

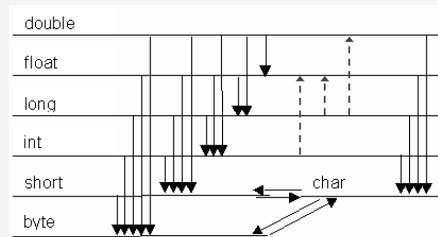
Java basics

<code>x = a ? b : c</code>	a true? x = b, a false? x = c
<code>0.1 + 0.1 == 0.3</code>	False, workaround: <code>Math.abs(0.1 + 0.1 - 0.2e6) < 0.2e6</code>
<code>a && b</code>	Only check b if a is true
<code>a b</code>	Only check b if a is false
<code>0b11001</code>	binary, leading "0b"
<code>0x1e</code>	hexadecimal, leading "0x"
<code>010 != 10</code>	Leading 0 means octal
<code>'A' == 'a'</code>	False (case sensitive)
<code>'A' < 'B'</code>	True

Java primitive data types

boolean	true, false
char	16 bit, UTF-16
byte	8 bit, -128...127
short	16 bit, -32.768 ... 32.767
int	32 bit, -2^{31} to $+2^{31}-1$
long	64 bit, -2^{63} to $+2^{63}-1$, <code>long x = 100L;</code>
float	32 bit, <code>float x = 100f;</code>
double	64 bit, <code>double x = 100d;</code>

Java type casting



red arrows = implicit (probably information loss due to inaccurate data format)
black arrows = explicit cast (heavy information loss possible --> developer)

Java interfaces

Methods in interfaces are implicitly public and abstract and can't be private.
Only constant variables are allowed: `public static final`
`int HIGHWAY_MAX_SPEED = 60;` optional.
Same named methods in basic interface and super interface must have the same return type.
Same named variables in basic interface and super interface can have different return types.
Default methods: Basic interface doesn't have to implement default methods. If one method is not overridden, it just takes the default method.

Java reference types

```
int[] x = new int[10];

int[][] m = new int[2][3];

Arrays.equals(a, b)

Arrays.deepEquals(a, b)

public enum Weekday{ MONDAY, ..., SUNDAY }
```

```
String a = "Progl";

String c = new String("Progl");

a.equals(b)
```

Java reference types (cont)

```
String s1 = new String("Progl");
String s2 = new String("Progl");
s1.equals(s2);
```

Java equals() example

```
@Override
public boolean equals (Object obj) {
    if (obj == null) {
        return false;
    } else if (getClass() != obj.getClass()) {
        return false;
    } else if (!super.equals(obj)) {
        return false;
    }
    // normal Array
    public int[] x = {1, 2, 3};
    // 2 dimensional array
    Student other = (Student) obj;
    return regNumber == other.regNumber;
}
// Compare array content
// Compare x dimensional array
Enum
```

If equals() is overridden, hashCode() must be overridden as well.
hashCode() is used to generate a hash code for the object.
new String-Object

Keywords

public	can be seen by all that imports this package
protected	can be seen by all classes in this package and all subclasses of this
package	can be seen by all classes in this package
private	can be seen only by this class
static	only once for all instances of this class

Keywords (cont)

final can only be defined once and not changed later. Class: no subclasses, Method: no overriding

static Means this is a constant
final

Javadoc

Start with /**	end with */	each line *
@author name	author	class / interface
@version number	version	class / interface
@param name	parameter	method
@return description	returnvalue	method
@throws/@exception type description	potential exception	method
@deprecated description	deprecated (outdated)	method

Java hashCode() example

```
public int hashCode() {
    return firstName.hashCode() + 31 * surName.hashCode();
}
```

Java compareTo example

```
class Person implements Comparable<Person> {
    private String firstName,
    lastName;
    // Constructor...
    @Override
    public int compareTo(Person other) {
        int c = compareToString(lastName, other.lastName);
    }
}
```

Java compareTo example (cont)

```
> if (c != 0) { return c; }
    else { return compareToStrings(firstName,
    other.firstName); }
}
private int compareToStrings(String a, String
b) {
    if (a == null) { return b == null ? 0 : 1; }
    else { return a.compareTo(b); }
}
}
```

Java collections

```
ArrayList<Object> al = new ArrayList<Object>();
LinkedList<Object> ll = new LinkedList<Object>();
Set<String> s1 = new TreeSet<String>(); or Set<String> s2 = new HashSet<String>();
// TreeSet is always efficient. HashSet = unordered
```

Java collections (cont)

```
Map<Integer, Object> m1 = new HashMap<Integer, Object>();
// or Map<Integer, Object> m2 = new TreeMap<Integer, Object>();
// Iterators
```

Java inheritance

```
Vehicle v1 = new Car();
Vehicle v = new Vehicle(); Car c = (Car) v;
if (v instanceof Car) { Car c = (Car) v; }
// super variable
```

```
Vehicle v = new Vehicle(); Car c = (Car) v;
if (v instanceof Car) { Car c = (Car) v; }
// super variable
```

```
super.variable
((SuperSubClass) this).variable
```

Dynamic dispatch: Methods: from dynamic type and variables from static type.

Java Lambdas / Stream API

```
Collection<Person> sortedPeople = (p1, p2) -> p1.getAge() > p2.getAge() ? p1 : p2;
// or
Collection<Person> sortedPeople = people.stream().sorted().collect(toList());
```

```
Collection<Person> sortedPeople = people.stream().sorted().collect(toList());
// or
Collection<Person> sortedPeople = people.stream().sorted().collect(toList());
```

```
Collection<Person> sortedPeople = people.stream().sorted().collect(toList());
```

```
Random random = new Random(4711);
// or
Random random = new Random(4711);
```

```
List<Person> list = people.stream().collect(toList());
```

Java Lambdas / Stream API (cont)

```
Collection<Person> sortedPeople = people.stream().sorted().collect(toList());
// or
Collection<Person> sortedPeople = people.stream().sorted().collect(toList());
```

Possible Stream API operations:

```
Collection<Person> sortedPeople = people.stream().sorted().collect(toList());
// or
Collection<Person> sortedPeople = people.stream().sorted().collect(toList());
```

Comparator & Methodreference

Interface used to compare 2 Objects (before you used lambdas).

Contains the method `compareTo()` which you need to override. Returns positive number if o1 is bigger than o2 and negative if opposite. 0 means that they are equal.

Instead of a comparator use methodreference class: `methodName` eg `Person Comparator::compareTo`

Nested class

Use this if a class is only used in another class. No separate classfile.

The inner class can use all members of the outer class (this includes private members).

Instantiation from outside eg `Polygon.Polygon` or `new Polygon()`.

Can be declared in a method -> All variables from outside are getting final.

```
int getMaxSpeed() {return 300; }
// or
int getMaxSpeed() {return 300; }
```

Java own exception class

```
Exception {
    private static final long serialVersionUID = 1L;

    public MyException(String msg) {
        super(msg);
    }

    for each (System.out::println);
}
```

Java package import conflict order

1. own class (incl. nested class)
2. single type imports -> `import p2.A;`
3. type in own package -> `package p1;`
4. Import on demand -> `import p2.*;`

Java regex

<code>1*</code>	<code>0 to *</code>
<code>1+</code>	<code>1 to *</code>
<code>1{2,5}</code>	<code>min 2 max 5 (11..111)</code>
<code>1{2,}</code>	<code>min 2 max * (11, 111, ...)</code>
<code>1{3}</code>	<code>exactly 111</code>
<code>-?1</code>	<code>-1 or 1, "-" is optional</code>
<code>Mo Di</code>	<code>Or</code>
<code>[a-zA-Z]</code>	<code>any letter from a to z</code>
<code>[a-zA-Z_]</code>	<code>any letter from a to z or underscore</code>
<code>\s</code>	<code>whitespace</code>
<code>[^abc]</code>	<code>anything except a, b, c</code>
<code>\$</code>	<code>end of string</code>
<code>\d</code>	<code>any digit</code>
<code>\D</code>	<code>not digit</code>
<code>(?<Group1>REGEX)</code>	<code>name capture group - r.g. group("Group")</code>

Example: Check daytime: `([0-1]?[0-9] | 12[0-3]): [0-5][0-9]`

Java regex code example

```
String input =
scanner.nextLine();
Pattern pattern = Pattern.compile ("([ 0-2 ]?[ 0-9 ]): -
([0 -5] [0- 9]) ");
Matcher matcher = pattern.matcher (input);
if (matcher.matches()) {
    String hoursPart = matcher.group(1);
    String minute sPart =
matcher.group(2);
    System.out.println (..);
}
```

Java JUnit

Java JUnit examples

```
@Test
public void testPrime_2() {
    assertEquals("2 is prime",
utils.isPrime(2));
}
```

Java generics

Example: class Node<T extends Number & Serializable>{ ... } Node<Integer> n = new Node<Integer>(1);

Wildcard type: Node<?> undefined Node; undefinedNode = new Node<Integer>(1);

static variables with generics NOT allowed eg static T maxSpeed;

Generic Method: public <T> T majority(T x, T y, T z) { if (x.equals(y)) { return x; } else if (x.equals(z)) { return x; } else if (y.equals(z)) { return y; } else { return null; } } **Call:** Double d = test.<Double>majority(1.0, 3.141, 2.718);

Rawtype: like you would insert Object -> you need to down cast the elements. e.g: Node n; // subDepartment

Serializable

Is a marker interface (is empty, just says that this class supports it)

Use it to say the developer that he can serialize objects of this class, which means he can write them in a bytecode and export them. Always serialize all the objects contained in the main object

Serializable (cont)

Use serialVersionUID to identify your class (no need to change it)

Example: OutputStream fos= new FileOutputStream("file.txt");

Example: InputStream fis= new FileInputStream("file.txt");

Java clone() method

Example: public class Department implements Serializable { ... } public Department clone() throws CloneNotSupportedException {

Department d = new Department(); d.setName("Department");

Example: public class Department implements Serializable { ... } public Department clone() throws CloneNotSupportedException { Department d = new Department(); d.setName("Department");

Example: public class Department implements Serializable { ... } public Department clone() throws CloneNotSupportedException { Department d = new Department(); d.setName("Department");

Example: public class Department implements Serializable { ... } public Department clone() throws CloneNotSupportedException { Department d = new Department(); d.setName("Department");

<code>assert Equals (expected, actual)</code>	actual «equals» expected
<code>assert Same (expected, actual)</code>	actual== expected (only reference compar- ation)
<code>assert Not Same (expected, actual)</code>	expected != actual (only reference compar- ation)
<code>assert True (condition)</code>	condition
<code>assert False (condition)</code>	!condition
<code>assert Null (value)</code>	value== null
<code>assert Not Null (value)</code>	value!= null
<code>fail()</code>	everytime false
<code>@Test (timeout= 5000)</code>	set test timeout
<code>@Test (expected= IllegalArgumentException.class)</code>	expect exception, if exception is thrown, test passes
<code>@Before public void setUp() { ... }</code>	run this before each test
<code>@After public void tearDown() { ... }</code>	run this after each test

