Tóm tắt điều hành

JEP 8209434 (“Concise Method Bodies”) là một đề xuất bổ sung cú pháp mới để viết thân phương thức ngắn gọn hơn, tương tự biểu thức lambda/method reference, nhằm giảm boilerplate trong các phương thức đơn giản. Tuy nhiên, nó vẫn ở trạng thái **Draft**, chưa có tiến triển rõ rệt kể từ lần cập nhật cuối và gặp nhiều thách thức kỹ thuật (overload ambiguity, tương thích, công cụ hỗ trợ). Việc đưa JEP này vào thực tế cần nhiều kiểm thử, giai đoạn preview và sự hỗ trợ từ công cụ như IDE, máy biên dịch và VM.

A. Chuẩn bị (thu thập & xác minh)

1. **Văn bản dự thảo chính thức** JEP chính thức có thể xem tại: “JEP draft: Concise Method Bodies” trên trang OpenJDK. Đây là tài liệu gốc, trạng thái **Draft**, cùng với phần “Discussion” dẫn đến mailing list Amber-dev.
2. **Phiên bản / ngày cập nhật / khác biệt**
    - **Ngày khởi tạo**: 13 tháng 8, 2018 (tại phần “Created”)
    - **Cập nhật gần nhất**: 25 tháng 3, 2019 (phần “Updated”)
    - Trạng thái: **Draft** (không có mục tiêu “Targeted” hay “Final”)
    - Không thấy phiên bản khác ghi rõ (ví dụ “Preview”) trong tài liệu; do đó có vẻ JEP vẫn giữ nguyên form draft ban đầu.
    - Không có bản cập nhật tiếp theo sau 2019 trong trang JEP (ít nhất tại tài liệu chính thức).

    Tóm tắt khác biệt / ghi chú: Vì JEP này không có phiên bản “candidate” hay “targeted” được ghi rõ, nên không thể so sánh nhiều phiên bản. Trạng thái gần như “đang nằm im” kể từ 2019 (theo các thảo luận cộng đồng).

B. Phân tích theo cấu trúc yêu cầu

1. Tên và số JEP

* **Tên đầy đủ**: Concise Method Bodies
* **Số hiệu**: 8209434
* **Trạng thái**: Draft
* **Ngày cập nhật gần nhất**: 25 tháng 3, 2019

---

2. Mục tiêu / Motivation

* **Vấn đề muốn giải quyết** Nhiều phương thức trong Java thực chất chỉ trả về một biểu thức hoặc chuyển tiếp (delegate) sang phương thức khác. Việc viết chúng bằng cú pháp đầy đủ với `{ ... return ... }` là verbose (rườm rà) và gây “line noise”. JEP muốn mở rộng cú pháp để những phương thức đơn giản này có thể viết ngắn gọn, tương tự cách lambda và method reference được viết trong Java.
* **Hạn chế hiện tại**

- Ngôn ngữ Java hiện chỉ cho phép thân phương thức dạng khối `{ }`, bất kể thân đơn giản hay phức tạp.
- Dù lambda có biểu thức đơn giản hay method reference, nhưng phương thức thông thường chưa được hưởng cú pháp tương đương.
- Khi phương thức chỉ delegate, lập trình viên phải lặp lại danh sách tham số, thậm chí lỗi khi gõ sai tham số hoặc thứ tự — JEP muốn loại bớt việc lặp lại này.
- Việc copy-paste nhiều getter, wrapper là boilerplate mà JEP hướng giảm.

* **Phân tích thuyết phục**
  - Lý luận của JEP là hợp lý: nếu lambda/method reference có thể viết gọn thì cùng kiểu cú pháp cũng nên áp dụng cho phương thức thông thường.
  - Việc giảm boilerplate, tăng readability cho các phương thức "một biểu thức" là lợi thế rõ ràng.
  - Tuy nhiên, bản thảo chưa chứng minh rõ về các tình huống phức tạp như overload ambiguity, tương tác với generic, exception handling, debug, stacktrace, v.v.
  - Ngoài ra, với nhiều ưu tiên khác của ngôn ngữ Java (performance, ổn định), có thể JEP này bị đánh giá là "nice-to-have" hơn là "must-have". Theo thảo luận cộng đồng, JEP này "nằm trên kệ" từ năm 2019 do "needs more work".

---

3. Chi tiết kỹ thuật

a) Thay đổi cú pháp

* Thêm hai cú pháp mới cho thân phương thức:
  - **Single-expression form**: sử dụng `->`

```
int length(String s) -> s.length();
```

  - **Method-reference form**: sử dụng `=`

```
int length(String s) = String::length;
```

* Những phương thức này chỉ hợp lệ với các phương thức **non-abstract, non-native** trong lớp hoặc interface. Không áp dụng cho constructor, instance initializers, static initializers.
* Phương thức reference có thể là static, unbound, bound, `super` reference, constructor reference, array reference.
* Khi phương thức dùng cú pháp gọn, phần **signature** của phương thức cung cấp "structural target type" để bind biểu thức hoặc method reference.
* Cú pháp mới không thay đổi cách override, overload hay inheritance hoạt động (cách thức áp dụng semantic nói là không ảnh hưởng).

b) Thay đổi ngữ nghĩa

* Khi phương thức dùng cú pháp `=` (method reference form), compiler sẽ thực hiện *overload resolution* đối với method reference, tìm phương thức phù hợp với kiểu target (signature). (Suy luận dựa trên nội dung JEP)

* Trong `->` form, biểu thức đơn sẽ được tách ra và sử dụng như `return expression`.
* Cú pháp mới không làm thay đổi nghĩa inheritance / overriding — nếu subclass override phương thức, vẫn dùng cú pháp gọn hoặc cú pháp truyền thống.
* Về generic, type inference, wildcards, boxing/unboxing — bản JEP không trình bày chi tiết. (Thiếu thông tin)
* Các trường hợp exception, try/catch, throw, multi-statement method — không sử dụng cú pháp gọn mà vẫn dùng cú pháp khối `{ }`.

c) Ảnh hưởng đến bytecode / JVM

* Bản JEP không chỉ rõ cách chuyển cú pháp gọn xuống bytecode.
* Suy luận: cú pháp `->` sẽ được compiler (javac) chuyển thành same bytecode như phương thức bình thường với `return` và body expression.
* Với `=` (method reference), compiler có thể chuyển thành *invokedynamic*-based method reference linkage hoặc tạo một method handle wrapper (tùy cách triển khai).
* Không có yêu cầu thay đổi metadata lớp, annotation hay cấu trúc classfile trong bản JEP.
* CMB không yêu cầu thay đổi VM hoặc classfile format, chủ yếu là thay đổi compiler / frontend.

d) Tác động đến toolchain

* **Javac**: cần mở rộng parser, type checker, overload resolution, error messages, warnings.
* **IDEs (IntelliJ, Eclipse, VS Code, NetBeans)**: cần hỗ trợ cú pháp mới (highlight, autocomplete, refactor, error detection).
* **Debuggers / stacktrace**: nếu phương thức gọn, tên phương thức, các dòng (line numbers) và biểu thức nội dung cần ánh xạ đúng để stacktrace đọc dễ hiểu. Có nguy cơ mất thông tin dòng lệnh nếu mapping không tốt.
* **Frameworks / annotation processors / Lombok / ASM / Bytecode tools**: những tool đọc AST hoặc source, nếu không cập nhật cú pháp mới, có thể không nhận ra method gọn; Lombok có thể cần tương tác với cách viết method gọn; các công cụ bytecode (ASM) nếu làm transform dựa vào source mapping cần cập nhật.

e) Tương thích ngược

* **Trường hợp code cũ bị thay đổi nghĩa**: nếu compiler mặc định chấp nhận `->` hoặc `=` trong method bodies, có thể gây parse ambiguty nếu code cũ dùng `->` hoặc `=` trong các context khác (ví dụ biểu thức lambda) — nhưng JEP không mô tả chi tiết về conflict grammar.
* **Không biên dịch được**: nếu code dùng `->` hoặc `=` trong context chưa được hỗ trợ, compiler hiện tại sẽ lỗi. Khi JEP mới được áp dụng, code legacy vẫn compile như trước nếu dùng cú pháp truyền thống.
* Ví dụ giả định:

  Trước (legacy):

```
int compute(int x) {
return x * 2;
}
```

Sau (nếu compile được CMB):

```
int compute(int x) -> x * 2;
```

Nếu compiler chưa hỗ trợ hoặc toolchain lỗi, code sau có thể biên dịch lỗi.

* Nếu có overload:

```
int f(int x) -> g(x);
int f(String s) -> g(s);
```

Có thể conflict nếu g là method overloaded — cần rõ cách resolution.

f) Hiệu năng

* **Compile-time**: parser và resolver thêm phức tạp, nhưng tác động nhỏ so với tổng biên dịch project lớn.
* **Runtime**: không có overhead thêm nếu biên dịch thành bytecode tương đương phương thức truyền thống.
* **Thử nghiệm cần làm**:
    - So sánh thời gian compile các dự án lớn với/không có CMB hỗ trợ.
    - So sánh bytecode generated & kích thước class nếu dùng method reference form.
    - Kiểm tra overhead liên quan đến link method reference (dùng invokedynamic) nếu áp dụng.
    - Đo hiệu năng gọi phương thức nội dung gọn so với phương thức truyền thống trong các tình huống nóng (hot path).

---

4. Ảnh hưởng

* **Lợi ích với lập trình viên**
    - Giảm boilerplate và noise cho các phương thức đơn giản (getter, wrapper, delegate).
    - Mã ngắn gọn, ý định rõ ràng hơn.
    - Khuyến khích refactoring: dễ tách biểu thức ra thành phương thức riêng mà ít code phụ trợ.
    - Khi dùng method reference form, tránh lỗi khi gõ lại danh sách tham số.
* **Rủi ro & hạn chế**
    - Cú pháp mới có thể gây mơ hồ — người đọc phải hiểu cú pháp `->`, `=` trong context method bodies.
    - Debug, stacktrace, exception trace có thể bị ảnh hưởng nếu mapping dòng/biểu thức không rõ.
    - Công cụ (IDEs, plugin, refactoring tools) phải cập nhật; nếu không, dev có trải nghiệm lỗi, báo lỗi sai.

- Overload resolution phức tạp, ambiguities dễ xảy ra, đặc biệt với generic và method references.
- Nếu lạm dụng, mã có thể trở nên "nén" quá mức, giảm khả năng đọc khi biểu thức dài.
- Vì JEP vẫn draft và ít được ưu tiên, có thể nó không được thực hiện trong tương lai gần.

* **Hướng di chuyển & chuyển đổi code**
  - Có thể cung cấp lint rule hoặc warning khi có thể chuyển sang cú pháp gọn.
  - Cung cấp công cụ refactor để chuyển các phương thức single-expression hoặc delegate sang CMB.
  - Giai đoạn đầu nên cung cấp preview flag / opt-in — dev bật flag để dùng cú pháp mới.
  - Giao tiếp với style guide, team quy định khi nào nên dùng CMB (chỉ cho phương thức đơn giản, không exception, không logic phức).
  - Khi codebase lớn, có thể chuyển dần theo module hoặc theo package, không ép toàn bộ cùng lúc.

* **Đề xuất quy trình triển khai**
  1. **Preview feature**: triển khai cú pháp mới dưới flag `--enable-preview` hoặc tương tự trong JDK thử nghiệm.
  2. **Thử nghiệm & đo đạc**: tích hợp JEP vào bản thử nghiệm, cho dev dùng thử, thu feedback.
  3. **Migration tools & lint/refactor**: cung cấp plugin chuyển đổi code tự động từ cú pháp truyền thống sang CMB nếu hợp lý.
  4. **Xếp vào roadmap JDK**: nếu feedback tốt, chuyển từ Draft → Candidate → Targeted → Delivered.

---

5. Ví dụ minh họa (trước / sau)

Dưới đây là các ví dụ giả định minh hoạ cách cú pháp CMB có thể hoạt động:

a) Method đơn giản rút gọn

Trước:

```
public int add(int x, int y) {
    return x + y;
}
```

Sau (CMB):

```
public int add(int x, int y) -> x + y;
```

Giải thích: compiler ghép `->` thành `return x + y`.

b) Ví dụ mơ hồ / overload

```
public int f(int x) -> g(x);
public long f(long x) -> g(x);
```

Nếu `g(int)` và `g(long)` đều tồn tại, compiler cần rõ ràng kiểu target để chọn overload `g`.

c) Debug / stacktrace

Trước:

```
int compute(int x) {
    return process(x);
}
```

Nếu throw exception trong `process`, stacktrace sẽ hiển thị `compute` → `process`. **Sau (CMB):**

```
int compute(int x) -> process(x);
```

Stacktrace cần đúng mapping dòng (source line) cho `compute` và `process` để dev dễ hiểu.

d) Ví dụ thực tế API nhỏ

```
class MyList<E> implements List<E> {
    private final List<E> inner;

    public int size() {
        return inner.size();
    }
    public E get(int idx) {
        return inner.get(idx);
    }
}
```

Sau dùng CMB:

```
class MyList<E> implements List<E> {
    private final List<E> inner;

    public int size() -> inner.size();
    public E get(int idx) -> inner.get(idx);
}
```

Hoặc method reference form:

```
public int size() = inner::size;
public E get(int idx) = inner::get;
```

Bytecode generated tương đương như phương thức bình thường delegate.

C. Đánh giá phản biện & kiểm thử

1. Bẫy rủi ro kỹ thuật / điểm còn bỏ ngỏ

1. **Xung đột ngữ pháp** — cú pháp `->` hoặc `=` có thể conflict với expression lambda hoặc other language constructs. *Kiểm chứng:* thử parsing code phức tạp (nested lambdas, annotation, generic) và kiểm tra lỗi grammar.
2. **Overload ambiguity / method reference resolution** — khi method reference form hoặc `->` dùng overloaded methods, không rõ chọn cái nào. *Kiểm chứng:* test các tình huống overload

phương thức delegate, generic, wildcards, gán method reference, và xem compiler chọn đúng hay báo lỗi.

3. **Hỗ trợ generic / wildcard / type inference** — bản JEP không đề cập rõ cách xử lý các trường hợp generic phức tạp. *Kiểm chứng:* viết phương thức generic / wildcards với cú pháp gọn và so sánh việc biên dịch với code bình thường.
4. **Xử lý exception / throws / control flow** — nếu method body cần try/catch hoặc throw, cú pháp gọn không hỗ trợ; hoặc nếu muốn mở rộng syntax để allow expression throw, cần quy định rõ. *Kiểm chứng:* viết phương thức đơn giản có throw hoặc try/catch và xem compiler xử lý thế nào.
5. **Stacktrace / debug mapping** — nếu dòng số (line number) hoặc mapping source -> bytecode không đúng, dev khó debug. *Kiểm chứng:* đưa breakpoint, in stacktrace khi exception xảy ra, kiểm tra tên phương thức và dòng trong source.
6. **Tác động lên toolchain / IDE / plugin** — nếu IDE không hỗ trợ, dev sẽ gặp lỗi hiển thị, refactor bị lỗi. *Kiểm chứng:* thử mở code dùng CMB trong IntelliJ / Eclipse / VS Code, xem cú pháp, autocompletion, refactor, warning.
7. **Adoption / fragmentation cú pháp** — nếu dev dùng hỗn hợp cú pháp truyền thống và CMB, có thể gây sự thiếu đồng nhất, khó đọc. *Giải pháp:* đưa quy định style, lint rule để khuyến khích hoặc hạn chế việc dùng CMB trong những trường hợp phù hợp.

2. Năm test case cụ thể

Dưới đây là năm test case đầu vào & hành vi mong đợi mà OpenJDK / nhóm ngôn ngữ nên kiểm thử:

#	Phương thức dùng CMB	Hành vi mong đợi
1	`public int inc(int x) -> x + 1;`	Biên dịch thành phương thức trả `(x + 1)` tương đương `return x + 1`
2	`public String name() = this::getName;`	Compiler bind method reference đúng, gọi `getName()`
3	Overload conflict: `public int f(int x) -> g(x);` `public long f(long x) -> g(x);`	Compiler báo lỗi ambiguity nếu không rõ g chọn overload nào

4	`public void doit() -> throw new RuntimeException();`	Biến cú pháp gọn không cho phép throw expression; compiler báo lỗi hoặc yêu cầu dùng khối `{}`
5	Dùng trong generic: `public <T> T ident(T x) -> x;`	Hỗ trợ binding generic, biên dịch thành `return x;` đúng với kiểu generic.

3. Ba bước triển khai thử nghiệm / preview

1. **Bản JDK preview** — tích hợp JEP trong một bản thử nghiệm JDK (Early Access), bật bằng flag `--enable-preview` để dev có thể thử nghiệm.
2. **Migration plugin / refactor tool** — cung cấp công cụ (ví dụ plugin IntelliJ hoặc `jdeprscan` extension) tự động chuyển các method single-expression hoặc delegate sang CMB nếu hợp lý và đưa cảnh báo (lint).
3. **Giai đoạn áp dụng thử trong các module OpenJDK** — áp dụng CMB vào các lớp wrapper đơn giản trong JDK (Collections, Streams, Optional, v.v.), thu feedback, đo performance, kiểm tra stacktrace, debug mapping.

D. Nguồn trích dẫn chính

1. **JEP draft page**: “JEP draft: Concise Method Bodies” — phần Summary, Description, Motivation, Status.
2. **JEP Index**: liệt kê 8209434 trong Draft JEPs.
3. **Baeldung – Project Amber & JEP 8209434**: mô tả ngắn JEP và trạng thái.
4. **JavaCodeGeeks - Composition Simplified với JEP Draft**: ví dụ delegate method reference, comparisons.

```
public class Main {

    private String name;

    // Getter truyền thống (Trước)

    public String getName() {
```

```
        return name;
    }
    // Sau (CMB):
    // public String getName() -> name;
    // Hoặc method reference form:
    // public String getName2() = this::name;  // giả sử this::name là hợp lệ (ý tưởng)

    // a) Method đơn giản rút gọn
    // Trước:
    public int add(int x, int y) {
        return x + y;
    }
    // Sau (CMB):
    // public int add(int x, int y) -> x + y;

    // b) Ví dụ mơ hồ / overload
    // Trước:
    public int f(int x) {
        return g(x);
    }
    public long f(long x) {
```

```
        return g(x);
    }
    private int g(int x) {
        return x + 1;
    }
    private long g(long x) {
        return x + 1L;
    }
    // Sau (CMB):
    // public int f(int x) -> g(x);
    // public long f(long x) -> g(x);

    // c) Debug / stacktrace mapping
    // Trước:
    int compute(int x) {
        return process(x);
    }
    int process(int x) {
        // Ví dụ minh họa: cố tình tạo lỗi để
stacktrace thể hiện compute -> process
        return 10 / (x - x); // sẽ ném
ArithmeticException khi x == x
```

```
    }
    // Sau (CMB):
    // int compute(int x) -> process(x);
    // Yêu cầu của CMB: mapping dòng nguồn cần tương
ứng để stacktrace dễ hiểu

    // d) Ví dụ thực tế API nhỏ (delegate)
    // Trước:
    static class MyList<E> {
        private final java.util.List<E> inner;

        MyList(java.util.List<E> inner) {
            this.inner = inner;
        }

        public int size() {
            return inner.size();
        }
        public E get(int idx) {
            return inner.get(idx);
        }
```

```
        // Sau dùng CMB:
        // public int size() -> inner.size();
        // public E get(int idx) -> inner.get(idx);

        // Hoặc method reference form:
        // public int size() = inner::size;
        // public E get(int idx) = inner::get;
    }

    public static void main(String[] args) {
        Main demo = new Main();
        demo.name = "Concise Method Bodies Demo";

        // Minh họa a)
        int sum = demo.add(3, 4); // 7
        System.out.println("add(3,4) = " + sum);

        // Minh họa b)
        System.out.println("f(10) = " + demo.f(10));
        System.out.println("f(10L) = " + demo.f(10L));

        // Minh họa d)
```

```
        java.util.List<String> base = new
java.util.ArrayList<>();
        base.add("A");
        base.add("B");
        MyList<String> myList = new MyList<>(base);
        System.out.println("myList.size() = " +
myList.size());
        System.out.println("myList.get(1) = " +
myList.get(1));

        // Minh họa c) (bắt lỗi để không dừng chương
trình)
        try {
            demo.compute(0);
        } catch (ArithmeticException ex) {
            System.out.println("Caught exception from
compute -> process: " + ex);
        }

        // Getter truyền thống demo
        System.out.println("getName() = " +
demo.getName());
    }
```

```
}
```