Warsaw, April 2025

Modern Java

Ron Veen

Agenda

1. Virtual threads
2. Structured Concurrency
3. Scoped Values

Faster

4. Record patterns
5. Unnamed patterns and variables
6. Unnamed patterns
7. Primitive patterns
8. Sequenced Collections
9. Statements before super
10. Stream Gatherers

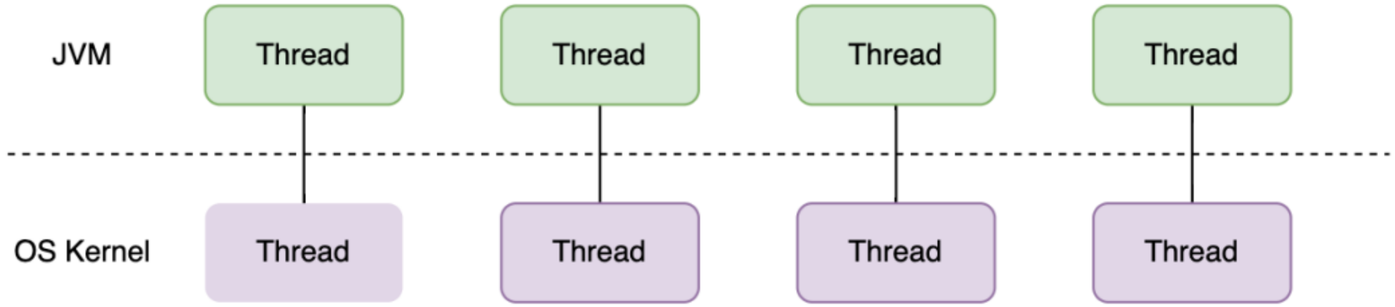
Better

11. Unnamed class and instance main methods.
12. Module Imports
13. Launch multi-file source code programs
14. Markdown Documentation

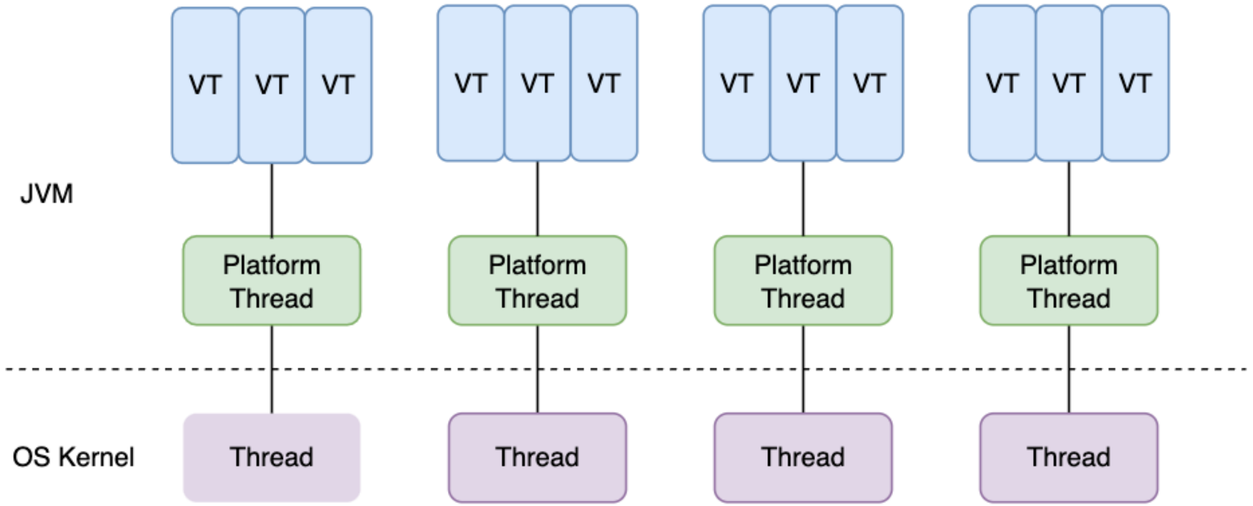
Easier

Virtual Threads

Java Threads



Virtual Threads



Factory methods

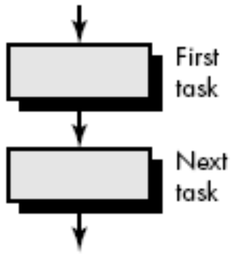
```
public class VirtualThread {
    Runnable runnable = () -> {

        System.out.println("I am a virtual thread: " +      Thread.currentThread().isVirtual());
    };

    public static void main(String... args) {
        new VirtualThread().run();
    }

    private void run() {
        Thread.ofVirtual().start(runnable);
        Thread.ofPlatform().start(runnable);
    }

}
```

Virtual Thread Per Task Executor

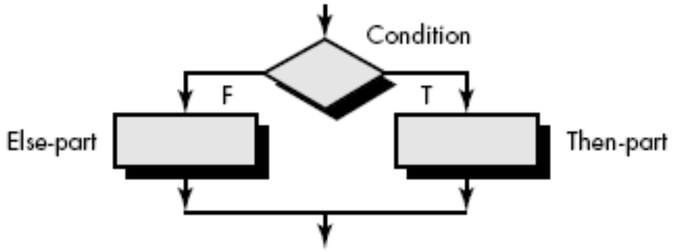

```
    private void run() {
        try(var executor = Executors.newVirtualThreadPerTaskExecutor()) {
            executor.submit(runnable);
        }
    }
```

Structured Concurrency

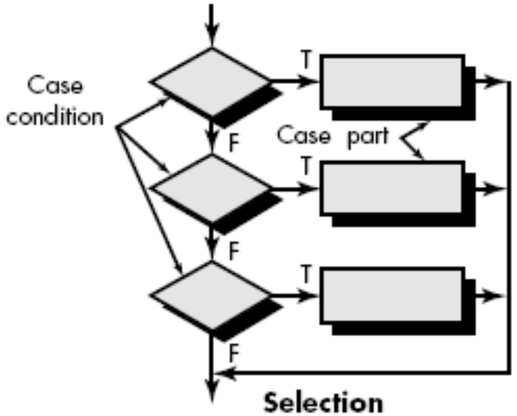
Structured Concurrency



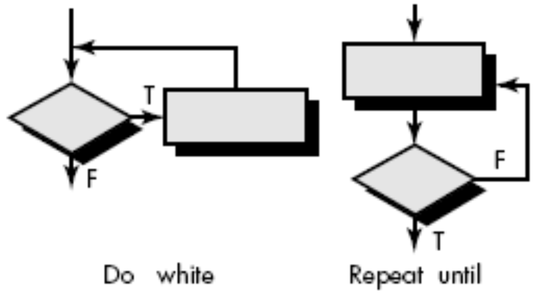
Sequence



If-then-else

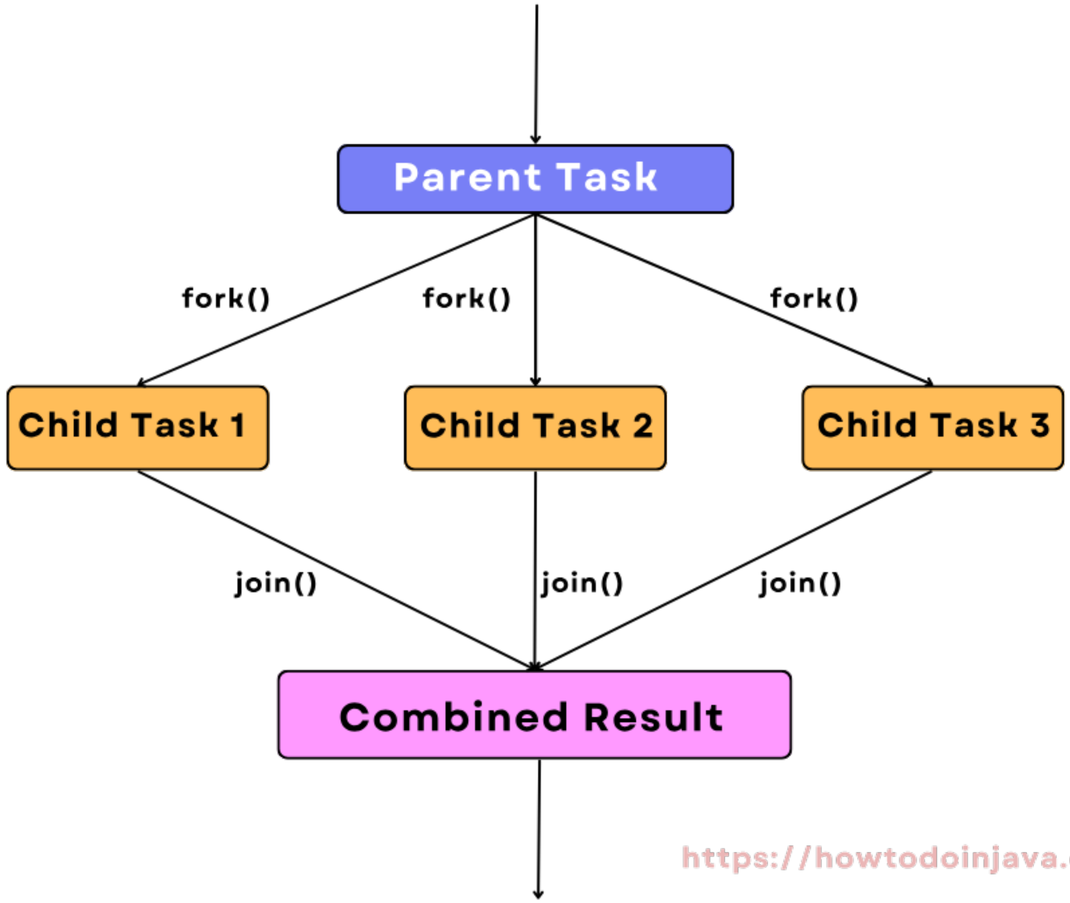


Selection



Repetition

Structured Concurrency



Invoke All pattern

```
8     RoutingInfo getData() throws ExecutionException, InterruptedException {
9
10        try(var scope = new StructuredTaskScope.ShutdownOnFailure) {
11
12            var route  = scope.fork(this::getRoute);
13            var traffic = scope.fork(this::loadTraffic);
14            var weather = scope.fork(this::loadWeather);
15
16            scope.join();
17            scope.throwIfFailed();
18
19            return new RoutingInfo(
20                    route.get(),
21                    traffic.get(),
22                    weather.get());
23
24        }
25    }
```

Invoke Any pattern

```
7     WeatherInfo getData() throws ExecutionException, InterruptedException {
8
9         try(var scope = new StructuredTaskScope.ShutdownOnSuccess<WeatherInfo>()) {
10
11            scope.fork(this::loadOpenWeather);
12            scope.fork(this::loadFrogCast);
13
14            scope.join();
15            return scope.result();
16        }
17    }
```

Scoped Values

Thread Locals

1. Share information between different components of your application
2. Create a ThreadLocal instance that is reachable from anywhere
3. You can provide an initial value on creation
4. Or set a value using set (T value)
5. Retrieve the value anywhere via get( )

The problem with Thread Locals

1. Mutable
2. Resource intensive
3. Leaking

Enter Scoped Values

1. Exist for a limited time (lifetime of the Runnable)
2. Only the thread that wrote the value can read it
3. Immutable
4. Passed by reference

Scoped Values

```java
public class ScopedExample {
    public static final ScopedValue<String> SCOPED_SECRET = ScopedValue.newInstance();
    public static final ScopedValue<Principal> SV_PRINCIPAL = ScopedValue.newInstance();

    Runnable runnable = () -> {
        new ScopedOutput().showIt();
    };


    public static void main(String... args) {
        new ScopedExample().shareIt();
        System.out.println("Is bound outside of method: " + SCOPED_SECRET.isBound());
    }

    public void shareIt() {
        Principal principal = new Principal() {
            @Override
            public String getName() {
                return "Duke!";
            }
        };
        ScopedValue
                .where(SCOPED_SECRET, "This is a scoped value")
                .where(SV_PRINCIPAL, principal)
                .run(runnable);
    }


    public void makeItCrash() { System.out.println("This will crash: " + SCOPED_SECRET.get()); }
```

Record Pattern Matching

Switch Pattern Matching

```
String normalSwitch(int value) {

    var result = switch (value) {
        case 0 -> "Zero"
        case 10 -> "Ten"
        case 100 -> "Hundred"
       default -> "Much"
    }

    return result;
}
```

Switch Pattern Matching


```
static String formatObject(Object obj) {
    var result = switch(obj) {
        case null      -> "null";
        case Integer i -> String.format("%d", i);
        case Long l    -> String.format("%d", l);
        case String s  -> s;
        case Boolean b -> b.toString();
        case Object o  -> String.format("%o", o);
    };
    return result;
}
```

Switch Pattern Matching


```
String normalSwitch(int value) {

    var result = switch (value) {
        case 0 -> "Zero"
        case 10 -> "Ten"
        case 100 -> "Hundred"
       default -> "Much"
    }

    return result;
}
```

```
static String formatObject(Object obj) {
    var result = switch(obj) {
        case null      -> "null";
        case Integer i -> String.format("%d", i);
        case Long l    -> String.format("%d", l);
        case String s  -> s;
        case Boolean b -> b.toString();
        case Object o  -> String.format("%o", o);
    };
    return result;
}
```

Switch Pattern Matching


```
public void sendEmails(List<FrequentFlyer> members) {

    for (var m : members) {
        switch (m) {
            case BlueWing   b -> sendEncouragingEmail(b);
            case SilverWing s -> sendNearlyThereEmail(s);
            case RoyalWing  r -> sendMajesticEmail(r);
        }
    }
}
```

Record Pattern Matching


```
public void sendEmails(List<Member> members) {

    for (var m : members) {
        switch (m) {
            case BlueWingRec(var id, var name, var email, var active) -> sendEncouragingEmail(name, email);
            case SilverWingRec(var id, var n, var e, var a) -> sendNearlyThereEmail(n, e);
            case RoyalWingRec(var id, var n, var e, var a, var v) -> sendMajesticEmail(n, e, v);
        }
    }
}
```

Record Pattern Matching

```
case BlueWingRec(var id, var name, var email, var active) -> sendEncouragingEmail(name, email);
```

- Checks if the object is of type BlueWingRec
- If so:
    - Create 4 local variables
        - Long id
        - String name
        - String email
        - Boolean active
    - Assigns the value if BlueWingRec.id to id
    - Assigns the value of BlueWingRec.name to name
    - Assigns the value of BlueWingRec.email to email
    - Assigns the value of BlueWingRec.active to active

Unnamed Patterns And Variables

Unnamed Patterns And Variables


```
public void sendEmails(List<Member> members) {

    for (var m : members) {
        switch (m) {
            case BlueWingRec(var id, var name, var email, var active) -> sendEncouragingEmail(name, email);
            case SilverWingRec(var id, var n, var e, var a) -> sendNearlyThereEmail(n, e);
            case RoyalWingRec(var id, var n, var e, var a, var v) -> sendMajesticEmail(n, e, v);
        }
    }
}
```

Unnamed Patterns And Variables


```
public void sendEmails(List<Member> members) {

    for (var m : members) {
        switch (m) {
            case BlueWingRec(var _, var name, var email, var _)   b -> sendEncouragingEmail(name, email);
            case SilverWingRec(var _, var n, var e, var _) s -> sendNearlyThereEmail(n, e);
            case RoyalWingRec(var _, var name, var email, var _, var visits)  r -> sendMajesticEmail(name, email, visits);
        }
    }
}
```

Unnamed Patterns And Variables


```
int count = 0;
for (String _ : fruits) {
    System.out.println("This is iteration: " + ++count);
}

//  Ignoring the return value of a method
var _ = fruits.getFirst();


} catch (Exception _) {
    System.out.println("Division by zero");
}

void open(File file) throws IOException {
    try (var _ = new FileInputStream(file)) {
        // ....
    }
    finally {

    }

}
```

Primitive Patterns, InstanceOf, And Switch

Primitive In Patterns, InstanceOf, And Switch

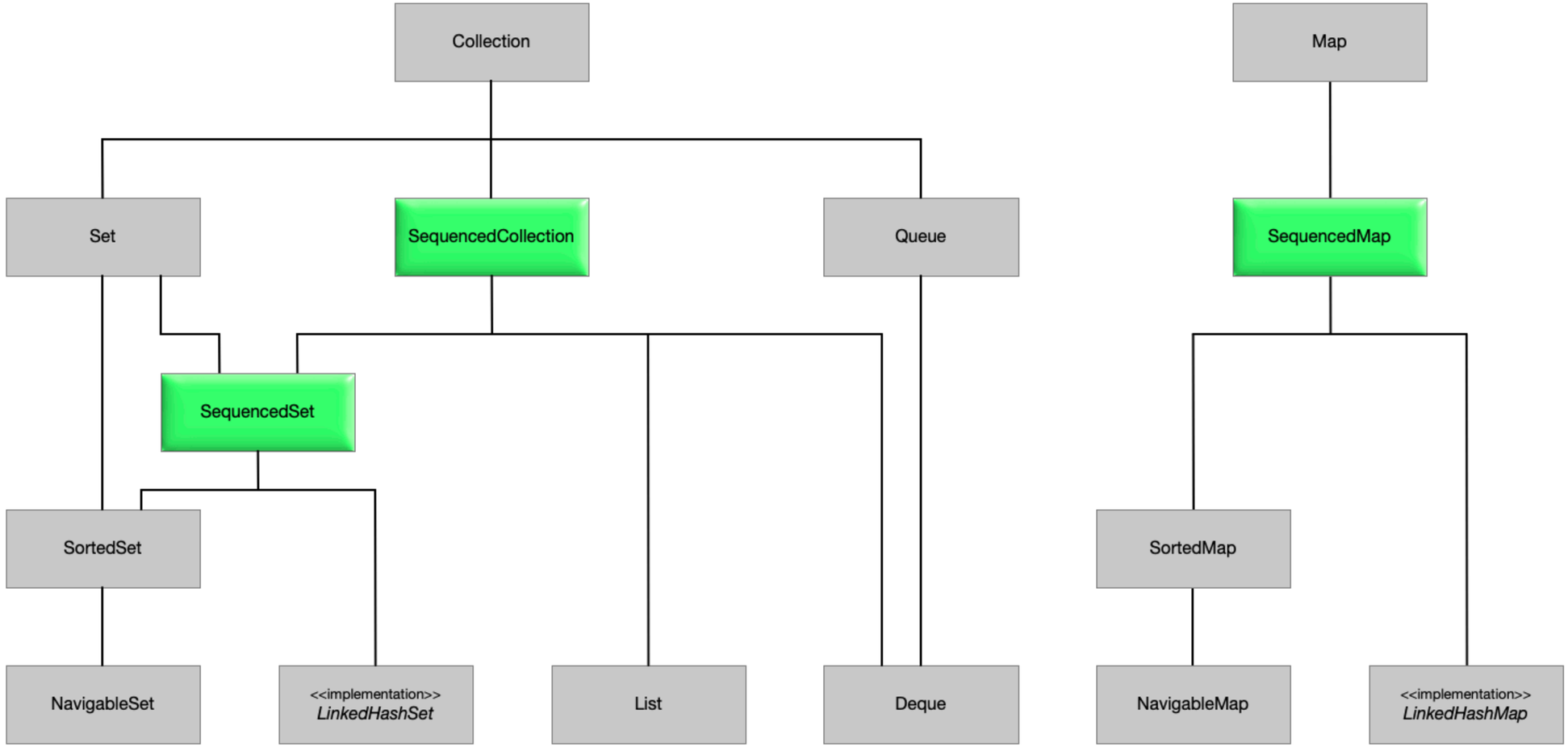

```
 switch (x.getStatus()) {
     case 0 -> "okay";
     case 1 -> "warning";
     case 2 -> "error";
     default -> "unknown status: " + x.getStatus();
 }
```

```
switch (x.getStatus()) {
    case 0 -> "okay";
    case 1 -> "warning";
    case 2 -> "error";
    case int i -> "unknown status: " + i;
}
```

```
 switch (x.getYearlyFlights()) {
     case 0 -> ...;
     case 1 -> ...;
     case 2 -> issueDiscount();
     case int i when i >= 100 -> issueGoldCard();
     case int i -> ... appropriate action when i > 2 && i < 100 ...
 }
```

Sequenced Collections

Sequenced Collections

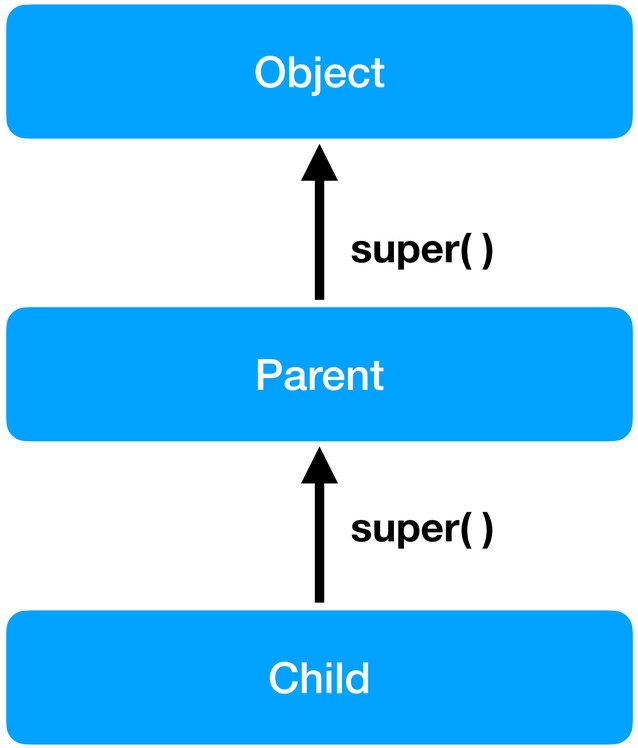


Sequenced Collections

```
public class SequencedCollections {
    ArrayList<String> list = new ArrayList<>(List.of("Rot", "Schwarz", "Gold"));

    public void someMethod() {
        list.getFirst();
        list.getLast();
        list.addFirst( element: "First now");
        list.addLast( element: "Last now");
        list.removeFirst();
        list.removeLast();
        list.reversed();
    }

    void main() {
        //  Prints Gold, Schwarz, Rot
        list.reversed().forEach(System.out::println);
    }
}
```

Flexible Constructor Bodies (Statements Before Super)

Statements Before Super



Statements Before Super

```java
public class StatementBeforeSuperParent {

    public StatementBeforeSuperParent(int size) {
        System.out.println("In StatementsBeforeSuperParent, size = " + size);
    }
}
```

```java
public class StatementsBeforeSuper extends StatementBeforeSuperParent {

    private List elements = null;

    public  StatementsBeforeSuper(int size) {
        System.out.println("size = " + size);
        super(size);
    }

}
```

Stream Gatherers

Stream setup

```
List.of("One", "Two", "Three", "Four", "Five")
        .stream()
        .filter(e -> !e.startsWith("T"))
        .map(e -> e.toUpperCase())
        .collect(Collectors.toList());
```




Stream sources

- Collections types
- Stream.of

Terminal Operations

- findFirst, findAny, etc.
- min, max
- Collectors.toCollection
- Collectors.toMap, Collectors.toList
- toList
- Implement your own via implementing the Collector interface

Intermediate Operations

- limit
- filter
- map
- flatMap
- takeWhile
- dropWhile
- skip
- sorted
- distinct

New Intermediate Operations

- mapConcurrent
- fold
- scan
- windowFixed
- windowSliding

Stream Gatherers


```java
public Map<Integer, List<String>> groupByLength(List<String> words) {
    return words.stream()                                          // ← Source
            .map(String::toUpperCase)                              // ← Intermediate operation
            .collect(Collectors.groupingBy(String::length));       // ← Terminal operation
}
```

```java
@SuppressWarnings("preview")
public List<List<String>> groupsOfThree(List<String> words) {
    return words.stream()                                          // ← Source
            .gather(Gatherers.windowFixed( windowSize: 3))         // ← Intermediate operation
            .toList();                                             // ← Terminal operation
}
```

Stream Gatherers

- Initializer
- Integrator
- Finisher
- Combiner

Gatherer API

initializer

```
default Supplier<A> initializer()
```

A function that produces an instance of the intermediate state used for this gathering operation.

Implementation Requirements:

integrator

```
Gatherer.Integrator[PREVIEW]<A,T,R> integrator()
```

A function which integrates provided elements, potentially using the provided intermediate state, optionally producing output to the provided `Gatherer.Downstream`[PREVIEW].

combiner

```
default BinaryOperator<A> combiner()
```

A function which accepts two intermediate states and combines them into one.

finisher

```
default BiConsumer<A,Gatherer.Downstream[PREVIEW]<? super R>> finisher()
```

A function which accepts the final intermediate state and a `Gatherer.Downstream`[PREVIEW] object, allowing to perform a final action at the end of input elements.

Create our own Map operation (stateless)

```java
public static <T, R> Gatherer<T, ?, R> altMap(Function<? super T, ? extends R> mapper) {
    Gatherer.Integrator<Void, T, R> integrator = (_, current, downstream) -> {
        var result = mapper.apply(current);
        downstream.push(result);
        return true;
    };
    return Gatherer.of(integrator);

}
```

```java
void main() {
    var list = List.of("Duke", "Duchess");
    var upper = list.stream()
            .gather(altMap(String::toUpperCase))
            .toList();
    System.out.println(upper);
}
```

```
[DUKE, DUCHESS]
```

Unnamed Classes and Instance Main Methods (Implicitly Declared Classes And Instance Main Methods)

Unnamed Classes And Instance Main Methods

```java
public class HelloLikeItUsedToBe {

    public static void main(String[] args) {
        System.out.println("Hello, familiar World!");
    }
}
```

```java
public class HelloLikeItUsedToBe {

    public void main(String[] args) {
        System.out.println("Hello, familiar World!");
    }
}
```

```java
public class HelloLikeItUsedToBe {

    void main(String[] args) {
        System.out.println("Hello, familiar World!");
    }
}
```

```java
public class HelloLikeItUsedToBe {

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

Unnamed Classes And Instance Main Methods

```java
void main() {
    println("Hi, Welcome to WeAreDevelopers");
}
```

```
veenr11@SJM-7WQ60R-r113CE5 src % java --enable-preview Simple.java
Hi, Welcome to WeAreDevelopers
```

Module Imports

Module Imports

- Explicitly import classes –> import java.Util.List
- Import with wildcards –> import java.util.*
- Import everything in a module
- Import module java.base would import 54 on–demand package imports

Module Imports

```java
import module java.base;

public class ModuleImport {

    void main() {
        List list = List.of("Java 23 is awesome");
        println(list);
    }
}
```

Module Imports

```
 */
module java.base {

    exports java.io;
    exports java.lang;
    exports java.lang.annotation;
    exports java.lang.classfile;
    exports java.lang.classfile.attribute;
    exports java.lang.classfile.components;
    exports java.lang.classfile.constantpool;
    exports java.lang.classfile.instruction;
    exports java.lang.constant;
    exports java.lang.foreign;
    exports java.lang.invoke;
    exports java.lang.module;
    exports java.lang.ref;
    exports java.lang.reflect;
    exports java.lang.runtime;
    exports java.math;
    exports java.net;
    exports java.net.spi;
    exports java.nio;
    exports java.nio.channels;
    exports java.nio.channels.spi;
    exports java.nio.charset;
    exports java.nio.charset.spi;
    exports java.nio.file;
    exports java.nio.file.attribute;
    exports java.nio.file.spi;
    exports java.security;
    exports java.security.cert;
    exports java.security.interfaces;
    exports java.security.spec;
    exports java.text;
    exports java.text.spi;
    exports java.time;
    exports java.time.chrono;
    exports java.time.format;
    exports java.time.temporal;
    exports java.time.zone;
    exports java.util;
    exports java.util.concurrent;
    exports java.util.concurrent.atomic;
    exports java.util.concurrent.locks;
    exports java.util.function;
    exports java.util.jar;
    exports java.util.random;
    exports java.util.regex;
    exports java.util.spi;
    exports java.util.stream;
    exports java.util.zip;
    exports javax.crypto;
    exports javax.crypto.interfaces;
    exports javax.crypto.spec;
    exports javax.net;
    exports javax.net.ssl;
    exports javax.security.auth;
    exports javax.security.auth.callback;
    exports javax.security.auth.login;
    exports javax.security.auth.spi;
    exports javax.security.auth.x500;
    exports javax.security.cert;
```

Launch Multi-File Source Code Program

Launch Multi-File Source Code Program

- Running source file from java was introduced in Java 11
- Does not require the source code to be compiled into byte code upfront
- Compiles the source code to memory and executes the main method
- Only direct java directly referenced java files are included

Markdown Documentation Comments

Markdown Documentation Comments

```
/**
 * Returns a hash code value for the object. This method is
 * supported for the benefit of hash tables such as those provided by
 * {@link java.util.HashMap}.
 * <p>
 * The general contract of {@code hashCode} is:
 * <ul>
 * <li>Whenever it is invoked on the same object more than once during
 *     an execution of a Java application, the {@code hashCode} method
 *     must consistently return the same integer, provided no information
 *     used in {@code equals} comparisons on the object is modified.
 *     This integer need not remain consistent from one execution of an
 *     application to another execution of the same application.
 * <li>If two objects are equal according to the {@link
 *     #equals(Object) equals} method, then calling the {@code
 *     hashCode} method on each of the two objects must produce the
 *     same integer result.
 * <li>It is <em>not</em> required that if two objects are unequal
 *     according to the {@link #equals(Object) equals} method, then
 *     calling the {@code hashCode} method on each of the two objects
 *     must produce distinct integer results.  However, the programmer
 *     should be aware that producing distinct integer results for
 *     unequal objects may improve the performance of hash tables.
 * </ul>
 *
 * @implSpec
 * As far as is reasonably practical, the {@code hashCode} method defined
 * by class {@code Object} returns distinct integers for distinct objects.
 *
 * @return  a hash code value for this object.
 * @see     java.lang.Object#equals(java.lang.Object)
 * @see     java.lang.System#identityHashCode
 */
```

```
/// Returns a hash code value for the object. This method is
/// supported for the benefit of hash tables such as those provided by
/// [java.util.HashMap].
///
/// The general contract of `hashCode` is:
///
///   - Whenever it is invoked on the same object more than once during
///     an execution of a Java application, the `hashCode` method
///     must consistently return the same integer, provided no information
///     used in `equals` comparisons on the object is modified.
///     This integer need not remain consistent from one execution of an
///     application to another execution of the same application.
///   - If two objects are equal according to the
///     [equals][#equals(Object)] method, then calling the
///     `hashCode` method on each of the two objects must produce the
///     same integer result.
///   - It is _not_ required that if two objects are unequal
///     according to the [equals][#equals(Object)] method, then
///     calling the `hashCode` method on each of the two objects
///     must produce distinct integer results.  However, the programmer
///     should be aware that producing distinct integer results for
///     unequal objects may improve the performance of hash tables.
///
/// @implSpec
/// As far as is reasonably practical, the `hashCode` method defined
/// by class `Object` returns distinct integers for distinct objects.
///
/// @return  a hash code value for this object.
/// @see     java.lang.Object#equals(java.lang.Object)
/// @see     java.lang.System#identityHashCode
```

Who is Ron Veen

- Java developer for 20+ years
- Special Agent @ Team Rockstars IT
- Author:
  - Migrating to cloud-native Jakarta EE
  - Project Loom

@ronveen

@ronveen
.bsky.social

ronveen

@ronveen@foojay.social

Feedback

Zeskanuj kod i zostaw swoją opinię

WDI WARSZAWSKIE DNI INFORMATYKI

Modern Java

Ron Veen

https://warszawskiedniinformatyki.pl/user.html#!/lecture/WDI25-2d73/rate