

Języki Programowania

dr inż. Tomasz Kubik

tomasz.kubik.staff.iiar.pwr.edu.pl

Klasy w Java

- Dokumentowanie
 - komentarz blokowy dla javadoc
 - z ogranicznikami `/**` oraz `*/`
 - wykorzystywany przez javadoc
 - może zawierać adnotacje służące **do dokumentowania** (nie mylić z adnotacjami **do opisywania kodu**) oraz znaczniki html
 - komentarz blokowy standardowy:
 - pomiędzy ogranicznikami `/*` oraz `*/`
 - niewykorzystywane przez javadoc
 - komentarz do końca linii,
 - rozpoczynający się od znaków `//`

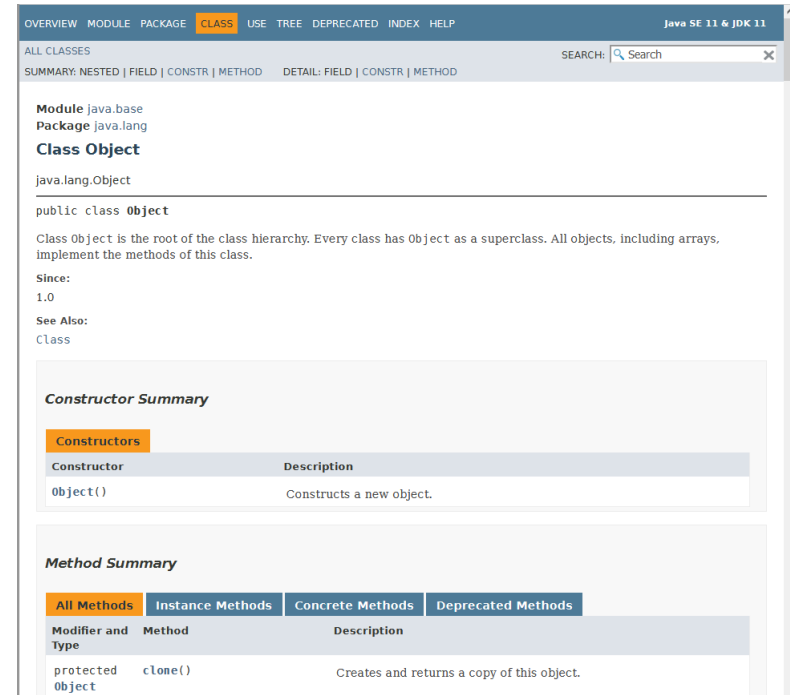
```
package java.lang;
```

```
import jdk.internal.HotSpotIntrinsicCandidate;
```

```
/**  
 * Class {@code Object} is the root of the class hierarchy.  
 * Every class has {@code Object} as a superclass. All objects,  
 * including arrays, implement the methods of this class.  
 *  
 * @author unascribed  
 * @see java.lang.Class  
 * @since 1.0  
 */
```

```
public class Object {  
  
    private static native void registerNatives();  
    static {  
        registerNatives();  
    }  
}
```

...



The screenshot shows the Java SE 11 API documentation for the `Object` class. The page is titled "Class Object" and is part of the `java.lang` package. It includes a "Constructor Summary" section with a table showing the `Object()` constructor, which constructs a new object. Below this is a "Method Summary" section with a table showing the `clone()` method, which creates and returns a copy of this object. The page also includes a search bar and navigation tabs for "OVERVIEW", "MODULE", "PACKAGE", "CLASS", "USE", "TREE", "DEPRECATED", "INDEX", and "HELP".

Constructor	Description
<code>Object()</code>	Constructs a new object.

Modifier and Type	Method	Description
protected Object	<code>clone()</code>	Creates and returns a copy of this object.

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html>

javadoc

Name

javadoc - generate HTML pages of API documentation from Java source files

Synopsis

javadoc [*options*] [*packagenames*] [*sourcefiles*] [*@files*]

options

Specifies command-line options, separated by spaces. See [Options for javadoc](#), [Extended Options](#), [Standard doclet Options](#), and [Additional Options Provided by the Standard doclet](#).

packagenames

Specifies names of packages that you want to document, separated by spaces, for example java.lang java.lang.reflect java.awt. If you want to also document the subpackages, then use the -subpackages option to specify the packages.

By default, javadoc looks for the specified packages in the current directory and subdirectories. Use the -sourcepath option to specify the list of directories where to look for packages.

sourcefiles

Specifies names of Java source files that you want to document, separated by spaces, for example Class.java Object.java Button.java. By default, javadoc looks for the specified classes in the current directory. However, you can specify the full path to the class file and use wildcard characters, for example /home/src/java/awt/Graphics*.java. You can also specify the path relative to the current directory.

@files

Specifies names of files that contain a list of javadoc tool options, package names, and source file names in any order.

Annotations	Since JDK/SDK
@author	1.0
{@code}	1.5
{@docRoot}	1.3
@deprecated	1.0
@exception	1.0
{@inheritDoc}	1.4
{@link}	1.2
{@linkplain}	1.4
{@literal}	1.5
@param	1.0
@return	1.0
@see	1.0
@serial	1.2
@serialData	1.2
@serialField	1.2
@since	1.1
@throws	1.2
{@value}	1.4
@version	1.0

Klasy w Java

- `Object`
 - klasa bazowa wszystkich klas
 - posiada metody:
 - `equals()` (razem z `hashCode()` jest wykorzystywane w kolekcjach, patrz: „*The general contract of hashCode...*” w opisie API)
 - `hashCode()` (natywna, dla obiektu o klasy potomnej można uzyskać wartość z tej metody wywołując `System.identityHashCode(o)`)
 - `finalize()`
 - `clone()` (natywna)
 - `toString()` (produkuje tekstową reprezentację obiektu)
 - `wait()`, `wait(long)`, `wait(long, int)`, `notify()`, `notifyAll()` (metody finalne, używane w programowaniu wielowątkowym)
 - `getClass()` (metoda finalna, używana do prześwietlania klasy, gdy wykorzystuje się mechanizmy refleksji)
- `String`
 - dziedziczy z `Object`
 - przesłania implementacje metod: `equals()` oraz `hashCode()`

```
// string hash code = s[0]*31^(n - 1) +  
s[1]*31^(n - 2) + ... + s[n - 1]  
System.out.println("FB".hashCode());  
System.out.println("Ea".hashCode());
```

Module java.base
Package java.lang
Class Object
java.lang.Object

public class Object

Constructors

Constructor	Description
<code>Object()</code>	Constructs a new object.

Method Summary

All Methods	Instance Methods	Concrete Methods	Deprecated Methods
Modifier and Type	Method		Description
protected Object	<code>clone()</code>		Creates and returns a copy of this object.
boolean	<code>equals(Object obj)</code>		Indicates whether some other object is "equal to" this one.
protected void	<code>finalize()</code>		Deprecated. The finalization mechanism is inherently problematic.
final Class<?>	<code>getClass()</code>		Returns the runtime class of this Object.
int	<code>hashCode()</code>		Returns a hash code value for the object.
final void	<code>notify()</code>		Wakes up a single thread that is waiting on this object's monitor.
final void	<code>notifyAll()</code>		Wakes up all threads that are waiting on this object's monitor.
String	<code>toString()</code>		Returns a string representation of the object.
final void	<code>wait()</code>		Causes the current thread to wait until it is awakened, typically by being notified or interrupted.
final void	<code>wait(long timeoutMillis)</code>		Causes the current thread to wait until it is awakened, typically by being notified or interrupted, or until a certain amount of real time has elapsed.
final void	<code>wait(long timeoutMillis, int nanos)</code>		Causes the current thread to wait until it is awakened, typically by being notified or interrupted, or until a certain amount of real time has elapsed.

Module java.base
Package java.lang
Class String
java.lang.Object
java.lang.String

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>, Constable, ConstantDesc

public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc

Cykl życia

- Przypomnienie (1 wykład)
 - obiekt „żyje” dopóki jego referencja przechowywana jest w jakiejś zmiennej,
 - gdy referencja ginie, obiekt staje się kandydatem do usunięcia,
 - przed usunięciem przez odśmiecacz uruchamiana jest finalizacja (ale nie zawsze nadąża),
 - podczas uruchomienia wirtualnej maszyny Java można zadeklarować, z jakim odśmiecaczem ma ona wystartować (po uruchomieniu aplikacji odśmiecacza nie da się już zmienić)
- Dostarczane GC
 - Serial Garbage Collector
 - `-XX:+UseSerialGC`
 - **Parallel Garbage Collector** (domyślny)
 - `-XX:+UseParallelGC`
 - `-XX:ParallelGCThreads=<N>`
 - `-XX:MaxGCPauseMillis=<N>`
 - `-XX:GCTimeRatio=<N>`
 - `-Xmx<N>`
 - CMS Garbage Collector
 - `-XX:+UseParNewGC`
 - `-XX:-UseGCOverheadLimit`
 - G1 Garbage Collector
 - `-XX:+UseG1GC`
 - `-XX:+UseStringDeduplication`
 - Z Garbage Collector
 - `-XX:+UseZGC`
 - `-XX:+UnlockExperimentalVMOptions`

```
// Wywoływanie: java -XX:+UseParallelGC -cp . ex01.A
package ex01;
public class A {

    public static A a = null;
    public void m() {
        System.out.println("I am 'a'");
    }

    @Override
    protected void finalize() {
        // Wywoływana nie więcej niż 1 raz dla danego obiektu
        // deprecated od jdk 9
        // Do poczytania: https://www.baeldung.com/java-finalize
        a = this;
        System.out.println("finalize()");
    }

    public static void main(String[] args)
        throws IOException {

        A a = new A();
        a.m();
        a = null;
        System.runFinalization();
        System.gc();
        System.in.read();
        a = A.a;
        a.m();
        a = null;
        System.gc();
        System.in.read();
    }
}
```

<https://www.baeldung.com/jvm-garbage-collectors>

<https://www.oracle.com/java/technologies/javase/gc-tuning-6.html>

Dziedziczenie klas

- Dziedziczenie jednokrotne, przy czym
 - **konstruktor**
 - bezargumentowy dostarczany jest przez kompilator (w przypadku braku jakichkolwiek konstruktorów),
 - domyślnie posiada w pierwszej linii wywołanie bezargumentowego konstruktora klasy nadrzędnej (co można zmienić, deklarując wywołanie wybranego konstruktora klasy nadrzędnej lub klasy bieżącej).
 - **metody instancyjne** klasy nadrzędnej
 - można przesłonić (panuje mechanizm „funkcji wirtualnych”, przy pisaniu kodu pomocna staje się adnotacja `@Override`),
 - **metody statyczne** klasy nadrzędnej
 - można nadpisać/ukryć (z ang. `overwrite/hide`),
 - **pola instancyjne** klasy nadrzędnej
 - można nadpisać/ukryć (z ang. `overwrite/hide`).

Uwaga:

części ukryte/nadpisane **nie giną** (dostęp jest uzależniony od zastosowanego typu),
zastosowanie słowa `final` ogranicza dziedziczenie (przypomnienie z wykładu 2)

```
public class A {
    public int i = 1;
    public static int j = 10;
    public static void m() {
        System.out.println("A.m() j=" + j);
    }
    public void n() {
        System.out.println("A.n() i="+i + " j=" + j);
    }
}

public class B extends A {
    public int i = 2;
    public static int j = 20;
    // ukrywanie metody, nie można zastosować @Override
    public static void m() {
        System.out.println("B.m() j=" + j);
        @Override // przesłanianie metody
    }
    public void n() {
        System.out.println("B.n() i="+ i + " j=" + j);
    }
}

public class C extends B{
    public static void main(String[] args) {
        A aa = new A(), ab = new B();
        aa.m(); // wypisze A.m() j=10, lepiej użyć A.m();
        aa.n(); // wypisze A.n() i=1 j=10
        ab.n(); // wypisze B.n() i=2 j=20
        ab.m(); // wypisze A.m() j=10
        ((B) ab).m(); // wypisze B.m() j=20
        ((B) ab).n(); // wypisze B.n() i=2 j=20
        System.out.println(ab.i); // wypisze 1
        System.out.println(((B)ab).i); // wypisze 2
        System.out.println(ab.j); // wypisze 10
        System.out.println(((B)ab).j); // wypisze 20
    }
}
```

Modyfikatory dostępu a dziedziczenie

- Modyfikatory dostępu
 - `public` dostęp możliwy z dowolnego miejsca,
 - `private` dostęp możliwy tylko w obrębie klasy,
 - `protected` dostęp możliwy w obrębie klasy oraz dla klas potomnych z tego samego pakietu,
 - `package` (*brak modyfikatora*) dostęp możliwy tylko dla klas z tego samego pakietu.
- Dostęp do instancji klasy nadrzędnej i bieżącej:
 - `super`, `this`
- Dostęp do konstruktora klasy nadrzędnej i bieżącej:
 - `super()`, `this()`

Specifier	class	subclass	package	world
private	X			
protected	X	X*	X	
public	X	X	X	X
package	X		X	

```
package ex03;
```

```
public class A {  
    protected void m() {}  
}
```

```
package ex04;  
import ex03.A;
```

```
public class B extends A {  
  
    public static void main(String[] args) {  
        A a = new A();  
        a.m();           // tu metoda niedostępna  
    }  
  
    @Override           // tu metoda dostępna:  
    protected void m() { // do przesłonięcia  
        super.m();      // do wywołania  
    }  
}
```

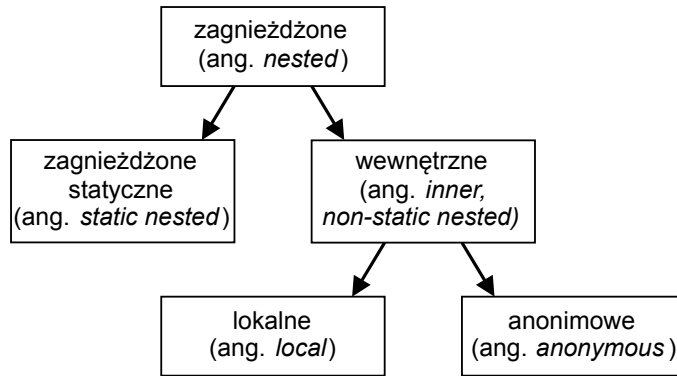
Zasady dziedziczenia

- Klasy abstrakcyjne
 - mogą pojawić się na dowolnym poziomie drzewa dziedziczenia, choć zwykle znaleźć je można przy korzeniu,
 - nie można tworzyć ich instancji, ale można wykorzystywać jako typy,
 - klasa może być abstrakcyjna nawet jeśli nie posiada żadnej metody abstrakcyjnej.
- Klasy potomne
 - klasa dziedzicząca po klasie abstrakcyjnej musi zaimplementować odziedziczone metody abstrakcyjne, aby nie być klasą abstrakcyjną,
 - implementacje odziedziczonych metod nie mogą zawężać dostępności,
 - implementacje odziedziczonych metod zgłaszających wyjątki mogą zgłaszać wyjątki tych samych typów bądź ich specjalizacji, mogą też nie zgłaszać tych wyjątków,
 - implementacje odziedziczonych metod mogą zwracać wartości tych samych typów bądź ich specjalizacji,
 - do metod statycznych lepiej odwoływać się poprzez klasę, a nie instancję.

```
abstract class A {  
    protected abstract Number m() throws Exception;  
}  
  
class B extends A {  
    @Override  
    public Integer m() throws IOException {  
        return null;  
    }  
}
```

```
class A {  
    public int i = 1;  
    public static int j = 10;  
    public static void m() {  
        System.out.println("A.m() j=" + j);  
    }  
    public void n() {  
        System.out.println("A.n() i="+i + " j=" + j);  
    }  
}  
  
class B extends A {  
    public int i = 2;  
    public static int j = 20;  
    // ukrywanie metody, nie można zastosować @Override  
    public static void m() {  
        System.out.println("B.m() j=" + j);  
    } @Override // przesłanianie metody  
    public void n() {  
        System.out.println("B.n() i="+ i + " j=" + j);  
    }  
}  
  
class C extends B {  
    public static void main(String[] args) {  
        A aa = new A(), ab = new B();  
        aa.m(); // wypisze A.m() j=10 (lepiej użyć A.m());  
        aa.n(); // wypisze A.n() i=1 j=10  
        ab.n(); // wypisze B.n() i=2 j=20  
        ab.m(); // wypisze A.m() j=10  
        ((B) ab).m(); // wypisze B.m() j=20  
        ((B) ab).n(); // wypisze B.n() i=2 j=20  
        System.out.println(ab.i); // wypisze 1  
        System.out.println(((B) ab).i); // wypisze 2  
        System.out.println(ab.j); // wypisze 10  
        System.out.println(((B) ab).j); // wypisze 20  
    }  
}
```


Klasy zagnieżdżone i wewnętrzne



- Klasy **zagnieżdżone** (ang. *nested classes*)
 - pojawiają się wewnątrz deklaracji klas, podobnie jak metody czy pola
 - mogą być statyczne (ang. *static*) lub instancyjne (ang. *non-static*)
 - mogą pojawić się wewnątrz deklaracji metod (wtedy mówi się o nich, że są klasami **lokalnymi**),
 - mogą pojawiać się jako klasy **anonimowe**/nienazwane (ang. *anonymous*)
- Klasy **wewnętrzne** (ang. *inner classes*)
 - są to klasy zagnieżdżone instancyjne (stanowią podzbiór klas zagnieżdżonych)
- Cechy klas zagnieżdżonych
 - można je deklarować jako `abstract` lub `final`,
 - mają one dostęp do metod i parametrów klasy w której zostały zadeklarowane, nawet do pól `private` (przy ograniczeniu wynikającym z użycia `static`),
 - poza klasami lokalnymi można je deklarować używając wszystkich modyfikatorów dostępu: `public`, `private`, `protected`, `package` (**bez modyfikatora**),
 - dostęp do instancji klasy zewnętrznej w klasie zagnieżdżonej wymaga specjalnej konstrukcji (`Outer.this`, patrz kod obok)
 - tworzenie instancji klas zagnieżdżonych na zewnątrz klasy wymaga specjalnej konstrukcji (`new Outer().new`, `new Outer.StaticNested()` patrz kod obok)
 - klasy **lokalne**
 - można je deklarować jedynie z dostępem pakietowym
 - nie mogą być `static`
 - mają dostęp do finalnych zmiennych metody, w której je zadeklarowano
 - nie można tworzyć ich instancji poza blokiem, w którym je zadeklarowano
 - mogą dziedziczyć po klasach abstrakcyjnych i implementować interfejsy
 - klasy **anonimowe**
 - zwykle deklarowane są w celu wstawienia in-line implementacji jakiegoś interfejsu

```
interface I {
    public abstract void n();
}

public class Outer {
    private int k = 0;
    private static int l = 0;

    public class Inner {
        public void accessTest() {
            System.out.println("k = " + k);
            System.out.println("l = " + Outer.this.k);
        }
    }

    public static class StaticNested {
        public void accessTest1() {
            System.out.println("l = " + l);
        }

        public static void accessTest2() {
            System.out.println("l = " + l);
        }
    }

    public void m(I i) {
        class LocalInner {
            private int j = 0;

            void n() { i.n(); }
        }
        LocalInner li = new LocalInner();
        li.j = 10;
        li.n();
    }

    public static void main(String args[]) {
        new Outer().new Inner().accessTest();
        new Outer.StaticNested().accessTest1();
        Outer.StaticNested.accessTest2();
        Outer o = new Outer();
        o.m(new I() { // nested anonymous
            @Override
            public void n() {
                System.out.println("p = " + o.k);
            }
        });
    }
}
```

Interfejsy

- Rozszerzają mechanizm dziedziczenia
 - można je wykorzystać jako typy (jak klasy abstrakcyjne).
- Deklarowane są podobnie do klas
 - zwykle z modyfikatorem `public` lub `package`, mogą być `strictfp`,
 - czasami wewnątrz klas, a wtedy mogą mieć modyfikator `private` lub `protected`,
 - nie mogą być deklarowane wewnątrz metod.
- Dziedziczenie interfejsów
 - wielokrotne
- Dostarczają:
 - stałe (zawsze `public static final`)
 - metody abstrakcyjne (zawsze `public abstract`),
 - metody z implementacją domyślną (zawsze `public`, mogą być `strictfp`),
 - metody statyczne (zwykle `public`, choć dla interfejsów wewnętrznych może być `private`)
- Konflikty
 - gdy w różnych interfejsach pojawiają się deklaracje metod abstrakcyjnych o tej samej nazwie, to klasa implementująca te interfejsy dostarcza jedną tylko metodę
 - implementowane metody podlegają regułom podobnym jak metody dziedziczone z klas (nie mogą mieć zawężonego dostępu, nie mogą wyrzucać bardziej ogólnych wyjątków itd.)

```
interface I {
    void f();
}
interface J {
    void g();
}
interface K extends I, J {
}
class A implements K {
    @Override
    public void f() {}

    @Override
    public void g() {}
}
```

```
interface I {
    void f();
    static void g() {}
}
interface J {
    void f();
    static void g() {}
}
class A implements I, J {
    public void f() {}
}
```

```
class D{
    protected interface I{ // deklaracja interfejsu
        // z prywatną metodą statyczną
        private static void g() {
        }
    }
    void m() {
        I.g(); // użycie prywatnej metody statycznej interfejsu
    }
}
@FunctionalInterface // Adnotacja interfejsu z 1 tylko metoda abstrakcyjna
interface I { // Deklaracja interfejsu, tu: package
    int i = 10; // Deklaracja stałej, tu niejawnie: public static final

    void f(); // Deklaracja metody bez implementacji,
              // tu niejawnie: public abstract

    default void g() { // Deklaracja metody z implementacją domyślną,
                      // tu niejawnie: public
        System.out.println("g");
    }

    static void h() { // Deklaracja metody statycznej z implementacją,
                    // tu niejawnie: public
        System.out.println("h");
    }
}
```

Interfejsy funkcyjne

- Posiadają tylko jedną metodę abstrakcyjną
- Dają podstawę do programowania funkcyjnego
 - patrz wyrażenia lambda (linki poniżej)
- Z ich pomocą przetwarza się strumienie (Stream API)

Module `java.base`

Package `java.util.function`

Functional interfaces provide target types for lambda expressions and method references. Each functional interface has a single abstract method, called the *functional method* for that functional interface, to which the lambda expression's parameter and return types are matched or adapted. Functional interfaces can provide a target type in multiple contexts, such as assignment context, method invocation, or cast context:

```
// Assignment context
Predicate<String> p = String::isEmpty;

// Method invocation context
stream.filter(e -> e.getSize() > 10)...

// Cast context
stream.map((ToIntFunction) e -> e.getSize())...
```

<code>BiConsumer<T,U></code>	<code>DoubleToIntFunction</code>	<code>IntUnaryOperator</code>	<code>ObjLongConsumer<T></code>
<code>BiFunction<T,U,R></code>	<code>DoubleToLongFunction</code>	<code>LongBinaryOperator</code>	<code>Predicate<T></code>
<code>BinaryOperator<T></code>	<code>DoubleUnaryOperator</code>	<code>LongConsumer</code>	<code>Supplier<T></code>
<code>BiPredicate<T,U></code>	<code>Function<T,R></code>	<code>LongFunction<R></code>	<code>ToDoubleBiFunction<T,U></code>
<code>BooleanSupplier</code>	<code>IntBinaryOperator</code>	<code>LongPredicate</code>	<code>ToDoubleFunction<T></code>
<code>Consumer<T></code>	<code>IntConsumer</code>	<code>LongSupplier</code>	<code>ToIntBiFunction<T,U></code>
<code>DoubleBinaryOperator</code>	<code>IntFunction<R></code>	<code>LongToDoubleFunction</code>	<code>ToIntFunction<T></code>
<code>DoubleConsumer</code>	<code>IntPredicate</code>	<code>LongToIntFunction</code>	<code>ToLongBiFunction<T,U></code>
<code>DoubleFunction<R></code>	<code>IntSupplier</code>	<code>LongUnaryOperator</code>	<code>ToLongFunction<T></code>
<code>DoublePredicate</code>	<code>IntToDoubleFunction</code>	<code>ObjDoubleConsumer<T></code>	<code>UnaryOperator<T></code>
<code>DoubleSupplier</code>	<code>IntToLongFunction</code>	<code>ObjIntConsumer<T></code>	

<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

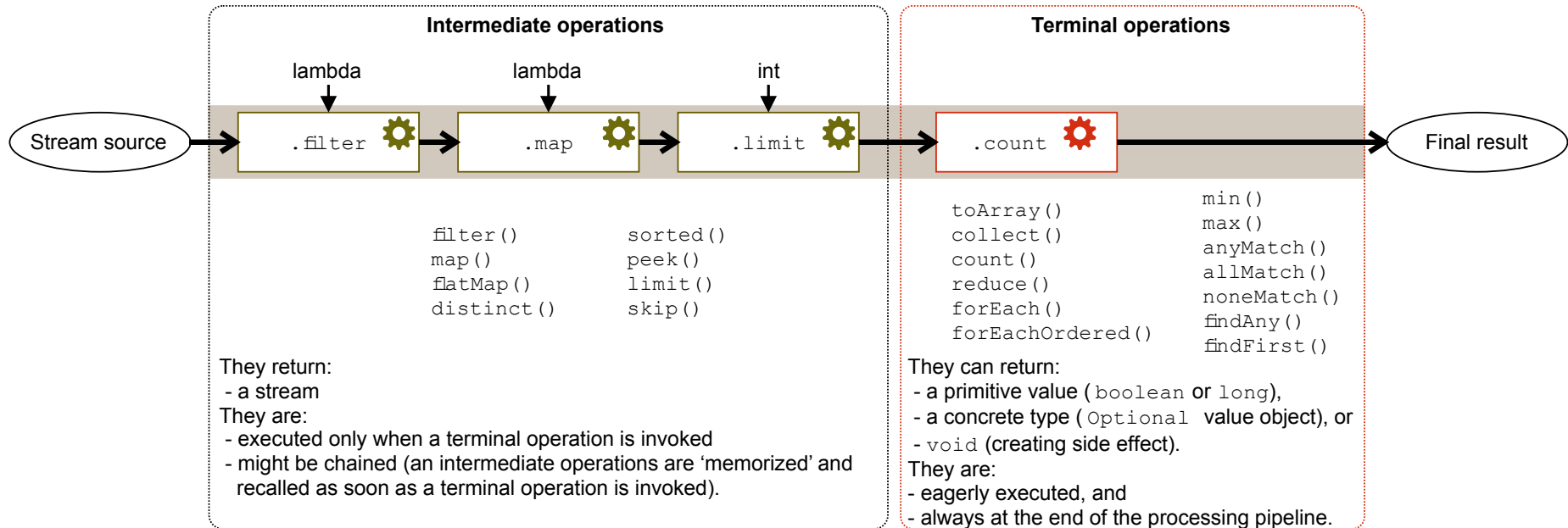
<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html>

https://www.studytrails.com/2016/09/10/java8_lambdas_functionalprogramming/

<https://openjdk.java.net/jeps/323>

<https://www.dariawan.com/tutorials/java/java-11-local-variable-syntax-lambda-parameters-jep-323/>

Java Streams



```
List<Integer> numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);  
numbers.stream().limit(4).forEach(System.out::println);  
//3,2,2,3
```