Java Tutorialsshort line

Amit Saha

Run Java

Suppose we have below folder structure for java files Amit.java and Shamit.java

src→saha →amit →Amit.java

→shamit →Shamit.java

**public class Amit {**

**public static void main(String[] args) {**

**System.out.println("Hello Amit");**

**}**

**}**

In java11 we can run directly without compiling like **java .\saha\amit\Amit.java**. This is called **single-file source-code.** But this will not work for Shamit.java which has reference to class Amit. So we have to compile first  **javac saha\amit\Amit.java saha\shamit\Shamit.java** then run **java saha.shamit.Shamit**  from src folder

**package saha.shamit;**

**import saha.amit.Amit;**

**public class Shamit {**

**Amit am = new Amit();**

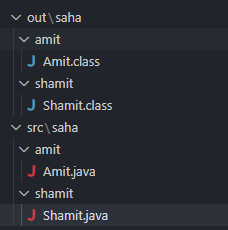
**public static void main(String[] args) {**

**System.out.println("Hello Shamit");**

**}**

**}**

By default the class files will be created next to .java files. If we want to keep our classes in a folder named “out” we have to use the -d option like ***javac -d out saha\amit\Amit.java saha\shamit\Shamit.java.*** After -d we mention the path where we want to create the class files. In our case it will give the below structure.



We may not always be in the folder from where we are running our command. This is not a problem for Amit.java which is single file source code and can be run from anywhere using java command**.**

***java JavaModule/src/saha/amit/Amit.java***

But classes like Shamit which have to run from class files need to be on classpath which is the root from where the package is starting. So for Shamit we need to be inside the “out” folder and run

***java saha.shamit.Shamit***

If we are ruining from location outside out we need to mention classpath or -cp option.

***java -cp JavaModule/out saha.shamit.Shamit***

Java Features

Java is a statically typed language which means you need to declare data types of variables before using. Type check is done at compile time. For dynamically typed languages like JS type check is done at run time. Each variable must have a type during declaration

Packages

How to access other classes

1. FQCN-Fully qualified class name e.g. ***java.util.ArrayList*** . If we are using two classes with the same name from two different packages we need to use FQCN.
2. Import **import java.util.ArrayList**
3. Import java.util\* discouraged but valid
4. **import static java.lang.Math.PI** import one static member of class for e.g. PI
5. **import static java.lang.Math\*** import all static members of a class for e.g. in this case all static members of class Math

No need to import java.lang package

Can’t import 2 classes with the same name from different packages, for e.g. **import java.util.Date; import java.sql.Date;** will not compile. It has to be like below. The date variable is referring to util and sql date using FQCN or vice versa.

**import java.util.Date;**

**public class Amit {**

**Date date;**

**java.sql.Date newDate;**

**}**

In the below case **java.sql** also has Date class but the date variable will refer to **java.sql.Date** and it takes presedence.

**import java.util.Date;**

**import java.sql.\*;**

**public class Amit {**

**Date date;**

**}**

Packages under java.lang are automatically imported and don't need to be imported separately. So we can directly use Math.PI in our code which is present inside the java.lang package. But if we want to use PI directly we either have to import in one of the following ways. The 2nd one includes all static variables and methods.

| **import static java.lang.Math.PI;**  **public class Test {**  **double d = PI;**  **}** | **import static java.lang.Math.\*;**  **public class Test {**  **static double d = PI;**  **public static void main(String[] args) {**  **sqrt(d);**  **}**  **}** |
| --- | --- |

We can’t **import java.lang.Math.PI** as PI is static so we need to keep a static keyword.

**import java.lang\*. Will import all classes under lang**

Primitives and String API

Primitives represent simple values not objects which can have complex Object oriented structure.Making the primitive types into objects would have degraded performance too much. Types of primitive.

byte : 1 byte 8 bits 2^8 or - 128 to 127

short : 16 bit 2^16 -32,768 to 32,767

int : 32 bit 2^32 -2,147,483,648 to 2,147,483,647

float : 123.4f

double : 123.4f

char : 16 bit similar to short 0-255

boolean : true or false

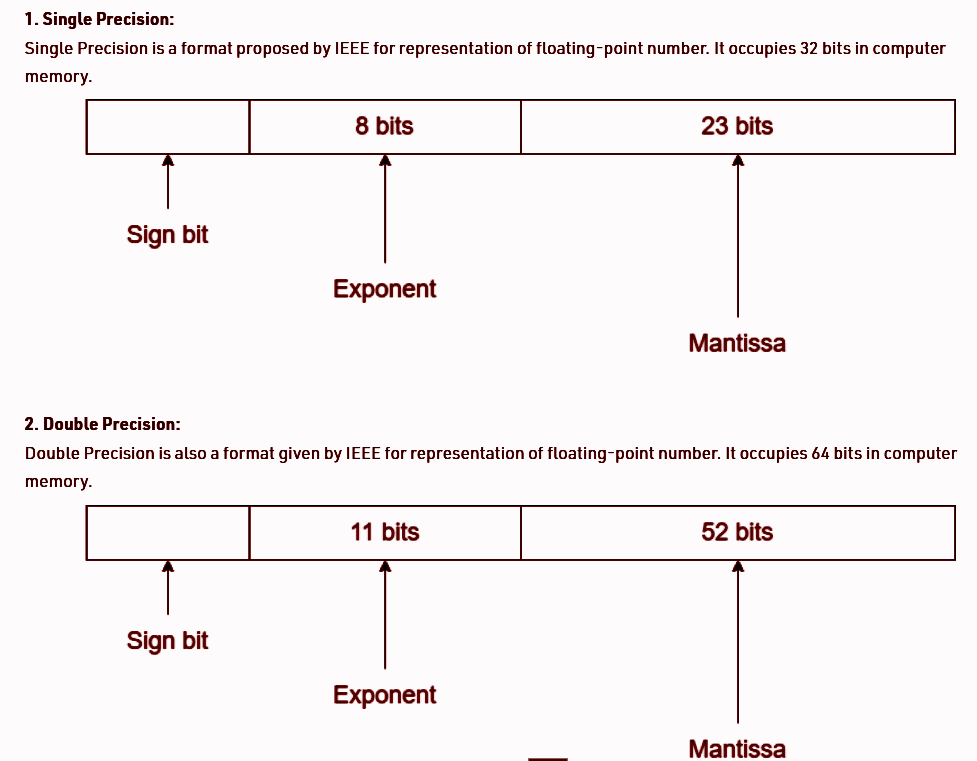
We can declare multiple variable in same line **int x, y;**

The smallest integer type is byte, byte is especially useful when you are working with a stream of data from a network or file. They are also useful when you are working with raw binary data that may not be directly compatible with Java's other built-in types.

Although you might think that using a byte or short would be more efficient than using an int in situations in which the larger range of an int is not needed, this may not be the case. The reason is that when byte and short values are used in an expression, they are promoted to int when the expression is evaluated

Char is denoted by ‘’ for eg **char c = 'A';** double quotes is for string.

The type float specifies a single-precision value that uses 32 bits of storage.Double precision, as denoted by the double keyword, uses 64 bits to store a value. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Floating-point literals in Java default to double precision. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.



For float it is mandatory to mention f or F. For double it's not mandatory but if not used results will be off.

| float f = 32.890880657f;  Prints 32.89088  double g = 32.3224234123;  Prints 32.3224234123 | float b = 10/3;  Prints 3.0  float c = 10/3F;  Prints 3.3333333 | double d = 10/3;  Prints 3.0  double e = 10/3d;  Prints 3.3333333333333335 | double z = 2.3;  Will compile  float x = 2.3;  Won't compile |
| --- | --- | --- | --- |

Can also use char literal for other types

**double d = 'a'; //97.0**

**float f = 'c'; //99.0**

**int i = 'Q'; //81**

STRING API

Strings are immutable, i.e. their values can’t be changed. When we assign new values to a string it just points to a new location in memory(thread pool), the original string stays in the pool.

**String s1 = "Hello";**

**s1.concat(" World");**

**System.out.println(s1); //Prints “Hello”**

When string is constructed without using a constructor then string is stored in a special area of heap called ***string pool*** whose purpose is to maintain a set of unique strings this is called ***interning***. When two strings have the same values then they point to the same location in the string pool hence the == method on two strings will return true. But this will not be the case when we use a specific constructor to create a string as shown below.

| **String s1 = "Hello";**  **String s2 = "Hello";**  **System.out.println(s1 == s2); //True** | **String s1 = new String("Hello");**  **String s2 = new String("Hello");**  **System.out.println(s1 == s2); //False** |
| --- | --- |
| **String s1 = new String("Hello").intern();**  **String s2 = new String("Hello").intern();**  **System.out.println(s1 == s2); //True** | **String s1 = new String();**  **String s2 = new String();**  **s1 = "hello"; s2 = "hello";**  **System.out.println(s1 == s2); //True** |

String can also be created from an array of bytes, char and integer. While passing array we can pass

**offset→** no of items that will be skipped from array

**count→** no elements taken from array

offset + count should be equal or less than the length of the array. Above params are optional for byte[] and char[] but mandatory for int[]. Let's say we create array of alphabets a -z and add them to string

***char[] ch = new char[26];***

***byte[] by = new byte[26];***

***int[] in = new int[26];***

***int i = 0;***

***for (char c = 'a'; c <= 'z'; c++, i++) {***

***ch[i] = c;***

***by[i] = (byte) c;***

***in[i] = c;***

***}***

***String s1 = new String(ch, 0, 26); //abcdefghijklmnopqrstuvwxyz***

***String s2 = new String(by, 2, 22);//cdefghijklmnopqrstuvwx ab from beginning and yz from end skipped***

***String s3 = new String(in, 1, 25); //count and offset is mandatory***

**String concat**  String uses concat() or + operator to join two strings

1. Although we can do string + 3 i.e. concat int or other types with string using “+” but **str.concat(3** will not compile.
2. Similarly **str + null** will compile and give something like “Hello null” but **str.concat(null)** will give a null pointer exception. However in either case we can’t use uninitialized variables like **String str2; str + str2 ;** This will not compile**.**
3. “+” operator converts int or any other literal to String using to string but we can override using ()

**int i = 20, j = 20;**

**System.out.println(i + j + " = " + i + j); // 40 = 2020**

**System.out.println(i + j + " = " + (i + j)); // 40 = 40**

String methods

| Methods | Purpose | Supported by StringBuilder |  |
| --- | --- | --- | --- |
| length() |  | y |  |
| charAt() |  | y |  |
| indexOf() | **sb= "Hello";**  **sb.indexOf("Hell")**  0 true -1 False | y |  |
| substring() | **sb.substring(0, 4)**  Hell | y |  |
| toLowerCase() |  | N |  |
| toUpperCase() |  | N |  |
| equals() |  | Y |  |
| equalsIgnoreCase() |  | N |  |
| startsWith() |  | N |  |
| endsWith() |  | N |  |
| replace() | **"abc".replace('a', 'A');**  **"abc".replace("a", "A");** | N | One format takes String other char sequence |
| contains() | **sb.contains("s");** | N |  |
| trim() |  |  |  |
| strip() |  |  |  |
| stripLeading() |  |  |  |
| stripTrailing() |  |  |  |
| intern() |  |  |  |

Functional Programming

# Lambda

A lambda expression is a short block of code which takes in parameters and can return a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method. For a Lambda we need a functional interface and we can override the functional interface in the Lambda code itself without the need of creating an implementation of Functional interface or writing an inner class implementation.

| **public class Test {**  **public static void main(String[] args) {**  **Shout sh = () -> System.out.println("Lambda");**  **sh.shout();**  **}**  **}**  **@FunctionalInterface**  **interface Shout {**  **void shout();**  **}** |  |
| --- | --- |

Without Lambda we have to Initialize the interface and implement the abstract method or do the same with Inner class implementation. But both have drawbacks as in both case lines of code increases and application becomes complicated and they create extra class files which can bloat the footprint of application

| **public class Test {**  **public static void main(String[] args) {**  **ShoutImpl sh = new ShoutImpl();**  **sh.shout();**  **}**  **}**  **@FunctionalInterface**  **interface Shout {**  **void shout();**  **}**  **class ShoutImpl implements Shout {**  **@Override**  **public void shout() {**  **System.out.println("Interface");**  **}**  **}** |  |
| --- | --- |

In Inner class we are not creating implementations but the inner class still creates Anonymous class.

| **public class Test {**  **public static void main(String[] args) {**  **Shout shoutInnerClass = new Shout() {**  **public void shout() {**  **System.out.println("Inner Class");**  **}**  **};**  **shoutInnerClass.shout();**  **}**  **}**  **@FunctionalInterface**  **interface Shout {**  **void shout();**  **}** |  |
| --- | --- |

A Lambda expression always needs a functional interface method to execute i.e. interface with one abstract method. It doesn't work with Abstract class. It is a good practice to keep one or two lines of code in Lambda expression.

# Method Reference

Method reference is used to refer to a method of functional interface. It is a compact and easy form of lambda expression. When we simply pass the parameters of Lambda expression to functional interface implementation we can instead use Method reference instead of Lambda which looks much cleaner. You can pretend the compiler turns your method references into lambdas for you.

| **TestClass obj = (text) -> System.out.println(text);**  **obj.testMethod("Dummy Text");** | **TestClass obj= System.out::println;**  **obj.testMethod("Dummy Text");** |
| --- | --- |

There are four formats for usage of method references, depending on where we are using it.

1. **CALLING STATIC METHODS :** Collection.sort(lst) becomes Collection::sort

**Consumer<List<Integer>> methodRef = Collections::sort;**

**methodRef.accept(lst);**

**Consumer<List<Integer>> lambda = s-> Collections.sort(s);**

**lambda.accept(lst);**

1. **CALLING INSTANCE METHODS ON OBJECT AND PASSING A PARAM**: str.startsWith("a") becomes str::startsWith("a")

**var str = "abc";**

**Predicate<String> methodRef = str::startsWith;**

**methodRef.test("a");**

**var str = "abc";**

**Predicate<String> lambda = s -> str.startsWith(s);**

**lambda.test("a");**

1. **CALLING INSTANCE METHODS ON OBJECT** : str.isEmpty()

**Predicate<String> methodRef = String::isEmpty;**

**methodRef.test("Yo");**

**Predicate<String> lambda = s -> s.isEmpty();**

**lambda.test("Yo");**

1. **CALLING CONSTRUCTORS**

**Supplier <List<String>> methodRef4 = ArrayList::new;**

**methodRef4.get();**

**Supplier<List<String>> lambda4= () -> new ArrayList();**

**lambda.get();**

# Functional Interface

Java provides some out of the box Functional Interface which can be reused. Mentioned below.

| **FI** | **method** | **info** | **Usage** |
| --- | --- | --- | --- |
| Supplier<T> | get() | generate or supply values without taking any input, like creating objects. | **Supplier<LocalDate> s1 = () -> LocalDate.now();**  **Supplier<LocalDate> s2 = LocalDate::now;**  **System.out.println(s1.get()+ s1.get());** |
| Consumer<T> | accept(T) | do something with a parameter but not return anything | **Consumer<String> c1 = System.out::println;**  **Consumer<String> c2 = x -> System.out.println(x);**  **c1.accept("Annie");**  **c2.accept("Annie");** |
| BiConsumer<T,U> | accept(T,U) | Same as consumer accepts 2 values | **var map = new HashMap<String, Integer>();**  **BiConsumer<String, Integer> b1 = map::put;**  **BiConsumer<String, Integer> b2 = (k, v) -> map.put(k, v);**  **b1.accept("chicken", 7);**  **b2.accept("chick", 1);** |
| Predicate<T> | test(T) | Used to test condition return boolean | **Predicate<String> p1 = String::isEmpty;**  **Predicate<String> p2 = x -> x.isEmpty();**  **System.out.println(p1.test(""));**  **System.out.println(p2.test(""));** |
| BiPredicate<T,U> | test(T,U) | Same as Predicate takes 2 input | **BiPredicate<String, String> b1 = String::startsWith;**  **BiPredicate<String, String> b2 = (s, p) -> s.startsWith(p);**  **System.out.println(b1.test("chicken", "chick"));**  **System.out.println(b2.test("chicken", "chick"));** |
| Function<T,R> | apply(T) |  | **Function<String, Integer> f1 = String::length;**  **Function<String, Integer> f2 = x -> x.length();**  **System.out.println(f1.apply("cluck"));**  **System.out.println(f2.apply("cluck"));** |
| BiFunction<U,T,R> | apply(U,T) |  | **BiFunction<String, String, String> b1 = String::concat;**  **BiFunction<String, String, String> b2 = (s, b) -> s.concat(b);**  **System.out.println(b1.apply("baby ", "chick"));**  **System.out.println(b2.apply("baby ", "chick"));** |
| UnaryOperator<T> | apply(T,U) |  | **UnaryOperator<String> u1 = String::toUpperCase;**  **UnaryOperator<String> u2 = x -> x.toUpperCase();**  **System.out.println(u1.apply("chirp"));**  **System.out.println(u2.apply("chirp"));** |
| BinaryOperator<T> | apply(T,U) |  | **BinaryOperator<String> b1 = String::concat;**  **BinaryOperator<String> b2 = (s, b) -> s.concat(b);**  **System.out.println(b1.apply("baby ", "chick"));**  **System.out.println(b2.apply("baby ", "chick"));** |

1 **Supplier<T>** : Returns an object and has no input. Has abstract method <T> get(). A Supplier is used when you want to generate or supply values without taking any input, like creating objects.

Supplier<LocalDate> s1 = LocalDate::now;

Supplier<LocalDate> s2 = () -> LocalDate.now();

LocalDate d1 = s1.get();

LocalDate d2 = s2.get();

System.out.println(d1 + "<--->" + d2);

Supplier<StringBuilder> sb1 = StringBuilder::new;

Supplier<StringBuilder> sb2 = () -> new StringBuilder("hello");

Supplier<String> su = () -> Integer.toString(new Random().nextInt());

String random = su.get();

System.out.println(su + " <----> " + random);

2 **Consumer<T> :** Returns nothing and has 1 input. Has abstract method accept(T),You use a Consumer when you want to do something with a parameter but not return anything

Consumer<String> c1 = System.out::println;

Consumer<String> c2 = x -> System.out.println(x);

c1.accept("Annie");

c2.accept("Annie");

3 **BiConsumer<T,U>** : Returns nothing and has 2 input. Has abstract method accept(T,U)

**var map = new HashMap<String, Integer>();**

**BiConsumer<String, Integer> b1 = map::put;**

**BiConsumer<String, Integer> b2 = (k, v) -> map.put(k, v);**

**b1.accept("chicken", 7);**

**b2.accept("chick", 1);**

**System.out.println(map);**

4 **Predicate<T> :** Takes an input return boolean. Has abstract method boolean test(T) .

**Predicate<String> p1 = String::isEmpty;**

**Predicate<String> p2 = x -> x.isEmpty();**

**System.out.println(p1.test(""));**

**System.out.println(p2.test(""));**

5 **BiPredicate<T,U> :** Takes 2 inputs and returns a boolean. Has an abstract method boolean test(T,U) .

**BiPredicate<String, String> b1 = String::startsWith;**

**BiPredicate<String, String> b2 = (string, prefix) -> string.startsWith(prefix);**

**System.out.println(b1.test("chicken", "chick"));**

**System.out.println(b2.test("chicken", "chick"));**

6 **Function<T,R> :** Represents a function that accepts one argument and produces a result. Has abstract method R apply(T) . It can take an input of one type and return something different.

**Function<String, Integer> f1 = String::length;**

**Function<String, Integer> f2 = x -> x.length();**

**System.out.println(f1.apply("cluck"));**

**System.out.println(f2.apply("cluck"));**

7 **BiFunction<U,T,R> :** Represents a function that accepts two arguments and produces a result. Has abstract method R apply(U,T) .

**BiFunction<String, String, String> b1 = String::concat;**

**BiFunction<String, String, String> b2 = (string, toAdd) -> string.concat(toAdd);**

**System.out.println(b1.apply("baby ", "chick"));**

**System.out.println(b2.apply("baby ", "chick"));**

8 **UnaryOperator<T> :** Represents a function that accepts an argument and produces a result of the same type. Has abstract method T apply(T) . This extends Function but the difference is that the return type and arguments are of the same type as compared to Function.

**UnaryOperator<String> u1 = String::toUpperCase;**

**UnaryOperator<String> u2 = x -> x.toUpperCase();**

**System.out.println(u1.apply("chirp"));**

**System.out.println(u2.apply("chirp"));**

9 **BinaryOperator<T> :** Represents a function that accepts an argument and produces a result of the same type. Has abstract method T apply(T) . This extends BiFunction but the difference is that the return type and arguments are of same type as compared to BiFunction

**BinaryOperator<String> b1 = String::concat;**

**BinaryOperator<String> b2 = (firstString, secondString) -> firstString.concat(secondString);**

**System.out.println(b1.apply("baby ", "chick"));**

**System.out.println(b2.apply("baby ", "chick"));**

# Optional

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java applications. You must **import java.util** package to use Optional. It provides methods which are used to check the presence of value for a particular variable.

Suppose we have an average method which takes an array of integers and calculates average for all the marks scored by a student. Now suppose a student was sick and did not take the exam, so input array length will be zero and so will be the average. This is not correct as average should not be applicable in this scenario. This we can handle using optional

**public static Optional<Double> average(int... scores) {**

**if (scores.length == 0)**

**return Optional.empty();**

**int sum = 0;**

**for (int score : scores)**

**sum += score;**

**return Optional.of((double) sum / scores.length);**

**}**

If no scores are passed then the method will return an empty optional, otherwise it will return the average wrapped in optional.

**System.out.println(average(90, 100)); *// Optional[95.0]***

**System.out.println(average()); *// Optional.empty***

As you can see above the Optional returns the result wrapped in Optional like Optional[95.0] instead of 95.0.So if we want to get the Double value then the Optional method provides a get() method to retrieve value from optional. But we must check if the value is present or not with ifPresent() method otherwise it will throw NoSuchElementException

**Optional<Double> opt = average(90, 100);**

**if (opt.isPresent())**

**System.out.println(opt.get()); *// 95.0***

**opt = average();**

**System.out.println(opt.get()); *// java.util.NoSuchElementException***

ifPresent() method does the same except that it Calls Consumer with value

**opt.ifPresent(System.out::println); *//95.0***

Other useful methods

**Optional<Double> opt2 = average();**

**System.out.println(opt2.orElse(Double.NaN));**

**System.out.println(opt2.orElseGet(() -> Math.random()));**

**System.out.println(opt2.orElseThrow()); *// NoSuchElementException***

**System.out.println(opt2.orElseThrow(() -> new IllegalStateException())); *// Throws exception created by calling Supplier***

# Stream

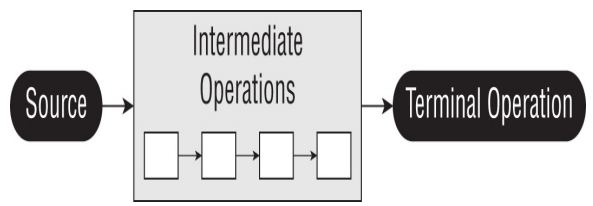
A stream in Java is a sequence of data. A stream pipeline consists of the operations that run on a stream to produce a result. There are three parts to a stream pipeline

Source:Where the stream comes from

Intermediate operations: Transforms the stream into another one. There can be as few or as many intermediate operations as you'd like. Since streams use lazy evaluation, the intermediate operations do not run until the terminal operation runs.

Terminal operation: Actually produces a result. Since streams can be used only once, the stream is no longer valid after a terminal operation completes.

Without Lambda we have to go for either Initializing the interface or Inner class implementation. But both have drawbacks that they create extra class files which can bloat the footprint of application



Creating Stream

**Stream<String> empty = Stream.empty();**

Empty Stream

**Stream<Integer> singleElement = Stream.of(1);**

**Stream<Integer> fromArray = Stream.of(1, 2, 3);**

Stream from elements

**var list = List.of("a", "b", "c");**

**Stream<String> fromList = list.stream();**

Stream from collections

Stream class provides generate method which takes a Supplier as input to create infinite stream

**Random rd = new Random();**

**Stream<Integer> intStream = Stream.generate(rd::nextInt);**

**intStream.limit(5).forEach(System.out::println);**

Infinite streams which generates Random Integer with Supplier

**Stream<Double> doubleStream = Stream.generate(Math::random);**

**doubleStream.limit(5).forEach(System.out::println);**

Infinite streams which generates Random Double with Supplier

**Stream<String> stringStream = Stream.generate(() -> "YO");**

**stringStream.limit(5).forEach(System.out::println);**

Infinite Stream of String "YO"

Stream also provides an iterate method to create streams. It takes in a seed which is the initial value and an unary operator to generate next values. This generates an infinite stream.

**var intStream = Stream.iterate(1, s -> s + 1);**

**intStream.limit(5).forEach(System.out::println);**

There is another overloaded version of iterate which takes a seed, an unary operator and a predicate to determine condition, This was introduced in Java 9 and the predicate can be used to make the stream finite. When the predicate returns false the stream stops

**var intStream = Stream.iterate(0, s -> s < 20, s -> s + 1);**

**intStream.forEach(System.out::println);**

Finite stream which stops when values reaches 19

# Stream Terminal Operations

1 **count() :** The count() method determines the number of elements in a finite stream. For an infinite stream, it never terminates

**Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");**

**System.out.println(s.count());**

2 **min() and max()** : The min() and max() methods allow you to pass a custom comparator and find the smallest or largest value in a finite stream according to that sort order. Like the count() method, min() and max() hang on an infinite stream. It returns a false for empty stream

**Optional<T> min(Comparator<? super T> comparator)**

**Optional<T> max(Comparator<? super T> comparator)**

**Stream<String> s = Stream.of("monkey", "donkey", "bonobo");**

**Optional<String> min = s.min((s1, s2) -> s1.length()-s2.length());**

**min.ifPresent(System.out::println);**

3 **findAny() and findFirst()** : The findAny() and findFirst() methods return an element of the stream unless the stream is empty. If the stream is empty, they return an empty Optional. It can terminate an infinite stream

**Stream<Double> randoms = Stream.generate(Math::random);**

**randoms.findAny().ifPresent(System.out::println);**

**Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);**

**oddNumbers.findFirst().ifPresent(System.out::println);**

4 **allMatch(), anyMatch(), and noneMatch()** : The findAny() and findFirst() methods return an element of the stream unless the stream is empty. If the stream is empty, they return an empty Optional. It can terminate an infinite stream

**anyMatch allMatch --> boolean = stream.anyMatch(Predicate <? super T> predicate)**

**var list = List.of("monkey", "2", "chimp");**

**Predicate<String> pred = x -> Character.isLetter(x.charAt(0));**

**System.out.println(list.stream().anyMatch(pred)); *// true***

**System.out.println(list.stream().allMatch(pred)); *// false***

**System.out.println(list.stream().noneMatch(pred)); *// false***

5 forEach() : Iterates over each element in a Stream, Won't work for infinite stream

**Stream<String> s1 = Stream.of("Monkey", "Gorilla", "Bonobo");**

**s1.forEach(System.out::print);**

6 reduce() : The reduce() method combines a stream into a single object. It is a reduction, which means it processes all elements. Reduction stream operations allow us to produce one single result from a sequence of elements, by repeatedly applying a combining operation to the elements in the sequence. It has 3 signatures.

REDUCE example 1 ->> T reduce(T identity, BinaryOperator<T> accumulator)

**Stream<String> stream = Stream.of("w", "o", "l", "f");**

**Stream<String> newstream = Stream.of("w", "o", "l", "f");**

**String word = stream.reduce("", (x, c) -> x + c);**

**String word2 = newstream.reduce("", String::concat);**

**System.out.println(word);**

**System.out.println(word2);**

REDUCE example 2 -->> Optional<T> reduce(BinaryOperator<T> accumulator) When identity is not provided it returns an optional

**BinaryOperator<Integer> op = (a, b) -> a \* b;**

**Stream<Integer> empty = Stream.empty();**

**empty.reduce(op).ifPresent(System.out::println); *// no output***

**Stream<Integer> oneElement = Stream.of(3);**

**oneElement.reduce(op).ifPresent(System.out::println); *// 3***

**Stream<Integer> threeElements = Stream.of(3, 5, 6);**

**threeElements.reduce(op).ifPresent(System.out::println); *// 90***

REDUCE example 3 -->> <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) The third method signature is used when we are dealing with different types. It allows Java to create intermediate reductions and then combine them at the end

**Stream<String> stream1 = Stream.of("w", "o", "l", "f!");**

**int length = stream1.reduce(0,**

**(i, w) -> i + w.length(),**

**(a, b) -> a + b);**

**System.out.println(length);**

7 collect() : The collect() method is a special type of reduction called a mutable reduction. It is more efficient than a regular reduction because we use the same mutable object while accumulating. It lets us get data out of streams and into another form

# Stream Intermediate Operations

Unlike a terminal operation, an intermediate operation produces a stream as its result. An intermediate operation can also deal with an infinite stream simply by returning another infinite stream

1 **filter() :** The filter() method returns a Stream with elements that match a given expression. Here is the method

signature: Stream<T> filter(Predicate<? super T> predicate)

**Stream<String> stream = Stream.of("monkey", "gorilla", "bonobo");**

**stream.filter(x -> x.startsWith("m")).forEach(System.out::print);*// monkey***

2 **distinct() :** The distinct() method returns a stream with duplicate values removed. Java calls equals() to determine whether the objects are the same

**Stream<String> s = Stream.of("duck-", "duck-", "duck-", "goose");**

**s.distinct().forEach(System.out::print);*//duck-goose***

3 **limit() and skip() :** The limit() and skip() methods can make a Stream smaller, or they could make a finite stream out of an infinite stream

**Stream<Integer> s2 = Stream.iterate(1, n -> n + 1);**

**s2.skip(5).limit(2).forEach(System.out::print);**

4 **map() :** The map() method creates a one‐to‐one mapping from the elements in the stream to the elements of the next step in the stream. <R> Stream<R> map(Function<? super T, ? extends R> mapper)

**Stream<String> s3 = Stream.of("monkey", "gorilla", "bonobo");**

**s3.map(String::length).forEach(System.out::print);**

5 **flatMap() :** The flatMap() method takes each element in the stream and makes any elements it contains top‐level elements in a single stream. This is helpful when you want to remove empty elements from a stream or you want to combine a stream of lists.

**List<String> zero = List.of();**

**var one = List.of("Bonobo");**

**var two = List.of("Mama Gorilla", "Baby Gorilla");**

**Stream<List<String>> animals = Stream.of(zero, one, two);**

**animals.flatMap(m -> m.stream()).forEach(System.out::println);**

6 **sorted() :** The sorted() method returns a stream with the elements sorted. Java uses natural ordering unless we specify a comparator.

**Stream<String> stream2 = Stream.of("brown-", "bear-");**

**stream2.sorted().forEach(System.out::print); *// bear-brown***

**Stream<String> stream3 = Stream.of("brown bear-", "grizzly-");**

**stream3.sorted(Comparator.reverseOrder()).forEach(System.out::print); *// grizzly-brown bear***

Take a look at the method signatures again. Comparator is a functional interface. This means that we can use method references or lambdas to implement it. The Comparator interface implements one method that takes two String parameters and returns an int. However, Comparator::reverseOrder doesn't do that. It is a reference to a function that takes zero parameters and returns a Comparator. This is not compatible with the interface. This means that we have to use a method and not a method reference

stream3.sorted(Comparator::reverseOrder);

7 **peek() :** It is useful for debugging because it allows us to perform a stream operation without actually changing the stream. Signature

Stream<T> peek(Consumer<? super T> action)

. peek() operation takes the same argument as the terminal forEach() operation. Think of peek() as an intermediate version of forEach() that returns the original stream back to you.

**Stream<String> stream2 = Stream.of("brown-", "bear-");**

**stream2.sorted().forEach(System.out::print); *// bear-brown***

**Stream<String> stream3 = Stream.of("brown bear-", "grizzly-");**

**stream3.sorted(Comparator.reverseOrder()).forEach(System.out::print); *// grizzly-brown bear***

# Primitive Stream

There are 3 type of Primitive streams

1 **IntStream**: : Used for the primitive types int, short, byte, and char.

2 **LongStream** : : Used for the primitive type long

3 **DoubleStream** : Used for the primitive types double and float

Common primitive stream methods

1. **OptionalDouble average()** : The arithmetic mean of the elements. Supported by all primitive Stream type
2. **Stream boxed()** : A Stream where T is the wrapper class associated with the primitive value. Supported by all primitive Stream types.
3. **OptionalInt max()** : The maximum element of the stream. Supported by integer primitive Stream type
4. **OptionalLong max()** : The maximum element of the stream. Supported by long primitive Stream type
5. **OptionalDouble max()** : The maximum element of the stream. Supported by double primitive Stream type
6. **OptionalInt min()** : The minimum element of the stream. Supported by integer primitive Stream type
7. **OptionalLong min()** : The minimum element of the stream. Supported by long primitive Stream type
8. **OptionalDouble min()** : The minimum element of the stream. Supported by double primitive Stream type
9. **IntStream range(int a, int b)** : Returns a primitive stream from a(inclusive) to b (exclusive)
10. **LongStream range(long a, long b)** : Returns a primitive stream from a(inclusive) to b (exclusive)
11. **IntStream rangeClosed(int a, int b)** : Returns a primitive stream from a(inclusive) to b (inclusive)
12. **LongStream rangeClosed(long a, long b)**: Returns a primitive stream from a(inclusive) to b (inclusive)
13. **int sum()** : Returns the sum of the elements in the stream
14. **long sum()** : Returns the sum of the elements in the stream
15. **double sum()** : Returns the sum of the elements in the stream
16. **IntSummaryStatistics summaryStatistics()** : Returns an object containing numerous stream statistics such as the average, min, max, etc.
17. **LongSummaryStatistics summaryStatistics()** : Returns an object containing numerous stream statistics such as the average, min, max, etc.
18. **DoubleSummaryStatistics summaryStatistics()** : Returns an object containing numerous stream statistics such as the average, min, max, etc.

Create Primitive stream

**var random = DoubleStream.generate(Math::random);**

**var fractions = DoubleStream.iterate(.5, d -> d / 2);**

**random.limit(3).forEach(System.out::println);**

**fractions.limit(3).forEach(System.out::println);**

FYI Random class provides methods which generate Primitive streams.

**Random random = new Random();**

**random.ints().limit(6).forEach(System.out::println);**

Primitive Stream can help in reducing the line of code and complexity drastically.

**IntStream count = IntStream.iterate(1, n -> n + 1).limit(5);**

**count.forEach(System.out::println); *// 12345***

We have to pass 6 as 2nd param is exclusive of last item

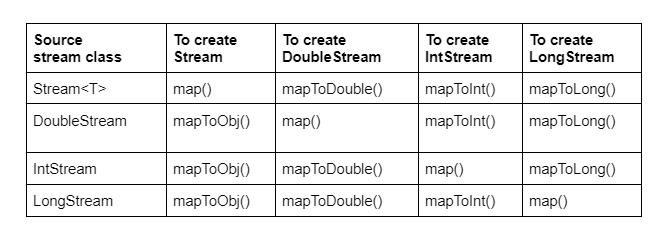
**IntStream rangeClosed = IntStream.rangeClosed(1, 5);**

**rangeClosed.forEach(System.out::println); *// 12345***

Since we are using rangeClosed() we can pass 5

# Mapping Stream

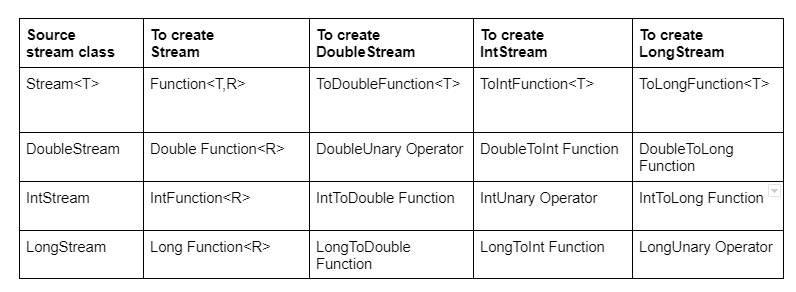
We can map streams from one type to another. They have to be compatible types for this to work. Java requires a mapping function to be provided as a parameter



This function takes an Object, which is a String in this case. The function returns an int. The function mappings are intuitive here. They take the source type and return the target type. In this example, the actual function type is ToIntFunction

**Stream<String> objStream = Stream.of("penguin", "fish");**

**IntStream intStream = objStream.mapToInt(s -> s.length());**

******

***Primitive Stream can help in reducing the line of code and complexity drastically.***

**IntStream count = IntStream.iterate(1, n -> n+1).limit(5);**

**count.forEach(System.out::println); *//12345***

***We have to pass 6 as 2nd param is exclusive of last item***

**IntStream rangeClosed = IntStream.rangeClosed(1, 5);**

**rangeClosed.forEach(System.out::println); *//12345***

***Since we are using rangeClosed() we can pass 5***

# Optional Primitives

***We have seen Optional can be used for methods which may or may not return value.***

**var stream = IntStream.rangeClosed(1, 10);**

**OptionalDouble optional = stream.average();**

***The OptionalDouble is different from Optional.It is used for Primitive streams. For Wrapper calls we use Optional<Double> and for Primitive class we user OptionalDouble. Working with the primitive optional class looks similar to working with the Optional class itself.The only noticeable difference is that we called getAsDouble() rather than get()***

**var stream = IntStream.rangeClosed(1, 10);**

**OptionalDouble optional = stream.average();**

**optional.ifPresent(System.out::println); *// 5.5***

**System.out.println(optional.getAsDouble()); *// 5.5***

**var stream2 = IntStream.of();**

**OptionalDouble optional1 = stream2.average();**

**System.out.println(optional1.orElseGet(() -> Double.NaN));**

***What will happen below?***

**var cats = new ArrayList<String>();**

**cats.add("Annie");**

**cats.add("Ripley");**

**var stream = cats.stream();**

**cats.add("KC");**

**System.out.println(stream.count());**

***We can add values to stream after creation unless its terminated***

I/O Stream

Different operating systems use different file systems to manage their data. But the JVM will automatically connect to the local file system, allowing you to perform the same operations across multiple platforms. Different operating systems vary in their format of pathnames. For example, Unix‐based systems use the forward slash, /, for paths, whereas Windows‐based systems use the backslash \ character. Java offers two options to retrieve the local separator character: a system property and a static variable defined in the File class

**System.out.println(System.getProperty("file.separator"));**

**System.out.println(java.io.File.separator);**

java.io.File class is used to read information about

* existing files and directories,
* list the contents of a directory,
* create/delete files and directories.

The File class cannot read or write data within a file, although it can be passed as a reference to many stream classes to read or write data

File operator usually takes a String which can be either an absolute path or relative path to the current directory. There are 3 ways to create files.

**File file1 = new File("text.txt"); Pass the relative or absolute path**

**File file2 = new File("test.txt", "new.txt"); Pass the path of a parent file to create child file**

**File file3 = new File(file1, "new.txt"); Pass the reference of parent file**

Note: Creating file object doesn't create a file in system

**System.out.println(file1.exists()); //Without file will be false**

# **FileMethods**

**file1.exists(); Returns true if file/directory exists**

**file1.delete(); Returns true if file/directory deleted**

**file1.getAbsolutePath(); Returns String path of file/directory**

**file1.getName(); Returns string Name of file/directory**

**file1.getParent(); Returns parent folder directory returns null if passed like this "test.txt"**

**file1.isDirectory(); Returns true if it is directory**

**file1.isFile(); Returns true if it is file**

**file1.lastModified(); Returns long timeStamp**

**file1.length(); Return long file size**

**file1.listFiles(); Returns list of files in directory**

**file1.mkdir();**

**file1.mkdirs();**

**File dest = new File("new.txt");**

**System.out.println(file1.renameTo(dest));**

It's important to notice from where we are running the java command in case of relative paths. For example if my class with below code is in folder src-->javatest-->Test.java. It also has a file test.txt. Now if I run file.exists() from src as java .\javaTest\Test.java it will return false as it will check for test.txt in src. But if we run it from javatest then it will find the file and return true

# I/O Stream

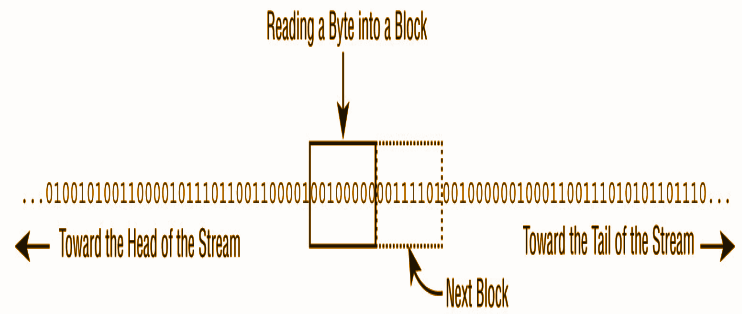
The contents of a file may be accessed or written via a stream, which is a list of data elements presented sequentially. Streams are like long, nearly never‐ending “streams of water” with data presented one “wave” at a time. The stream is so large that once we start reading it, we have no idea where the beginning or the end is. We just have a pointer to our current position in the stream and read data one block at a time.

Each type of stream segments data into a “wave” or “block” in a particular way. For example, some stream classes read or write data as

* Individual bytes.
* Individual characters or strings of characters
* Larger groups of bytes or characters at a time.

Irrespective of how the data is read or written nearly all are built on top of reading or writing an individual byte or an array of bytes at a time.

The reason higher-level streams exist is for convenience, as well as performance. For example, writing a file one byte at a time is time-consuming and slow in practice because the round-trip between the Java application and the file system is relatively expensive. By utilizing a BufferedOutputStream, the Java application can write a large chunk of bytes at a time, reducing the round-trips and drastically improving performance.



Files can be big like 1 TB. What stream does is helps reading chunks of data thus allowing systems with smaller specifications to be able to read data.

# Stream Types

## Byte vs. Character Streams

Stream can be classified as

| **Byte Stream** | **Character streams** |
| --- | --- |
| Which read/write binary data ( 0s and 1s) | Which read/write text data |
| have class names that end in InputStream or OutputStream | have class names that end in Reader or Writer |
| The byte streams are primarily used to work with binary data, such as an image or executable file | character streams are used to work with text files |
| FileInputStream | FileReader |

Even though character streams do not contain the word Stream in their class name, they are still I/O streams

Byte stream classes can write all types of binary data, including strings. So one might think that Character streams like reader are not necessary. However Character streams specifically focused on managing character and string data so that one doesn’t worry about the underlying character encoding of the file.

Java supports a wide variety of character encodings

**Charset usAsciiCharset = Charset.forName("US-ASCII");**

**Charset utf8Charset = Charset.forName("UTF-8");**

**Charset utf16Charset = Charset.forName("UTF-16");**

## Input VS Output Streams

Most InputStream stream classes have a corresponding OutputStream class, and vice versa. For example, the FileOutputStream class writes data that can be read by a FileInputStream. If you understand the features of a particular Input or Output stream class, you should naturally know what its complementary class does

It follows, then, that most Reader classes have a corresponding Writer class. For example, the FileWriter class writes data that can be read by a FileReader

**Exception**

**PrintWriter** no reader **PrintStream** has no Input stream

## Low vs High level Stream

A low‐level stream connects directly with the source of the data, such as a file, an array, or a String. Low‐level streams process the raw data or resource and are accessed in a direct and unfiltered manner

A high‐level stream is built on top of another stream using wrapping. Wrapping is the process by which an instance is passed to the constructor of another class

**try (var reader = new BufferedReader(new FileReader("text.txt"))) {**

**System.out.println(reader.readLine());**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

In this example, FileReader is the low‐level stream reader, whereas BufferedReader is the high‐level stream that takes a FileReader as input. Many operations on the high‐level stream pass through as operations to the underlying low‐level stream, such as read() or close(). Other operations override or add new functionality to the low‐level stream methods. The high‐level stream may add new methods, such as readLine(), as well as performance enhancements for reading and filtering the low‐level data.

High‐level streams can take other high‐level streams as input

**try (var ois = new ObjectInputStream(new BufferedInputStream(new FileInputStream("text.txt")))) {**

**System.out.print(ois.readObject());**

**} catch (ClassNotFoundException e) {**

**e.printStackTrace();**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

Buffered classes read or write data in groups, rather than a single byte or character at a time. The performance gain from using a Buffered class to access a low‐level file stream cannot be overstated. Unless you are doing something very specialized in your application, you should always wrap a file stream with a Buffered class in practice.

The java.io library defines four abstract classes that are the parents of all stream classes defined within the API: **InputStream, OutputStream, Reader, and Writer**. The constructors of high‐level streams often take a reference to the abstract class. For example, BufferedWriter takes a Writer object as input, which allows it to take any subclass of Writer.

**InputStream (Abstract) --> FileInputStream --> BufferedInputStream**

**OutputStream(Abstract) --> FileOutputStream--> BufferedOutputStream**

**Reader(Abstract)--> FileReader --> BufferedReader**

**Writer(Abstract)-->FileWriter -->BufferedWriter**

**new BufferedInputStream(new FileReader("z.txt")); *// DOES NOT COMPILE Reader wrapped in Input Stream***

**new BufferedWriter(new FileOutputStream("z.txt")); *// DOES NOT COMPILE Same***

**new ObjectInputStream( new FileOutputStream("z.txt")); *// DOES NOT COMPILE output wrapped in input***

**new BufferedInputStream(new InputStream()); *// DOES NOT COMPILE Input Stream is abstract***

# Summary

| **Class Name** | **Type** | **Description** |
| --- | --- | --- |
| InputStream | Abstract | Abstract class for all input byte streams |
| OutputStream | Abstract | Abstract class for all output byte streams |
| Reader | Abstract | Abstract class for all input character streams |
| Writer | Abstract | Abstract class for all output character streams |
| FileInputStream | Low | Reads file data as bytes |
| FileOutputStream | Low | Writes file data as bytes |
| FileReader | Low | Read file data as character |
| FileWriter | Low | Write file data as character |
| BufferedInputStream | High | Reads byte data from an existing InputStream in a buffered manner, which improves efficiency and performance |
| BufferedOutputStream | High | Writes byte data from an existing OutputStream in a buffered manner, which improves efficiency and performance |
| BufferedReader | High | Reads character data from an existing Reader in a buffered manner, which improves efficiency and performance |
| BufferedWriter | High | Writes character data to an existing Writer in a buffered manner, which  improves efficiency and performance |
| ObjectInputStream | High | Deserializes primitive Java data types and graphs of Java objects from an existing InputStream |
| ObjectOutputStream | HIgh | Serializes primitive Java data types and graphs of Java objects to an existing OutputStream |
| PrintStream | High | Writes formatted representations of Java objects to a binary stream |
| PrintWriter | High | Writes formatted representations of Java objects to a character stream |

# I/O Stream Methods

I/O Stream uses read and write method to read and write data to file

**public int read() throws IOException*// InputStream and Reader***

**public void write(int b) throws IOException *// OutputStream and Writer***

The int represents the bytecode value when it is -1 It represents the end of Stream.

# I/O Stream Operations

## Using FileInputStream and FileOutputStream

**File fileIn = new File("in.txt");**

**File fileOut = new File("out.txt");**

**try (var in = new FileInputStream(fileIn);**

**var out = new FileOutputStream(fileOut)) {**

**int data;**

**while ((data = in.read()) != -1) {**

**out.write(data);**

**}**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

The FileOutputStream class includes overloaded constructors that take a boolean append flag. When set to true, the output stream will append to the end of a file if it already exists

**var out = new FileOutputStream(fileOut, true)**

If the source file does not exist, a ***FileNotFoundException***, which inherits IOException, will be thrown. If the destination file is not existing then it will be created otherwise if it already exists.

Other implementation for read and write

**byte[] buffer = new byte[1024];**

**while ((data = in.read(buffer)) != -1) //data is read in buffer size provided for faster execution**

**while ((data = in.read(buffer, 2, 3)) != -1)// 2 bytes skipped and 3 bytes limit which is read**

Other implementation for write methods

In the below scenario during read() operation buffer should not be **~~in.read(buffer)~~**passed otherwise data will be jumbled.

**byte[] buffer = new byte[1024];**

**int data;**

**while ((data = in.read()) != -1) {**

**out.write(data);**

When buffer is passed during read it must be passed during write also otherwise again data will get jumbled

**while ((data = in.read(buffer)) != -1) {**

**out.write(buffer);**

**while ((data = in.read(buffer)) != -1) {**

**out.write(buffer, 0, data);**

## BufferedInput and Output Stream.

Previous implementations perform poorly on large files. As discussed earlier, that's because there is a cost associated with each round‐trip to the file system. We can easily enhance our implementation using BufferInputStream and BufferOutputStream

**try (var in = new BufferedInputStream(new FileInputStream(fileIn));**

**var out = new BufferedOutputStream(new FileOutputStream(fileOut))) {**

**byte[] buffer = new byte[1024];**

**int data;**

**while ((data = in.read(buffer)) != -1) {**

**out.write(buffer, 0, data);**

**out.flush();**

**}**

**} catch (Exception e) {**

**e.printStackTrace();**

**}**

Since the read/write methods that use byte[] exist in InputStream/OutputStream, why use the Buffered classes at all? Answer is because Buffered classes contain a number of performance improvements for managing data in memory

## FileReader Writer Stream

The FileReader and FileWriter classes, along with their associated buffer classes, are among the most convenient I/O classes because of their built-in support for text data.

**try (var fr = new FileReader(fileIn);**

**var fw = new FileWriter(fileOut)) {**

**int data;**

**while ((data = fr.read()) != -1) {**

**fw.write(data);**

**fw.flush();**

**}**

**} catch (Exception e) {**

**e.printStackTrace();**

**}**

The FileWriter inherits a method from the Writer class that allows it to write String values

**fw.write("Hello World");**

## BufferWriter and BufferReader

BufferWriter and BufferReader add two new methods, readLine() and newLine(), that are particularly useful when working with String values. These are not present for BufferInputStream and BufferOutputStream

**try (BufferedReader br = new BufferedReader(new FileReader(fileIn));**

**BufferedWriter bw = new BufferedWriter(new FileWriter(fileOut))) {**

**String line = "";**

**while ((line = br.readLine()) != null) {**

**System.out.println(line);**

**bw.write(line);**

**bw.newLine();**

**}**

**} catch (Exception e) {**

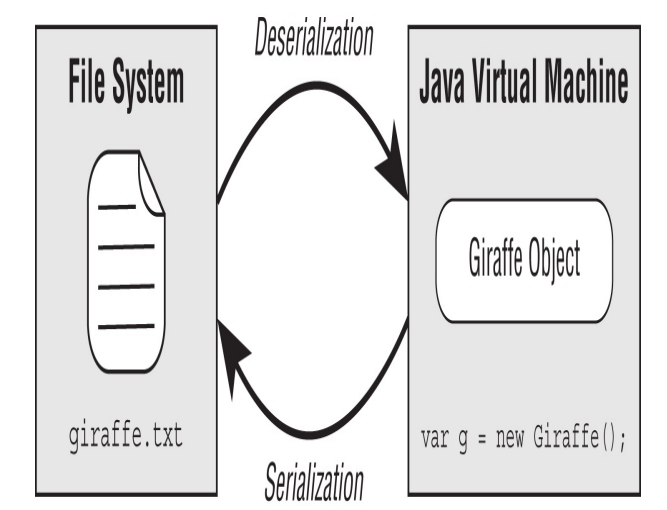
**e.printStackTrace();**

**}**

# Serialization & Deserialization

**Serialization** is the process of converting an in‐memory object to a byte stream

**Deserialization** is the process of converting a byte stream into an object



With this understanding we can say

Any Writer or Output stream is part of the Serialization process.

Any Reader or Input Stream is part of the Deserialization process.

To serialize an object using the I/O API, the object must implement the java.io.Serializable interface. The Serializable interface is a ***marker interface***. It means the interface does not have any methods. Any class can implement the Serializable interface since there are no required methods to implement.

But we should only mark data‐oriented classes serializable.

The purpose of using the Serializable interface is to inform any process attempting to serialize the object that you have taken the proper steps to make the object serializable. How to Make a Class Serializable

1. The class must be marked Serializable.
2. Every instance member of the class is ***serializable***, marked **transient**, or has a ***null value*** at the time of serialization.

All Java primitives and many of the built‐in Java classes that you have worked with throughout this book are Serializable. For e.g. String, Long, Double etc.

Class like String and Boolean directly implement serializable where as Long, Double, Integer extend Number which implement Serializable

**public class Cat implements Serializable {**

**private Tail tail = new Tail();**

**}**

**class Tail implements Serializable {**

**private Fur fur = new Fur();**

**}**

**class Fur {}**

In the above example Cat contains an instance of Tail, and both of those classes are marked Serializable, so no problems there. Unfortunately, Tail contains an instance of Fur that is not marked Serializable. So either Fur should be serializable

**class Fur implements Serializable{}**

Or instance of Fur in Tail should be transient

**class Tail implements Serializable {**

**private transient Fur fur = new Fur();**

**}**

We could also make our tail or fur instance members null, although this would make Cat serializable only for particular instances, rather than all instances

**serialVersionUID** : It's a good practice to declare a static serialVersionUID variable in every class that implements Serializable. The version is stored with each object as part of serialization. Then, every time the class structure changes, this value is updated or incremented. The serialVersionUID helps inform the JVM that the stored data may not match the new class definition. If an older version of the class is encountered during deserialization, a java.io.InvalidClassException may be thrown

# ObjectOutputStream and ObjectInputStream

The ObjectInputStream class is used to deserialize an object from a stream, while the ObjectOutputStream is used to serialize an object to a stream. They are high‐level streams that operate on existing streams.

**public ObjectInputStream(InputStream in) throws IOException**

**public ObjectOutputStream(OutputStream out) throws IOException**

Reading the file

**List<Employee> lst = new ArrayList<>();**

**lst.add(new Employee("employeeId", "name", "dept", 2));**

**lst.add(new Employee("employeeId", "name", "dept", 22));**

**try (ObjectOutputStream out = new ObjectOutputStream(new BufferedOutputStream(new FileOutputStream(fileOut)))) {**

**for (Employee emp : lst) {**

**out.writeObject(emp);**

**}**

**} catch (Exception e) {**

**e.printStackTrace();**

**}**

Writing to file. Unlike our earlier techniques for reading methods from an input stream, we need to use an infinite loop to process the data, which throws an EOFException when the end of the stream is reached.

**try (var in = new ObjectInputStream(new BufferedInputStream(new FileInputStream(fileOut)))) {**

**while (true) {**

**var object = in.readObject();**

**if (object instanceof Employee) {**

**lst.add((Employee) object);**

**}**

**}**

**} catch (EOFException e) {**

**e.printStackTrace();**

**} catch (ClassNotFoundException e) {**

**e.printStackTrace();**

**}**

If your program happens to know the number of objects in the stream, then you can call readObject() a fixed number of times, rather than using an infinite loop.

**try (var in = new ObjectInputStream(new BufferedInputStream(new FileInputStream(fileOut)))) {**

**int i = 0;**

**while (i > 2) {**

**var object = in.readObject();**

**if (object instanceof Employee) {**

**lst.add((Employee) object);**

**System.out.println(i);**

**i++;**

**}**

**}**

**System.out.println(lst);**

**} catch (Exception e) {**

**e.printStackTrace();**

**}**

If a variable is marked transient it will not be written to file. Also for static the last set value will be send to file

**class Employee implements Serializable {**

**private static final long serialVersionUID = 2L;**

**public transient String employeeId;**

**public static String name;**

**public String dept;**

**public int age;**

**}**

NIO.2

At its core, NIO.2 is a replacement for the legacy java.io.File class you learned about in Chapter 19. The goal of the API is to provide a more intuitive, more feature‐rich API for working with files and directories

The cornerstone of NIO.2 is the java.nio.file.Path interface. A Path instance represents a hierarchical path on the storage system to a file or directory. Both java.io.File and Path objects may refer to an absolute path or relative path within the file system. In addition, both may refer to a file or a directory. Unlike the java.io.File class, the Path interface contains support for symbolic links. A symbolic link is a special file within a file system that serves as a reference or pointer to another file or directory

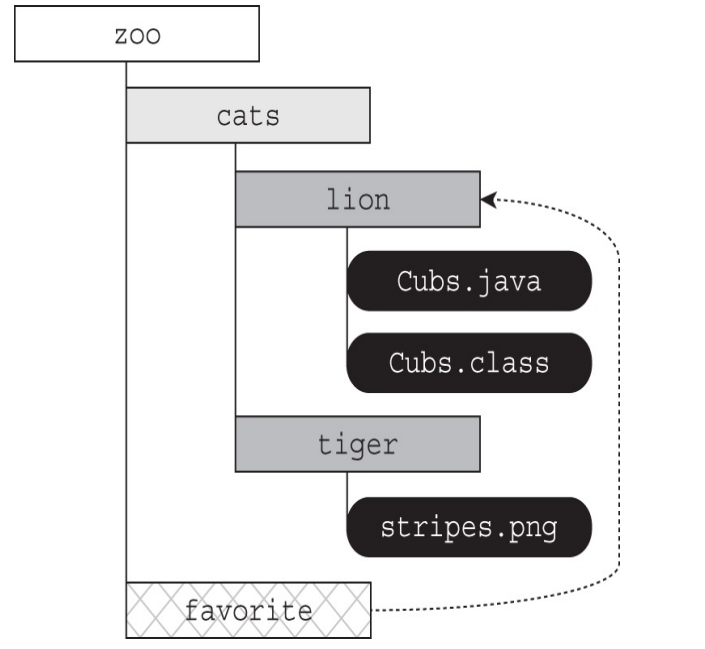


Figure shows a symbolic link from /zoo/favorite to /zoo/cats/lion. The lion folder and its elements can be accessed directly or via the symbolic link. For example, the following paths reference the same file:

/zoo/cats/lion/Cubs.java

/zoo/favorite/Cubs.java

A symbolic link (or "symlink") is a file system feature that can be used **to create a link to a specific file or folder**.

# Path

Since Path is an interface, we can't create an instance directly, as interfaces don't have constructors! Java provides a number of classes and methods that you can use to obtain Path objects. The simplest and most straightforward way to obtain a Path object is to use the static factory method defined within the Path interface.

**Path path1 = Path.of("pandas/cuddly.png");**

**Path path2 = Path.of("c:\\zooinfo\\November\\employees.txt");**

**Path path3 = Path.of("/home/zooDirectory");**

The Path.of() method also includes a varargs to pass additional path elements. The values will be combined and automatically separated by the operating system

**Path path1 = Path.of("pandas", "cuddly.png");**

**Path path2 = Path.of("c:", "zooInfo", "November","employees.txt");**

**Path path3 = Path.of("/", "home", "zooDirectory");**

The Path.of() method is actually new to Java 11. Another way to obtain a Path instance is from the java.nio.file.Paths factory class

**Path path = Paths.get("pandas/cuddly.png");**

**Path path = Paths.get("c:\\zooinfo\\November\\employees.txt");**

**Path path = Paths.get("/", "home", "zooDirectory");**

Another way to construct a **Path** using the **Paths** class is with a **URI** value. A uniform resource identifier (URI) is a string of characters that identify a resource. It begins with a schema that indicates the resource type, followed by a path value. URI throws a checked exception **URISyntaxException** so we need to handle it.

**URI a = new URI("file://icecream.txt");**

**Path b = Path.of(a);**

**Path c = Paths.get(a);**

A URI can be used for a web page or FTP connection.NIO.2 gives us the power to connect to both local and remote file systems, which is a major improvement over the legacy java.io.File class.

**Path path = Paths.get(new URI("http://www.wiley.com"));**

**Path path = Paths.get(new URI("ftp://username:secret@ftp.example.com"));**

FileSystems can also be used to create path and access files

**Path path1 = FileSystems.getDefault().getPath("pandas/cuddly.png");**

**Path path2 = FileSystems.getDefault().getPath("c:\\zooinfo\\November\\employees.txt");**

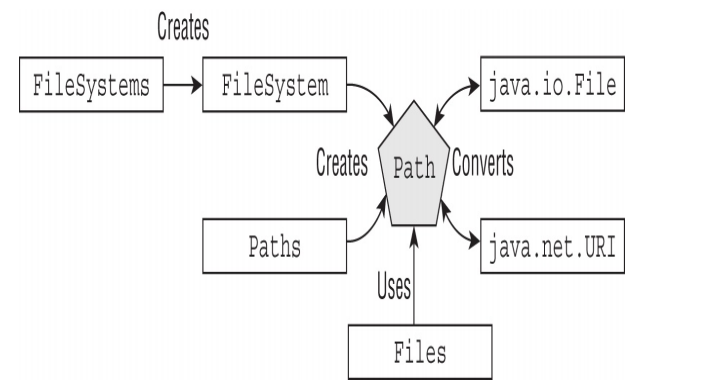
**Path path3 = FileSystems.getDefault().getPath("/home/zooDirectory");**

Last but not least, we can obtain Path instances using the legacy java.io.File class. In fact, we can also obtain a java.io.File object from a Path instance. These methods are available for convenience and also to help facilitate integration between older and newer APIs.

**File file = new File("husky.png");**

**Path path = file.toPath();**

**File backToFile = path.toFile();**



When working with NIO.2, keep an eye on whether the class name is singular or plural. The classes with plural names include methods to create or operate on class/interface instances with singular names. Remember, a Path can also be created from the Path interface, using the static factory of() method.

The java.io.File is the I/O class whereas Files is an NIO.2 helper class. Files operates on Path instances, not java.io.File instances

# Path Methods

**toString():** returns a String representation of the entire path.

The **getNameCount()** and **getName()** methods are often used in conjunction to retrieve the number of elements in the path and a reference to each element, respectively. These two methods do not include the root directory as part of the path.

**Path path = Paths.get("/land/hippo/harry.happy");**

**for (int i = 0; i < path.getNameCount(); i++) {**

**System.out.println(path.getName(i));**

**}**

**subpath(int i, int j)** returns a subpath from a path. The index starts with zero and the last index is ignored. Throws IllegalArgumentException when index are wrong

**System.out.println(path.subpath(1, 3)); //prints hippo/harry.happy**

**getFileName()** returns filename

**getParent()**  returns the parent directory but if the relative path is passed with file name only then null. Same thing for **getRoot()** method also.

**isAbsolute()**, returns true if the path the object references is absolute and false if the path the object references is relative. **toAbsolutePath()**, converts a relative Path object to an absolute Path object by joining it to the current working directory

**Path path = Paths.get("in.txt");**

**System.out.println(path.isAbsolute()); // false**

**System.out.println(path.toAbsolutePath());//C:\Users\justa\OneDrive\Desktop\JavaModule\src\javaPackage\in.txt**

**resolve()** method is used if you want to concatenate paths in a similar manner as we concatenate strings.

**public Path resolve(Path other)**

**public Path resolve(String other)**

The object on which the resolve() method is invoked becomes the basis of the new Path object, with the input argument being appended onto the Path

**Path path1 = Path.of("/cats/panther");**

**Path path2 = Path.of("food");**

**System.out.println(path1.resolve(path2)); // Prints /cats/panther/food**

The object on which the resolve is applied should be an absolute path and the param passed should be relative. If both are relative or both absolute or any mismatch like window and unix path then the 2nd param is returned.

**Path path3 = Path.of("C:src\\javaPackage\\new.txt");**

**System.out.println(path3.resolve("C:\\src\\javaPackage\\new2.txt")); // Both absolute 2nd returned**

**Path path3 = Path.of("/turkey/food");**

**System.out.println(path3.resolve("/tiger/cage"));// Both relative 2nd one returned**

**Path path3 = Path.of("C:\\src\\javaPackage\\in.txt");**

**System.out.println(path3.resolve("/tiger/cage")); // One Window path 2nd one unix 2nd one returned**

**relativize(Path path)** method helps to find the relative path from the object on which the method is applied to the path object which is passed. It's not necessary that the path should exist in the file system, java assumes if the files are accessible from the current directory where the program is run then what should be the relative path. Note, either both should be relative paths or both should be absolute paths otherwise it will throw an illegal argument exception. Also On Windows‐based systems, it also requires that if absolute paths are used, then both paths must have the same root directory or drive letter. For example, the following would also throw an IllegalArgumentException

**var path1 = Path.of("fish.txt");**

**var path2 = Path.of("friendly/birds.txt");**

**System.out.println(path1.relativize(path2)); //../friendly/birds.txt**

**System.out.println(path2.relativize(path1)); //../../fish.txt**

**Path path = Paths.get("c:\\primate\\chimpanzee");**

**Path path1 = Paths.get("d:\\storage\\bananas.txt");**

**path.relativize(path1); *// IllegalArgumentException***

**normalize()** method helps remove redundancies in a path.

**var p1 = Path.of("./armadillo/../shells.txt");**

**System.out.println(p1.normalize()); *// shells.txt***

**var p2 = Path.of("/cats/../panther/food");**

**System.out.println(p2.normalize()); *// /panther/food***

**var p3 = Path.of("../../fish.txt");**

**System.out.println(p3.normalize()); *// ../../fish.txt***

**toRealPath()** helps to verify the path actually exists within the file system. This method is similar to normalize(), in that it eliminates any redundant path symbols. It is also similar to toAbsolutePath(), in that it will join the path with the current working directory if the path is relative. toRealPath() will throw an exception if the path does not exist

**try {**

**System.out.println(Paths.get("/zebra/food.txt").toRealPath());**

**System.out.println(Paths.get(".././food.txt").toRealPath());**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

**If path exists both will print something like /horse/food.txt otherwise will throw exception**

# Files

Most of the methods we covered in the Path interface operate on theoretical paths, which are not required to exist within the file system. File classes are used if we want to rename a directory, copy a file, or read the contents of a file.

**Files.exists**

**var b1 = Files.exists(Paths.get("/ostrich/feathers.png"));**

**System.out.println("Path " + (b1 ? "Exists" : "Missing"));**

**isSameFile** Since a path may include path symbols and symbolic links within a file system, it can be difficult to know if two Path instances refer to the same file. isSameFile takes two Path objects as input, resolves all path symbols, and follows symbolic links. Despite the name, the method can also be used to determine whether two Path objects refer to the same directory

| **try {**  **System.out.println(Files.isSameFile(**  **Path.of("/animals/cobra"),**  **Path.of("/animals/snake")));**  **System.out.println(Files.isSameFile(**  **Path.of("/animals/monkey/ears.png"),**  **Path.of("/animals/wolf/ears.png")));**  **} catch (IOException e) {**  **e.printStackTrace();**  **}** |  |
| --- | --- |

Since cobra is a symbolic link to the snake file, the first example outputs true. In the second example, the paths refer to different files, so false is printed.

**createDirectory()** will create a directory and throw an exception if it already exists or the paths leading up to the directory do not exist

The **createDirectories()** works just like the java.io.File method mkdirs(), in that it creates the target directory along with any nonexistent parent directories leading up to the path. If all of the directories already exist, createDirectories() will simply complete without doing anything. This is useful in situations where you want to ensure a directory exists and create it if it does not.

**try {**

**Files.createDirectory(Path.of("dir"));**

**Files.createDirectories(Path.of("dir/dir2"));**

**} catch (IOException e) {**

**e.printStackTrace();**

**copy** The method copies a file or directory from one location to another using Path objects. When directories are copied, the copy is shallow. A shallow copy means that the files and subdirectories within the directory are not copied. A deep copy means that the entire tree is copied, including all of its content and subdirectories

**try {**

**Files.copy(Path.of("in.txt"), Path.of("folder/in.txt")); //in.txt and folder should exist**

**} catch (IOException e) { //Otherwise will throw exception**

**e.printStackTrace();**

**}**

By default, if the target already exists, the copy() method will throw an exception. You can change this behavior by providing the StandardCopyOption enum value REPLACE\_EXISTING to the method. The following method call will overwrite the movie.txt file if it already exists:

**try {**

**Files.copy(Path.of("in.txt"), Path.of("folder/in.txt"), StandardCopyOption.REPLACE\_EXISTING);**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

The Files class also includes two copy() methods that operate with I/O streams.

**move()** method provides ways to move or rename a file or directory.

**try {**

**Files.move(Path.of("in.txt"), Path.of("out.txt")); //Since the location is same file will be renamed**

**} catch (IOException e) {**

**e.printStackTrace();**

**}**

**Files.move(Path.of("in.txt"), Path.of("folder/out.txt"));**

**Files.move(Path.of("folder"), Path.of("folder2"));**

An atomic move is one in which a file is moved within the file system as a single indivisible operation which prevents an incomplete or partially written file. If the file system does not support this feature, an AtomicMoveNotSupportedException will be Thrown. Note that while ATOMIC\_MOVE is available as a member of the StandardCopyOption type, it will likely throw an exception if passed to a copy() method.

**Files.move(Path.of("mouse.txt"), Path.of("gerbil.txt"), StandardCopyOption.ATOMIC\_MOVE);**

**delete() and deleteIfExists() -** throw an exception if operated on a nonempty directory. In addition, if the path is a symbolic link, then the symbolic link will be deleted, not the path that the symbolic link points to.The delete() method throws an exception if the path does not exist, while the deleteIfExists() method returns true if the delete was successful.

**Files.delete(Paths.get("/vulture/feathers.txt"));**

**Files.deleteIfExists(Paths.get("/pigeon"));**

**newBufferedReader(),newBufferedWriter**

**var path = Path.of("in.txt");**

**var outPath = Path.of("out.txt");**

**try (var reader = Files.newBufferedReader(path);**

**var writer = Files.newBufferedWriter(outPath)) {**

**String currentLine = null;**

**while ((currentLine = reader.readLine()) != null) {**

**writer.write(currentLine);**

**writer.newLine();**

**}**

**} catch (Exception e) {**

**e.printStackTrace();**

**}**

**readAllLines()**

**var path = Path.of("in.txt");**

**try {**

**List<String> lines = Files.readAllLines(path);**

**lines.forEach(System.out::println);**

**} catch (IOException e1) {**

**e1.printStackTrace();**

**}**