

Georgiy Shevoroshkin

- Function<T,V> Predicate<T> Stream<T> Collection<T>
- cannot override final methods
- cannot be subclass of final class

Final (Attributes/Parameters)

TODO

Static (Attributes/Methods)

TODO

Private (Attributes/Methods)

TODO

Types

```
long l = 1L; long ll = 0b1l;
float f = 0.0f; long d = 0.0d;
String multiline = """
    Hello, "world"
""";
"abc".split(":",2).length == 2; // true
var ints = new ArrayList<Integer>();
boolean isTrue = 0.1 + 0.1 != 0.2;
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");
}
```

Misc

```
int[] intarr = new int[] {1, 2, 3, 4, 5};
int[] sub = Arrays.copyOfRange(intarr, 1, 3); // 2,3
var intlist = new ArrayList<Integer>();
intlist.add(1);
intarr.length; intlist.size();
// Multiply first to not lose precision
int percent = (int)((filled * 100) / capacity);
obj.clone();
5/2 == 2; // true, int div truncates to 0
Double.POSITIVE_INFINITY; // exists
```

Variable args

```
long l = 1L; long ll = 0b1l;
static int sum(int... numbers) {
    int sum = 0;
    for (int i = 0; i < numbers.length; i++) sum += numbers[i];
    return sum;
}
```

Implicit casting

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

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

Equality

```
s.equals(s0ther); // 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();
    }
}
```

Hashing

Should be added to equals fn's for strict equality

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.equals(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
```

Switch

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

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;
import p1.sub.E;
public class C {
 E e = new E();
}

package p1; public class A { }
package p2; public class A { }
import p1.A; import p2.*; // OK
import p1.*; import p2.*; // reference to A is ambiguous
import static java.lang.Math.*; // sin, PI

TODO

Anonymous Classes TODO

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

IO

TODO

Enums

TODO

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

private boolean workDay;
```

Weekday

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

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
```

TODO

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
```

Hiding

```
super.description = ((Vehicle)this).description
super.super // doesn't exist, use v
((SuperSuperClass)this).variable
```

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) { }
}
```

Dynamische Typ: Effektiver Typ der Instanz zur Laufzeit

Compiler Quirks

Iterators

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

Mutating Collection whilst iterating over it: ConcurrentModificationException

Set: No duplicates

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();
}
```

Child Exception gets caught in a 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;
} catch (Exception e) {
    // ABOVE e WON'T GET CATCHED!
}
```

Unchecked

Checked

Try with

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

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();
}
```

Collection

boolean add(E e);	boolean remove(Object o);
boolean equals(Object o);	int hashCode();
int size();	boolean isEmpty();
boolean contains(Object o);	

Comparable

```
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
```

Person

```
class Person implements ComparablePerson {
    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 p1, Person p2) {
    return Integer.compare(p1.getAge(), p2.getAge());
}
List<Person> people = ...
Collections.sort(people);
people.sort(Person::compareByAge);
```

AgeComparator

```
class AgeComparator implements Comparator<Person> {
    @Override
    public int compare(Person p1, Person p2) {
        return Integer.compare(p1.getAge(), p2.getAge());
}
```

Exceptions

Error	Exception
Critical, don't handle	Runtime, handleable
OutOfMemoryError, StackOverflowError	IOException

Checked	Unchecked
Must be handled (or throws-declaration)	Not necessary
Checked by compiler	Compiler doesn't check
Exception, not RuntimeException	RuntimeException, Error

```
Georgiy Shevoroshkin
}
}
Collections.sort(people, new AgeComparator());
people.sort(new AgeComparator());
```

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

Predicate

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

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
```

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);

list.stream().mapToInt(Integer::intValue);
```

Terminal operations:

forEach(Consumer), forEachOrdered(Consumer), count(),
min(), max(), average(), sum(), findAny(), findFirst()