

Final (Attributes/Parameters)

Variable	Method	Class
Constant	No overriding	No inheritance

Initialisation

- 1) Default-Values ↓
- 2) Attribute Assignments
- 3) Initialisation block
- 4) Constructor

Default Values

Type	Default	Type	Default
boolean	false	char	'\u0000'
byte	0	short	0
int	0	long	0L
float	0.0f	double	0.0d

Types

Type	Size (bit)	From	To
byte	8	−128	127
short	16	−32'768	32'767
char	16	UTF-16 chars	
int	32	−2 <sup>31</sup>	2 <sup>31</sup> − 1
long	64	−2 <sup>63</sup>	2 <sup>63</sup> − 1
float	32	±1.4 · 10 <sup>−45</sup>	±3.4 · 10 <sup>38</sup>
double	64	±4.9 · 10 <sup>−324</sup>	±1.7 · 10 <sup>308</sup>

```
// short * int      => int
// float + int      => float
// int / double     => double
// int + long * float => float
// 0.0 / 0.f        => NaN
```

```
long l = 1L;
long ll = 0b1L;
float f = 0.0f;
double d = 0.0d;
```

```
12 = '.'; // implicit int/char conversion
0.1 + 0.1 ≠ 0.2; // true
5/2 = 2; // true, int div truncates to 0
NaN = NaN; // false
Integer.MAX_VALUE + 1 = Integer.MIN_VALUE;
1 / 0.0 = Double.POSITIVE_INFINITY; // true
```

```
var ints = new ArrayList<Integer>();
int[] jnts = new int[69];
int[][] matrix = new int[6][9];
```

```
if (obj instanceof ArrayList<Integer>)
    ((ArrayList<Integer>)obj).add(2);
```

```
public List<String> method(
    BiFunction<Integer, String, List<String>> fn) {
    return fn.apply(5, "FooBar");
}
```

Implicit casting

No information loss int→float, to larger type int→long  
Sub→Super is implicit, Super→Sub ClassCastException

```
// explicit casting
float f = (float) 1;
// type conversion
Integer.parseInt("2");
Float.parseFloat("2.0");
```

Widening primitive conversion

- If either operand is double, the other is converted to double
- Otherwise, if either is float, the other is converted to float
- Otherwise, if either is long, the other is converted to long
- Otherwise, both operands are converted to type int

Variable args

```
static int sum(int... numbers) {
    int sum = 0;
    for (int i = 0; i < numbers.length; i++)
        sum += numbers[i];
    return sum;
}
```

Misc

```
int[] intarr = new int[] {1, 2, 3, 4, 5};
int[] sub = Arrays.copyOfRange(intarr, 1, 3); // 2,3
```

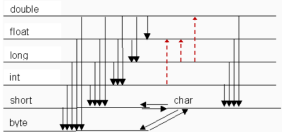
```
var intList = new ArrayList<Integer>();
intarr.length;
intList.size();
```

```
// Multiply first to not lose precision
int percent = (int)((filled * 100) / capacity);
```

obj.clone();

```
Math.min(x, y);
Math.max(x, y);
```

Rekapitulation: Primitive Datentypen → explizit  
----- implizit (mit evtl. Genauigkeitsverlust)  
Sonstige Richtungen implizit



Equality

```
s.equals(sOther); // Strings / Objects
Arrays.equals(a1, a2); // arrays
Arrays.deepEquals(a1, a2); // nested arrays
```

```
class Student extends Person {
    @Override
    public boolean equals(Object obj) {
        if (obj == null) return false;
        if (getClass() ≠ obj.getClass()) return false;
        if (!super.equals(obj)) return false;
        Student other = (Student) obj;
        return getNumber() = other.getNumber();
    }
}
```

Strings

```
String multiline = """
    Hello, "world"
    """;
"a:b:c".split(":",2).length = 2; // true
String str = Integer.toString(123456789);
str.length(); // 9
str.charAt(1); // 2
str.toUpperCase();
str.toLowerCase();
str.trim();
str.substring(1, 3); // 2,3
```

String pooling

```
String first = "hello", second = "hello";
System.out.println(first == second); // true
```

```
String third = new String("hello");
String fourth = new String("hello");
System.out.println(third == fourth); // false
System.out.println(third.equals(fourth)); // true
```

```
String a = "A", b = "B", ab = "AB";
System.out.println(a + b == ab); // false
```

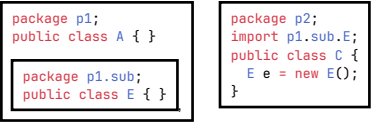
```
final String d = "D", e = "E", de = "DE";
System.out.println(d + e == de); // true
```

Visibility

public	all classes
protected	package + sub-classes
private	only self
(none)	all classes in same package

Packages

p1.sub won't be automatically imported in p1.  
Package name collisions: first gets imported.



```
package p1; // public class A { }
package p2; // public class A { }
```

```
// OK
import p1.A;
import p2.*;
```

```
// reference to A is ambiguous
import p1.*;
import p2.*;
```

```
// sin, PI
import static java.lang.Math.*;
```

Modules

```
// ./foo/module-info.java
module foo.bar.baz {
    exports com.my.package.foo;
}
```

```
// ./main/module-info.java
module main.module {
    requires com.my.package.foo;
}
```

Enums

```
public enum Weekday {
    MONDAY(true),
    TUESDAY(true),
    WEDNESDAY(true),
    THURSDAY(true),
    FRIDAY(true),
    SATURDAY(false),
    SUNDAY(false);

    private boolean workDay;
```

```
Weekday(boolean workDay) { // private constructor
    this.workDay = workDay;
}

public boolean isWorkDay() {
    return workDay;
}
```

```
switch (wd) {
    case Weekday.MONDAY:
    case Weekday.TUESDAY:
        System.out.println("First two");
        break;
    default:
        System.out.println("Rest");
}
```

```
public enum Level {
    LOW,
    MEDIUM,
    HIGH
}
```

Switch

```
switch (x) {
    case 'a':
        System.out.println("1");
        break;
    default:
        System.out.println("2");
}

int y = switch (x) {
    case 'a' -> 1;
    default -> 2;
}
```

Shadowing

Variables with same names in same scope

Hiding

Variables with same names in different classes  
Gets statically chosen by compiler

```
super.description = ((Vehicle)this).description
super.super // doesn't exist, use v
((SuperSuperClass)this).variable
// for multiple parent interfaces v
ParentInterface1.super.defaultMethod();
// ^ executes defaultMethod on ParentInterface1
ParentInterface2.super.defaultMethod();
```

Overloading

Methods with same names but different parameters  
Gets statically chosen by compiler

```
void print(int i, double j) { } // 1
void print(double i, int j) { } // 2
void print(double i, double j) { } // 3
```

```
print(1.0, 2.0); // 3
print(1,2); // error: reference to print is ambiguous
print(1.0, 2); // 2
print(2.0, (double) 2); // 3
```

Overriding

Methods with same names and signatures  
Dynamically chosen (Dynamic dispatch / Virtual call)  
Error: Cannot override the final method...  
Error: Cannot be subclass of final class...

```
class Fruit {
    void eat(Fruit f) { System.out.println("1"); }
}

class Apple extends Fruit {
    void eat(Fruit f) { System.out.println("2"); }
    void eat(Apple a) { System.out.println("3"); }
}

Apple a = new Apple();
Fruit fa = new Apple();
Fruit f = new Fruit();
```

```
a.eat(fa); // 2
a.eat(a); // 3
fa.eat(a); // 2
fa.eat(fa); // 2
f.eat(fa); // 1
f.eat(a); // 1
((Fruit) a).eat(fa); // 3
((Apple) fa).eat(a); // 2
((Apple) f).eat(a); // ClassCastException
```

Abstract classes

```
public abstract class Vehicle {
    private int speed;
    public abstract void drive();
    public void accelerate(int acc) {
        this.speed += acc;
    }
}

public class Car extends Vehicle {
    @Override
    public void drive() { }
    @Override
    public void accelerate (int acc) { }
```

Interfaces default methods

```
interface Vehicle {
    default void printModel() {
        System.out.println("Undefined vehicle model");
    }
}
```

Interfaces

Cannot have Attributes

```
interface RoadV {
    int MAX_SPEED = 120;
    void drive();
}

interface WaterV {
    int MAX_SPEED = 80;
    void drive();
}

class AmphibianMobile implements RoadV, WaterV {
    @Override // because ambiguous
    public void drive() {
        println(RoadV.MAX_SPEED); // MAX_SPEED ambiguous
    }
}
```

```
interface RoadV { String getModel(); }
interface WaterV { int getModel(); }
// Error, because of different return types
class AmphibianMobile implements RoadV, WaterV { }
```

Anonymous Classes

```
var v = new RoadV() {
    @Override
    public void drive() {
        System.out.println("Anon");
    }
}
```

Inheritance

```
public class Vehicle {
    private int speed;
    public Vehicle(int speed) {
        this.speed = speed;
    }
}
```

```
public class Car extends Vehicle {
    private int doors;
    public Car(int speed, int doors) {
        super(speed);
        this.doors = doors;
    }
}
```

```
}

Car c = new Car(); // Points to Car
Vehicle v = new Car(); // Points to Car
Object o = new Car(); // Points to Car
// ^static ^dynamic
Car c = (Car) new Vehicle(); // ClassCastException
```

More Inheritance

```
public class Qwer {
    public void print() {
        System.out.println("1");
    }
}
```

```
public class Asdf extends Qwer {
    @Override
    public void print() {
        System.out.println("2");
    }
    public void dostuff () { }
```

```
var x = new Asdf();
x.print(); // 2
((Qwer) x).print(); // 2
((Qwer) x).dostuff(); // cannot find symbol
```

Static Type: According to var declaration at compiletime  
Dynamic Type: Type of the instance at runtime

## Iterators

```
Iterator<String> it = stringList.iterator();
while (it.hasNext()) {
    String s = it.next();
    System.out.println(s);
}
```

Mutating Collection while iterating over it: ConcurrentModificationException

## Exceptions

Error	Exception
Critical, don't handle	Runtime, handleable
OutOfMemoryError, StackOverflowError, AssertionError	IOException
Checked	Unchecked
Must be handled (or throws-declaration)	Not necessary
Checked by compiler	Compiler doesn't check
Exception, not RuntimeException	RuntimeException, Error

Child Exception gets caught in catch clause with parent class

```
void test() throws ExceptionA, ExceptionB {
    String c = clip("asdf");
    throw new ExceptionB("wack");
}

// finally ALWAYS executes, even on unhandled Exc.
try {
    test();
} catch (ExceptionA | ExceptionB e) {
    // ...
} finally { }
```

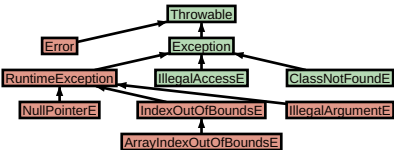
```
try { ... } catch (NullPointerException e) {
    throw e; // →leaves blocks→
} catch (Exception e) {
    // above e won't get caught!
} finally {
    // will still get executed
}
```

1 / 0; // ArithmeticException div by zero

```
String s = "";
s = null;
s.toUpperCase(); // NullPointerException
```

```
int[] arr = new int[] {1, 2, 3};
int elem = arr[8]; // ArrayIndexOutOfBoundsException
```

Unchecked Checked



## Important stuff

- Hashing should be added to equals fn's for strict equality
- Check if input == null
- Check if array.length == 0
- IllegalArgumentException("reason")
- try/catch finally block **always** executes

## IO

```
try (var fr = new FileReader("text.txt")) {
    int input = fr.read();
    while (input ≥ 0) {
        if (input == ';') { /* do something */ }
        input = fr.read();
    }
}
```

```
try (FileWriter writer = new FileWriter("out.txt",
    StandardCharsets.UTF_8, true)) { // append
    writer.write("weeoo\n");
}
```

```
try {
    var input = new FileInputStream("text.txt");
    int i = input.read();
    while(i ≠ -1) {
        System.out.print((char)i);
        i = input.read();
    }
    input.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

```
try (BufferedReader reader = new BufferedReader(
    new FileReader("text.txt",
        StandardCharsets.UTF_8))) {
    String line;
    while ((line = reader.readLine()) ≠ null) {
        System.out.println(line);
    }
}
```

```
try {
    FileReader reader = new FileReader("in.txt");
    FileWriter writer = new FileWriter("out.txt")
} {
    int i = input.read();
    while(i ≥ 0) {
        writer.write(i);
        i = input.read();
    }
}
```

## Try with

```
try (var output = new FileOutputStream("f.txt")) {
    output.write("Hello".getBytes());
} catch (IOException e) {
    System.out.println("Error writing file");
} finally {
    System.out.println("Done");
}
```

## Serializing

```
class X implements Serializable { }
// Serializing
try (var stream = new ObjectOutputStream(
    new FileOutputStream("s.bin"))) {
    stream.writeObject(new X());
}
// Deserializing
try (var stream = new ObjectInputStream(
    new FileInputStream("s.bin"))) {
    X x = (X) stream.readObject();
}
```

## Function

```
public interface Function<T, R> {
    R apply(T t);

    static <T> Function<T, T> identity();

    <V> Function<T, V> andThen(
        Function<? super R, ? extends V> after);

    <V> Function<V, R> compose(
        Function<? super V, ? extends T> before);
}
```

## Predicate

```
public interface Predicate<T> {
    boolean test(T t);
}
```

```
static void removeAll(Collection<Person> collection,
    Predicate criterion) {
    var it = collection.iterator();
    while (it.hasNext())
        if (criterion.test(it.next()))
            it.remove();
}
```

## Comparable

```
public interface Comparable<T> {
    int compareTo(T obj);
}
```

```
var l = new ArrayList<Integer>(asList(3,2,4,5,1));
l.sort((a, b) → a > b ? 1 : -1); // =
l.sort((a, b) → a - b); // 1,2,3,4,5
```

```
class Person implements Comparable<Person> {
    private final String firstName, lastName;
    @Override
    public int compareTo(Person other) {
        int result = lastName.compareTo(other.lastName);
        if (result == 0)
            result = firstName.compareTo(other.firstName);
        return result;
    }
}
```

```
static int compareByAge(Person a, Person b) {
    return Integer.compare(a.getAge(), b.getAge());
}
```

```
List<Person> people = ...;
Collections.sort(people);
people.sort(Person::compareToByAge);
```

## Comparator

```
class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person a, Person b) {
        return Integer.compare(a.getAge(), b.getAge());
    }
}
```

```
Collections.sort(people, new AgeComparator());
people.sort(new AgeComparator());
```

```
people.sort(Comparator
    .comparing(Person::getAge)
    .thenComparing(Person::getFirstName)
    .reversed())
```

```
Comparator.comparing(Person::getName,
    (s1, s2) → s2.compareTo(s1));
// =
Comparator.comparing(Person::getName).reversed();
```

```
Comparator<T> nullsLast(Comparator<T> c);
Comparator<T> nullsFirst(Comparator<T> c);
Comparator<T> comparing(Function<T,U> keyExtractor,
    Comparator<U> c);
Comparator<T> comparingInt(ToIntFunction<T,U> f);
```

## FunctionalInterface

Any interface with a single abstract method is a functional interface

```
@FunctionalInterface
public interface ShortToByteFunction {
    byte applyAsByte(short s);
}

@FunctionalInterface
public interface PersonStringifier {
    String getNameAndAge(Person p);
}
```

## Collection

```
boolean add(E e);
boolean remove(Object o);
boolean equals(Object o);
int hashCode();
int size();
boolean isEmpty();
Object[] toArray();
void clear();
boolean contains(Object o);
boolean addAll(Collection<? extends E> c);
boolean containsAll(Collection<?> c);
boolean removeAll(Collection<?> c);
boolean retainAll(Collection<?> c);
```

```
Set<String> noDup = new HashSet<>();
```

## Collection implementations

```
// List
int indexOf(Object o);
int lastIndexOf(Object o);
E get(int index);
subList(int from, int to);
void sort(Comparator<? super E> c);

// Stack
E peek();
E pop();
E push(E item);
boolean empty();
int search(Object o);

// Queue
E element(); // throws → peek(); doesn't
E remove(); // throws → poll(); doesn't
boolean add(E e); // throws → offer(E e); doesn't
```

```
// Set
// (I) SortedSet → (C) TreeSet
// (C) HashSet, (C) LinkedHashSet
```

```
// Map
// HashMap
boolean containsKey(Object key);
boolean containsValue(Object value);
Set<Map.Entry<K, V>> entrySet();
V get(Object key);
V put(K key, V value);
V putIfAbsent(K key, V value);
V replace(K key, V value);
V remove(Object key);
V getOrDefault(Object key, V defaultValue);
Set<K> keySet();
Collection<V> values();
```

## Lambdas

```
String pattern = readFromConsole();
// vvv not final → Error
while (pattern.length() == 0)
    pattern = readFromConsole();
Utils.removeAll(people, p →
    p.getLastName().contains(pattern));
// local variable ... referenced from a lambda
expression must be final or effectively final
```

```
// Predicate :: a → boolean
Predicate<Integer> isLarge = (v) → v > 69428;
// Function :: a → b
Function<Integer, String> str = (v) → "" + v;
// Supplier :: a
Supplier<String> hello = () → "Hello, World!";
// Consumer :: a → void
Consumer<Integer> consommer = (v) → log(v);
// UnaryOperator :: a → a
UnaryOperator<Integer> more = (v) → v * v;
// BinaryOperator :: a → a → a
BinaryOperator<Integer> less = (a, b) → a - b;
```

## Streams

```
import java.util.stream.*;
```

```
people
    .stream()
    .distinct()
    .filter(p → p.getAge() ≥ 18)
    .skip(5)
    .limit(10)
    .map(p → p.getLastName())
    .sorted()
    .forEach(System.out::println);
```

```
people
    .stream()
    .reduce(0, (acc, cur) → acc + cur.getAge());
```

```
List.stream().mapToInt(Integer::intValue);
List.stream().mapToInt(Integer::parseInt);
```

## Optional (Haskell / Rust in boat)

```
T get(); // NoSuchElementException
boolean isPresent();
void ifPresent(Consumer<T> consumer);
T orElse(T other);
static Optional<T> empty();
static Optional<T> of(T value);
```

## Methods

```
boolean allMatch(Predicate<T> predicate);
boolean anyMatch(Predicate<T> predicate);
boolean noneMatch(Predicate<T> predicate);
static Stream<T> concat(Stream<T> a, Stream<T> b);
Stream<T> distinct();
Stream<R> flatMap(Function<T, R>> mapper);
Stream<T> limit(Long maxSize);
Stream<T> skip(Long maxSize);
Stream<sorted>(Comparator<T> comparator);
```

## Terminal operations

```
Optional<T> min(Comparator<T> comparator);
Optional<T> max(Comparator<T> comparator);
Optional<T> findAny();
Optional<T> findFirst();
void forEach(Consumer<T> action);
long count();
void forEachOrdered(Consumer<T> action);
// IntStream
average();
sum();
```

## Collectors

```
// List
s.collect(Collectors.toList());
// TreeSet
s.collect(Collectors.toCollection(TreeSet::new));
// String
s.collect(Collectors.joining(", "));
// Integer
s.collect(Collectors.summingInt(Person::getAge));
// Map<String, Person>
s.collect(Collectors.groupingBy(Person::getCity));
// Map<String, Integer>
s.collect(Collectors.groupingBy(Person::getCity,
    Collectors.summingInt(Person::getSalary));
// Map<boolean, List<Person>>
s.collect(Collectors.partitioningBy(s →
    s.getAge() > 18))
```