    System.out.println(greeting);  
}
```

## Java SE 21: névtelen osztályok és példány main metódusok (10)

Példa névtelen osztályra (Hello.java):

```
String sayHelloTo(String who) {  
    return STR."Hello, \{who}!";  
}
```

```
void main() {  
    var s = sayHelloTo("World");  
    System.out.println(s);  
}
```